PAGERANK FOR NETWORKS, GRAPHS, AND MARKOV CHAINS

UDC 519.21

C. ENGSTRÖM AND S. SILVESTROV

Dedicated to Professor Dmitrii Silvestrov on his 70th birthday

ABSTRACT. In this work it is described how a partitioning of a graph into components can be used to calculate PageRank in a large network and how such a partitioning can be used to re-calculate PageRank as the network changes. Although calculating PageRank is considered a problem, it is worth noing that the same partitioning method could be used when working with Markov chains in general or solving linear systems as long as the method used for solving a single component is chosen appropriately. An algorithm for calculating PageRank using a modified partitioning of the graph into strongly connected components is described.

Moreover, the paper also focuses on the calculation of PageRank in a changing graph from two different perspectives, by considering specific types of changes in the graph and calculating the difference in rank before and after certain types of edge additions or removals between components. Moreover, some common specific types of graphs for which it is possible to find analytic expressions for PageRank are considered, and in particular the complete bipartite graph and how PageRank can be calculated for such a graph. Finally, several open directions and problems are described.

1. INTRODUCTION

PageRank is a modern information technology age application of the theory of Markov chains or more generally Perron–Frobenius theory for non-negative matrices in the novel context of internet generated information networks and linked information resources and similar linked structures in databases and other big data [2]. PageRank is a method for ranking nodes in network structures such as pages of the Internet in order of "importance" given the link structure of the complete system. The PageRank method was invented by the founders of the Google search engine in order to improve the quality of ranking of search results using the ranking of web pages on the internet by using the links structure (graph) of the network to measure the relative importance or relevance of the nodes or parts of the webpages network [8].

This application and technology turned out to be very stimulating as a resource of new interesting directions and open problems in mathematics and development of algorithms in networks analysis in general. It is important that the method is extremely fast since there are a huge number of Internet pages. It is also important that the algorithm helps to return the most relevant search results first since very few people will look through more than a couple of pages when doing a search using a search engine [2, 8].

While PageRank as such was developed as a method constructed for ranking webpages on the Internet with the goal of improving performance of search engines using network

structure, PageRank-like methods and ideas have been also considered in other areas, such as ranking in P2P networks, citation ranking, ranking and classification of big data, communication networks, social networks, biologic and medical data networks, natural language processing, economics and resource optimization and distribution networks, logistic networks, traffic planing networks and many other applications.

Computing values of PageRank for nodes in a networks is a complicated mathematical and numerical problem amounting to computing an eigenvector with largest eigenvalue for huge stochastic, substochastic or more general non-negative matrices, or when put in the context of Markov chains, to computation of the stationary distribution vector. The arising problems are complicated due to both the specifics of the structure and the huge size of networks.

Usually, a simple iterative algorithm such as the Power method is used for very large systems. The convergence speed of the Power method and its dependence on certain parameters have been studied to some extent both theoretically and experimentally. For example the Power method on a huge graph structure such as that created by the Web was noticed experimentally to converge with a convergence rate of c, where c is one of the parameters used in the definition [19], and the problem is well conditioned unless c is very close to 1 [21].

Since the number of pages on the Web or in other big data networks is huge, further work has been done in trying to improve the computation time of PageRank even further. One example is by aggregating webpages that are "close" and are expected to have a similar PageRank as in [20]. Another method, used to speed up calculations, is found in [22] where they do not compute the PageRank of pages that have already converged in every iteration. Other methods to speed up calculations include removing "dangling nodes" before computing PageRank and then calculating them at the end or exploring other methods such as using a power series formulation of PageRank [2]. There are also works done on the large scale using PageRank and other measures in order to learn more about the Web, for example looking at the distribution of PageRank both theoretically and experimentally such as in [9].

While the theory behind PageRank is well known in the framework of Perron–Frobenius theory for non-negative matrices and the theory of Markov chains [6,7,17,24], how PageRank is affected from changes in the network system graph structure or in weights or other parameters in the network graph is not as well understood and clearly is an important direction in connection to real networks. In context of Markov chains, it is connected to algorithms for computation of stationary distribution vector and other spectral properties when Markov chain state space and transition matrices are growing or decreasing in size in special ways or when parts of the transition matrix are modified or neglected induced from various kinds of changes or re-prioritizing of nodes, parts or links in networks (graphs). This article can be viewed as further contribution towards this new and interesting direction.

PageRank was originally defined by S. Brin and L. Page as the eigenvector to the dominant eigenvalue of a modified version of the adjacency matrix of a graph [8].

Definition 1.1. PageRank $\vec{R}$ for the vertices in a graph $G := (V; E)$ is defined as the (right) eigenvector with eigenvalue one to the matrix

$$(1) \qquad \mathbf{M} = c \left(\mathbf{A} + \vec{g}\vec{w}^{\top} \right)^{\top} + (1 - c)\vec{w}\vec{e}^{\top},$$

where $\mathbf{A}$ is the adjacency matrix weighted such that the sum over every non-zero row is equal to one, $\vec{g}$ is a vector with zeros for vertices with outgoing edges and 1 for all vertices with no outgoing edges, $\vec{w}$ is a non-negative vector with norm $\|\vec{w}\|_1 = 1$, $\vec{e}$ is a

one-vector and $0 < c < 1$ is a scalar. All vectors are of length n and all matrices of size $n \times n$.

The original normalized version of PageRank has the disadvantage in that it is harder to compare PageRank between graphs or components. Because of that we use a non-normalized version of PageRank as described in for example [11].

Definition 1.2. Consider a random walk on a graph described by $\mathbf{A}$, which is the adjacency matrix weighted such that the sum over every non-zero row is equal to one. In each step with probability $0 < c < 1$, move to a new vertex from the current vertex by traversing a random outgoing edge from the current vertex with probability equal to the weights on corresponding edge weight. With probability $1 - c$ or if the current vertex has no outgoing edges we stop the random walk. Then PageRank $\vec{R}$ for a single vertex v_j can be written as

$$
(2) \qquad R_j = \left(w_j + \sum_{v_i \in V, v_i \neq v_j} w_i P_{ij} \right) \left(\sum_{k=0}^{\infty} (P_{jj})^k \right) ,
$$

where P_{ij} is the probability to hit vertex v_j in a random walk starting in vertex v_i. This can be seen as the expected number of visits to v_j if we do multiple random walks, starting in each vertex once and weighing each of these random walks by $\vec{w}$.

Note that although this definition makes no special treatment of dangling vertices, the normalized and non-normalized definitions of PageRank are still proportional to each other and hence give the same rankings [11].

The rest of this paper is organized as follows. In Section 2 we describe an algorithm using a modified partitioning of the graph into strongly connected components to calculate PageRank. We start by describing the graph partitioning and how such a partitioning can be used to calculate PageRank. Later in Subsections 2.1 to 2.4 we describe the algorithm and the different parts in more detail. After a short theoretical look at the time complexity of the algorithm in Subsection 2.5 we take a look at some results using the method in Subsection 3.1.

The second half of the paper is focused on the calculation of PageRank in a changing graph from two different perspectives. First we take a look at specific types of changes in the graph in Section 4 where we calculate the difference in rank before and after certain types of edge additions or removals between components. In Section 5 we instead look at common specific types of graphs, for which it is possible to find analytic expressions for PageRank. In particular we will look at the complete bipartite graph and how PageRank can be calculated for such a graph.

Finally, we give some conclusions and discuss some open problems yet to be solved related to the subject.

2. Algorithm

In this section we will describe a way to calculate PageRank by first partitioning the graph into components and calculate PageRank for each component individually. We will start by defining what we mean by a connected acyclic component.

Definition 2.1. A connected acyclic component (CAC) of a directed graph G is a subgraph S of G such that no vertex in S is part of any non-loop cycle in G and the underlying graph is connected. Additionally any edge in G that exists between any two vertices in S is also a part of S. A vertex in the CAC with no edge to any other edge in the CAC we call a leaf of the CAC.

CACs can be seen as a connected collection of 1-vertex SCCs forming a tree. While CACs keep the property that all internal edges between vertices in the component are preserved from those in the original graph, it is not maximal in the sense that no more vertices could be added to the component as is the case for SCCs. The reason for this is that we want to be able to create a graph partitioning into components in which the underlying graph is a directed acyclic graph (DAG) in the same way as for the ordinary partition into SCCs.

Definition 2.2. Consider a graph G with partition P into SCCs and CACs such that each vertex is part of exactly one component and the underlying graph created by replacing every component with a single vertex. If there is an edge between any two vertices between a pair of components, then there is an edge in the same direction between the two vertices representing those two components as well. Consider the case where the underlying graph is a DAG (such as for the commonly known partitioning of a graph into SCCs).

- The level L_C of component C is equal to the length of the longest path in the underlying DAG starting in C.
- The level L_{v_i} of some vertex v_i is defined as the level of the component for which v_i belongs ($L_{v_i} \equiv L_C$, if $v_i \in C$).

We note that an SCC made up of only a single vertex is also a CAC; in our work it will be easier to consider these components CACs rather than SCCs. We also note that while a single vertex can only be part of a single SCC, it could be part of multiple CACs of different size. There is however a unique graph partitioning into SCCs and CACs as seen below.

Theorem 2.1. *Consider a directed graph G with a partition into SCCs. Let the underlying graph be the DAG constructed by replacing every component with a single vertex. If there is an edge between any two vertices between a pair of components, then there is an edge in the same direction between corresponding vertices in the underlying DAG. To each vertex in the underlying DAG we attach a level equal to the longest existing path from this vertex to any other vertex in the underlying graph. Next we start to merge SCCs consisting of a single vertex into CACs under the following conditions.*

- *We start merging from the lowest level (vertices in the DAG with no edge to any other vertex in the DAG) and only start merging on the next level when we cannot merge any more components on the current level.*
- *All merges are done by merging a single 'head' 1-vertex CAC of level L containing vertex v with all CACs of level $L-1$ to which there is an edge from v, unless v has an edge to at least one SCC (of more than 1 vertex) of level $L-1$ in which case no merge is made. If a merge takes place, then the level of the new merged CAC is $L-1$.*

Then the following holds:

(1) *This gives a unique partitioning of the graph into SCCs and CACs and does not depend on the order in which we apply merges of 'head' components on the same level.*

(2) *This partition of SCCs and CACs can also be seen as a DAG where we attach a level to each vertex equal to the longest existing path from this vertex to any other vertex in the DAG.*

Remark. Note that after a merge some vertices with a level higher than the one where the merge was made might get a lower level compared to before.

Proof. That a directed graph can be partitioned into SCCs is a well-known and easy to show result from graph theory. Applying a level to the vertices in the DAG is nothing other than a topological ordering of the vertices in the graph, something also well known, hence we start at the merging. Obviously all 1-vertex SCCs are also 1-vertex CACs since any vertex that is part of any (non-loop) cycle must be part of a SCC of more than one vertex.

Since the head CAC of a merge is always connected with each other CAC that is part of a merge, the subgraph representing the component is connected as well. It is also still obviously acyclic since no vertex of any of the CACs is part of any cycle in G from the definition of a CAC. Adding all edges between the head and all other merged CACs also ensures that there is no missing edge between any two vertices of the new CAC. This holds since there can be no edges between any two components on the same level. Hence we can conclude that merging CACs creates a new CAC.

Next we prove statement (1) that the given partitioning is unique and does not depend on the order of merges. CACs are created using a bottom-up approach and it is clear that the level of a CAC never change after its first merge. This means that the level of a CAC is uniquely defined by the level of any leaf in the CAC. All the leaves of a CAC are those 1-vertex CACs which could not be the head of any merge as well because they have either no outgoing edges or they have at least one outgoing edge to a SCC (of more than 1 vertex) of the next lower level.

Assume we have done all merges with head CACs of level L. Consider a 1-vertex CAC with vertex v and level $L + 1$ and edges to one or more CACs but no SCC of more than one vertex of level L (or higher). From the previous argument we get that the level of any CAC linked to by v will not change from any future merges. This means that eventually v will be part of the same CAC as all vertices parts of any CAC with level L to which there is an edge from v regardless of which order merges are made on level $L + 1$.

Repeating this argument for all 1-vertex CACs of level $L + 1$ we get that for each such vertex v the neighboring vertices part of a CAC with a lower level, for which v should be part of the same CAC, is uniquely determined after all the merges on level L. Repeating this for all levels gives us a unique partitioning. This proves statement (1).

Last we show statement (2) by showing that merging of components does not create any cycle in the underlying DAG created by the components.

Consider a merge of head CAC with vertex v, level L and an edge to each of n CACs $C_1, C_2, \ldots, C_n$ with level $L - 1$. Since all CACs $C_1, C_2, \ldots, C_n$ have the same level, the resulting CAC after merging can only have edges to components of level at most $L - 2$ since we merge it with all CACs of level $L - 1$ to which there is an edge from v, but do not merge if there is an edge from v to any SCC of level $L - 1$ and there can be no edges between any components of the same level from the initial SCC partitioning. Since initially there can be no edges from any component to another of the same or larger level and we do not create any such edges when doing a merge, we do not create any cycle in the underlying DAG.

Since we only merge CACs, and merges do not create any cycles in the DAG, we have proved that the new partitioning can also be represented by a DAG where each vertex corresponds to one SCC or CAC. This proves statement (2). □

Corollary 2.1. *If a directed graph G has a SCC partitioning with maximum level L_{SCC} and a SCC/CAC partitioning with maximum level $L_{SCC/CAC}$, then*

$$L_{SCC/CAC} \leq L_{SCC}.$$

Proof. Every merge lowers the level of the 'head' vertex and possibly one or more components of a higher level. This makes it easy to find a graph such that $L_{SCC/CAC} < L_{SCC}$ such as the 2-vertex graph with a single edge between them in one direction. Similarly we can easily find a graph for which equality holds (such as the single vertex graph). However, since doing a merge can never result in any vertex getting a higher level, $L_{SCC/CAC} \leq L_{SCC}$. □

For algorithms (such as PageRank), which can be done on the components of one level at a time, in parallel the SCC/CAC partitioning have the advantage over the usual SCC partitioning in that it generally creates a lower number of levels and thus a larger amount of vertices (but not necessary components) on the same level in the new partitioning on average. In effect this reduces the chance of bottlenecks where we have a level with only a single or a few small components. The only components that get larger are the CACs, which are acyclic apart from any loops and thus often have specialized faster methods (for example by exploiting the triangular adjacency matrix). Thus increasing the size of these components is often not as much of an issue and might in fact often be beneficial instead by reducing overhead.

An example of a directed acyclic graph and its SCC/CAC partitioning can be seen in Figure 1.
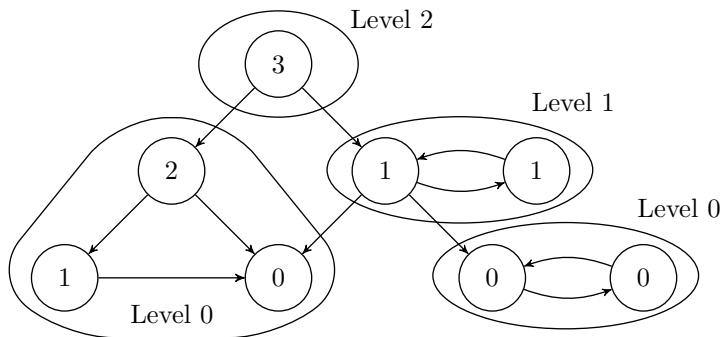

FIGURE 1. Example of a graph and corresponding components from SCC/CAC partitioning of the graph (2 SCCs, 1 CAC and 1 1-vertex component). Vertex labels denote the level of each vertex if we had only partitioned the graph into SCCs (for the SCC/CAC partitioning the vertex levels is the same as the level of corresponding component in the figure.)

From this figure it is clear why we cannot merge when the 'head' has an edge to any SCC of the next lower level. If the top (level 2) component merged with the left (level 0) component, then this would have created a cycle in the underlying graph. It is also possible to see how merging some components can result in the partitioning getting a lower max-level, the SCC/CAC partitioning have only 3 levels while the SCC partitioning would need 4 levels.

In [10] we showed how to calculate PageRank for the five different types of vertices defined below.

Definition 2.3. For the vertices of a simple directed graph we can define 5 distinct groups, $G_1, G_2, \ldots, G_5$.

(1) G_1: Vertices with no outgoing or incoming edges.

(2) G_2: Vertices with no outgoing edges and at least one incoming edge (also called dangling vertices).

(3) G_3: Vertices with at least one outgoing edge but no incoming edges (also called root vertices).

(4) G_4: Vertices with at least one outgoing and incoming edge, but which is not part of any (non-loop) directed cycle (no path from the vertex back to itself apart from the possibility of a loop).

(5) G_5: Vertices that are part of at least one non-loop directed cycle.

Qing Yu et al. gave a similar but slightly different definition of 5 (non-distinct) groups for vertices, namely dangling and root vertices (G_2 and G_3), vertices that can be made into dangling or root vertices by recursively removing dangling or root vertices (part of G_4) and the remaining vertices (part of G_4 and G_5) [32]. Given PageRank of a vertex not part of a cycle (groups 1–4), then the PageRank of other vertices can be calculated by removing the vertex and modifying the initial weight of other vertices.

Theorem 2.2. *Given PageRank $\vec{R}_g^{(3)}$ of vertex v_g where v_g is not part of any non-loop cycle, the PageRank of another vertex v_i, from which there exist no path to v_g, can be expressed as*

$$(3) \quad R_i = \left(W_i + R_g ca_{gi} + \sum_{\substack{v_j \in V \\ v_j \neq v_i, v_g}} (W_j + R_g ca_{gj}) P(v_j \to v_i) \right) \left(\sum_{k=0}^{\infty} (P(v_i \to v_i))^k \right),$$

where ca_{gi} is the one-step probability to go from v_g to v_i.

The proof with minor modifications is similar to the one found in [10], where it is formulated for vertices in G_3 on graphs with no loops.

Proof. Consider R_g from Definition 1.2. Since we know that there is no path from v_i back to v_g (or v_g would be part of a non-loop cycle) we know that the right-hand side will be identical for all other vertices. We rewrite the influence of v_g using

$$(4) \quad R_g P(v_g \to v_i) = R_g ca_{gi} + \sum_{\substack{v_j \in V \\ v_j \neq v_i, v_g}} R_g ca_{gj} P(v_j \to v_i).$$

We can now rewrite the left sum in Definition 1.2:

$$(5) \quad \sum_{v_i \in V, v_i \neq v_j} W_i P(v_i \to v_j) = R_g ca_{gi} + \sum_{\substack{v_j \in V \\ v_j \neq v_i, v_g}} (W_j + R_g ca_{gj}) P(v_j \to v_i),$$

which when substituted into (2) proves the theorem. □

It is also easy to show that any SCC can also be divided into one of the first four groups if we consider each SCC as a vertex in the underlying DAG (a SCC can never be part of a cycle). The important part of this is that it is also possible to calculate PageRank one component at a time rather than for the whole graph at once.

Corollary 2.2. *Let $\vec{R}_{L+}$ be the PageRank of all vertices belonging to components of level L or greater. Then the PageRank of a vertex v_i belonging to a component of level $L-1$*

can be computed by

$$R_i = \left(\sum_{k=0}^{\infty} \left(P(v_i \to v_i) \right)^k \right)$$

$$\times \left(\left(W_i + c \left(\vec{R}_{L^+} \right)^\top \vec{a}_{L^+,i} \right) + \sum_{\substack{v_j \in V \\ v_j \neq v_i}} \left(W_j + c \left(\vec{R}_{L^+} \right)^\top \vec{a}_{L^+,j} \right) P(v_j \to v_i) \right),$$

where $\vec{a}_{L^+,i}$ is a vector containing all 1-step probabilities from vertices of level L or greater to vertex v_i.

Proof. The proof follows immediately from Theorem 2.2 by replacing the rank of a single vertex with the sum of the rank of all vertices belonging to components of a higher level. Those in lower level components or other components on the same level do not affect the rank since they automatically do not have any path to v_i. $\square$

Using Corollary 2.2 it is clear that after calculating the PageRank of all vertices belonging to components of level L and above we can calculate those of level $L - 1$ by first changing their initial weight and then considering the component by itself. In matrix notation we can update the weight vector for all components of lower level by calculating

$$(6) \qquad \vec{W}_{L-1}^{new} = \vec{W}_{L-1}^{old} + c\mathbf{A}_{L^+,L-1}\vec{R}_{L^+},$$

where $\mathbf{A}_{L^+,L-1}$ corresponds to the submatrix of A with all rows corresponding to vertices of level L or greater and all columns of level $L - 1$. This is essentially the same method which is used in [26], but here we have formulated it for any component instead of for dangling vertices (vertices with no outgoing edges).

2.1. **Method.** The complete PageRank algorithm can be described in three main steps.

(1) Component finding: Finding the SCC/CAC partitioning of the graph.
(2) Intermediate step: Create relevant component matrices and weight vectors.
(3) PageRank step: Calculate PageRank one level at a time and components on the same level one at a time or in parallel.

In order to be able calculate PageRank for each component we obviously first need to find the components themselves. This is done in the component finding part of the algorithm where we find a SCC/CAC partitioning of the graph as well as the level of each component. By using the CAC/SCC partitioning rather than the usual SCC partitioning we reduce the risk of having very few vertices on the same level. The aim of this is to be able to avoid some of the disadvantages with some other similar methods such as the one in [25], where a large number of small levels (small diagonal blocks) increases the overhead cost [32]. This step is similar to the initial matrix reordering made by [4]. However instead of only finding a partial ordering, we have modified the depth first search slightly in order to identify components that can be calculated in parallel as well as group 1-vertex components on the same level together. Another advantage is that different methods can be used for different types of components as we will see later. The component finding step is described in Subsection 2.2.

In the intermediate step the data (edge list, vertex weights) need to be managed such that the individual matrices for every component can quickly and easily be extracted. This section of the code can vary a lot between implementations and is one of the main contributors of overhead in the algorithm. The SCC/CAC partitioning can easily be transformed into a permutation matrix and used to permute the graph matrix and then

solve the resulting linear system; this can be seen as an alternative to the recursive reordering algorithm described in [25]. This step is described in Subsection 2.3.

After the intermediate step we are ready to start calculating PageRank of the vertices. This is done one level at a time starting with the highest and modifying vertex weights between levels using (6). Components on the same level can use different methods to calculate PageRank and can either be calculated sequentially or in parallel. The PageRank step is described in Subsection 2.4.

2.2. **Component finding.** The component finding part of the algorithm consists of finding an SCC/CAC partitioning of the graph as well as the corresponding levels of the components. Since any loops in the graph have no effect on which SCC or CAC a vertex is part of, these will be ignored in the component finding step. Finding the components and their level can be done through a modified version of Tarjan's well-known SCC finding algorithm using a depth first search [31]. For every vertex v we assign six values:

- v.index containing the order in which it was discovered in the depth first search;
- v.lowlink for an SCC representing the lowest index of any vertex we can reach from v, or for a CAC representing the 'head' vertex of corresponding component;
- v.comp representing the component the vertex is part of, assigned at the end of the component finding step;
- v.depth used to implement efficient merges of components. It can be removed if the extra memory is needed, but it could result in slowdown because of merges for some graphs;
- v.type indicates if v is part of an SCC or a CAC (1-vertex SCCs are considered CACs);
- v.level indicating the level of the component to which v belongs.

Of these the first three can be seen in Tarjan's algorithm as well, and play virtually the same role here (although in Tarjan's the comp value can be assigned as components are created). During the depth first search each vertex v goes through three steps in order.

(1) Discover: Initialize values for the vertex.
(2) Explore: Visit all neighbors of v, finishing the depth first search (DFS) of any unvisited neighbors before going to the next. After a vertex is visited we update v.lowlink and v.level.
(3) Finish: After all neighbors are visited we create a new component if appropriate. If a CAC is created we also check for and do any merge with v as head.

During the discover step values are initialized after which the vertex is put on the stack. The index and lowlink values are initialized using a counter starting at 1 and increasing by one for every new vertex we discover while level and depth values are initialized to 1 (type and level does not need to be initialized). The explore step as well works much like Tarjan's depth first search except that it also updates the level of the vertex we are exploring.

- Loop through all neighbors by doing the Discover, Explore and Finish steps before going to the next.
- For any neighbor w that is part of a formed component (i.e., not part of the same SCC as v) we update v.level to the max of the old level and the level of any such neighbors plus one.
- For neighbors that are not part of a formed SCC (i.e. part of the same component) we we update v.level and v.lowlink to the max of the old level and lowlink and the level or lowlink of any such neighbors respectively.

The last step in the DFS is where we evaluate if a new component should be created and handle any merges needed with this vertex as 'head' component. The initial component is created in the same way as in Tarjan's algorithm: if v.lowlink = v.index we create an SCC by popping vertices from the stack until we pop v from the stack.

If the vertex belongs to a component of size one, then we also check if any merges into a CAC should take place by checking if the type of all neighbors of one level lower belong to a CAC (or is a single vertex). In case this is true we merge the vertex and all such CACs into a single CAC component. The components are stored in a merge-find data structure through the .lowlink and .depth attribute. A merge-find data structure allows us to do the two operations we need: merging two components and finding a 'head' vertex representing a component (used in merge, and when finally assigning the .comp attribute). This way both operations can be done in constant amortized time ($O(\alpha(|V|))$) as well as requiring very little memory.

In Tarjan's algorithm you don't need the .depth values since the .comp value can be assigned while creating a component. The reason we do not do it here is because it cannot be updated efficiently when doing a merge of two CACs.

Looking at the computational complexity of the component finding step we see that discover, explore and finish are all done once for every vertex; of these discover is obviously done in constant time. During explore we will eventually have to go through all edges exactly once but all operations take only constant time. Lastly, finish is called once for every vertex doing $O(\alpha(|V|))$ work for the SCC creation part (amortized constant because we do merges rather than assigning component values directly). When checking for merges of CAC we will at most visit every edge once (over all vertices) doing $O(\alpha(|V|))$ work. Thus in total we end up with $O(|V|+|E|+|V|\alpha(|V|)+|E|\alpha(|V|)) \approx O(|E|\alpha(|V|))$, if $|E| > |V|$, in other words linear amortized time in the number of edges.

Before returning the results $|V|$ find operations also need to be done in order to assign the .comp value for each vertex. Since the find operation in a merge-find data structure takes $O(\alpha(|V|))$ time, this takes $O(|V|\alpha(|V|))$ time in total.

Hence the complete component finding algorithm takes $O(|E|\alpha(n))$ time which is comparable to Tarjan's which can be implemented in $O(|E|)$ time. We note that if the .depth value is ignored everything works, but the merges are no longer guaranteed to be made in constant amortized time. If memory is a concern or if the size of the CACs are assumed to be small, it might be worthwile to work without the .depth value even though merges could be slow in the worst case.

2.3. **Intermediate step.** This part is responsible for organizing the data in such a way that we can quickly construct corresponding matrices and continue with PageRank calculations of components effectively.

We sorted components first in order of level and second in the size of the component (both descending order) so that we can work on one component at a time starting with the largest on every level. Similarly we grouped edges belonging to the same component together and those between components together depending on the level of the source vertex.

This part of the code is highly dependent on the choice of programming language chosen; hence we will not cover it in detail here.

It is worth noting that in our implementation this section of the code contains much of the extra overhead needed for our method (more than the previous component finding step itself).

2.4. **PageRank step.** Now that all preliminary work is done we can start the actual PageRank calculation where we calculate PageRank for all vertices of one level at a time (starting by the largest). The PageRank step can be described by the following steps:

(1) Initiate L to the maximum level among all components.
(2) For each component of level L: pick a suitable method and calculate PageRank.
(3) Update weight vector V for all remaining components (of lower level).
(4) If $L > 0$, decrease L by one and go to step 2; otherwise we are finished.

Depending on the type and size of the component PageRank, we calculate PageRank in one of four different ways:

- Component is made up of a collection of single vertex components: PageRank of any such collection of 1-vertex components is the initial weight w_i for vertices with no loop and $w_i/(1 - ca_{ii})$ for any vertex with a loop, where a_{ii} is the weight on the loop.
- CAC of more than one vertex: Calculate PageRank using a slightly modified breadth first search (BFS).
- SCC but small (for example less than 100 vertices): Calculate PageRank by directly solving the linear system $\left(\mathbf{I} - c\mathbf{A}^\top\right) \vec{R} = \vec{W}$ (using LU factorization).
- SCC and large: Use the iterative method, in our case using a power series.

Out of these the first one is done in $O(|V|)$ time (copy the weight vector) or $O(1)$ if no separate vector is used for the resulting PageRank and we assume there are no loops, and the second and fourth is done in $O(|E|)$. However the coefficient in front of the first is much lower since it is guaranteed to only visit every edge once, while the iterative method needs to visit every edge in every iteration (number of which depend on error tolerance and method chosen). The third is done in $O(|V|^3)$ using LU factorization. However since $|V|$ is small this is still faster than the iterative method unless the error tolerance chosen is large.

We also note that only the fourth method actually depends on the error tolerance at all; every other method can be done in the same time regardless of error tolerance (down to machine precision).

After we have calculated PageRank for all components on the current level we need to adjust the weight of all vertices in lower level components as shown in (6). This can be done using a single matrix-vector multiplication using the edges between the two sets of components

$$\vec{W}_{(L-1)^-}^{\text{new}} = \vec{W}_{(L-1)^-}^{\text{old}} + \mathbf{M}_{L^+,(L-1)^-}^\top \vec{R}_L.$$

This is the same kind of correction as is done in for example [2] and for the non-normalized PageRank used here in [10].

In the weight adjustment step every edge is needed once if it is an edge between components and is never needed if it is an edge within a component. Hence if we look over the whole algorithm: every edge is visited at most twice in the DFS, then every edge that is not part of an SCC is visited exactly once more (either as part of a CAC or as an edge between components), while those that are part of an SCC are typically visited a significantly larger number of times depending on algorithm, error tolerance and convergence criterion used. Of course there is also some extra overhead that would need to be taken into consideration for a more in-depth analysis.

We note that calculating PageRank for all components on a single level can be computed in parallel (hence why we sort them by their size starting with the largest). The weight adjustment can either be done in parallel for each component or as we have done

here, once for all components of the same level. In case there is a single very large component on a level it might be more appropriate to instead do it one component at a time to reduce the time waiting for the large component to finish.

2.5. **Error estimation.** Looking at the error we have two different kinds of errors to consider, first errors from the iterative method used to calculate PageRank of large SCCs (depending on error tolerance) and second any errors because of errors in data ($\mathbf{M}$ or $\vec{W}$). We will mainly concern ourselves with the first type which is likely to dominate unless the error tolerance is very small.

We start by looking at a single isolated component. If this component is a CAC or a small SCC we calculate PageRank analytically and errors can be assumed to be small as long as an appropriate method is used to solve the linear system for the small SCCs using for example LU-decomposition. For large SCCs we stop iterations after the maximum change of rank for any vertex between any two iterations is less than the error tolerance (tol). Since the rank is monotonically increasing we can be sure that the true rank is always a little higher in reality than what is actually calculated.

The true rank can be described by $\vec{R} = \sum_{k=0}^{\infty} \mathbf{M}^k \vec{w}$, where we let $\vec{p}_k = \mathbf{M}^k \vec{w}$. Since every row sum of $\mathbf{M}$ is less than or equal to c, one has $c|\vec{p}_{k-1}| \geq |\vec{p}_k|$. This means that the maximum change in rank over all vertices in the graph after K iterations is bounded by

$$\sum_{k=1}^{\infty} c^k |\vec{p}_K| = \frac{c|\vec{p}_K|}{1-c} \approx 5.66|\vec{p}_k|, \qquad c = 0.85.$$

This does not change if there are any edges to or from other components in the graph although the difference will be spread over a larger amount of vertices if there are edges from the component. There might also be additional additive error from components with edges to the single component we are considering. Over all vertices and all components we can estimate bounds for the total error ε_{tot} over all vertices as well as the average error ε_{avg} given some error tolerance tol:

$$\varepsilon_{tot} < |\mathrm{SCC}_l| \cdot \mathrm{tol} \, \frac{c}{1-c},$$

$$\varepsilon_{avg} < \frac{|\mathrm{SCC}_l|}{|V|} \cdot \mathrm{tol} \, \frac{c}{(1-c)} \leq \mathrm{tol} \, \frac{c}{(1-c)},$$

where $|\mathrm{SCC}_l|$ is the number of vertices part of a 'large' SCC (for which we need to use an iterative method) and $|V|$ is the total number of vertices in the graph. It should be noted that this estimate is likely to be many times larger than in reality unless all the vertices have approximately the same rank. Given that PageRank for many real systems approximately follows a power distribution [5], most vertices will have orders of magnitude smaller change in rank when finally those with a very high rank have a change smaller than the error tolerance. Additionally if the graph contains some dangling vertices (vertices with no outgoing edges), then these will further reduce the error.

3. Experiments

Our implementation of the algorithm is done in a mixture of c and c++ for the graph search algorithms (component finding and CAC-PageRank algorithm) and Matlab for the ordinary PageRank algorithm for SCCs and weight adjustment as well as the main code gluing the different parts together. The reason for using c/c++ for some parts is that while Matlab is rather fast at doing elementary matrix operations (PageRank of SCCs and weight adjustment), it is very slow when you attempt to do for example DFS or BFS on a graph.

- Component finding: Implemented in c++ as a variation of the depth first search in the boost library. The method is implemented iteratively rather than recursively (hence it can handle large graphs which could otherwise give a very large recursion depth).
- CAC-PageRank algorithm: Implemented in c using a modified BFS.
- Power series PageRank algorithm: Implemented in Matlab. Used for SCCs as well as on the whole graph for comparison.
- Main program: Implemented in Matlab, with c/c++ parts used through mex files.

3.1. **Experiments.** For evaluation of the method we have used a graph released by Google as part of a contest in 2002 [1], part of the collection of datasets maintained by the SNAP group at Stanford University [27]. This graph contains 916428 vertices of which 399605 are part of a CAC and 5105039 edges; there are 302768 1-vertex CACs out of 321098 CACs in total as well as 12874 SCCs. Maximum component size is 434818 vertices (SCC). This graph has both the scale-free and small-world properties making it a good example of the type of graph we would be interested in to calculate PageRank in real applications.

All experiments are performed on a computer with a quad core 2.7Ghz (core)-3.5Ghz (turbo) processor (Intel(R) Core(TM) i7-4800MQ) using Matlab R2014a with four threads on any of the parts computed in parallel. Three different methods were used:

(1) Calculate PageRank as a single large component using a power series.
(2) Using the method described in Subsection 2.1 with components on the same level calculated sequentially.
(3) Using the method described in Subsection 2.1 with components on the same level calculated in parallel.

We note that the intermediate step between methods 2 and 3 differs. Because of limitations in how the parallelization can be implemented, we had to separate edges of the same component into their own cell-array for the parallel versions. This accounts for the main difference in overhead between methods 2 and 3. The intermediate step is not parallelized (in either version), for the parallel method this adds a significant amount of extra overhead.

After doing the SCC/CAC partitioning of the graph and sorting all components according to their level and component size (both descending order), we can visualize the non-zero values of this new reordered adjacency matrix. The density of non-zeros before and after reordering for the Web graph can be seen in Figure 2.

Note that the diagonal lines are not single vertex components but rather a large amount of small components on the same level (hence they can be computed in parallel). Any 1-vertex components are not colored since they have no internal edges, and two large section of 1-vertex components are right before the middle large component and in the bottom right corner of the matrix.
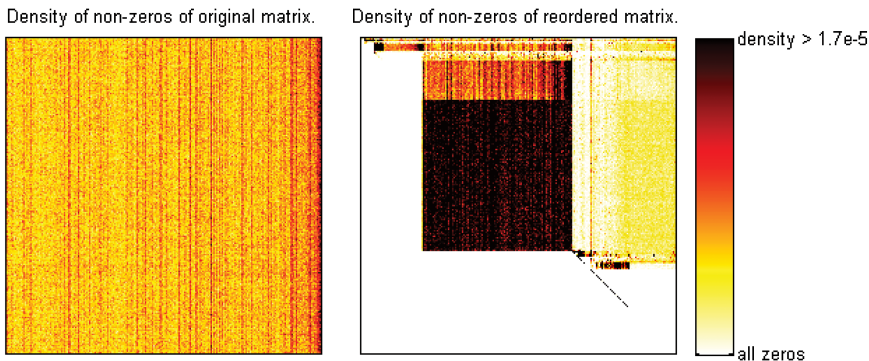
FIGURE 2. Non-zero values of adjacency matrix for the Web graph be-
fore and after sorting vertices according to level and component

After finding the SCC/CAC partitioning large sections of zeros can clearly be seen, something which is not present in the original matrix. The single very large component in the graph is seen in the middle of the matrix, with a section of small components both above and below it.

Langville and Meyer do a similar reordering by recursively reordering the vertices by putting any dangling vertices last and not considering edges to those already put last once for any further reordering of remaining vertices [25]. This effectively creates one or more CACs along with one large component. The advantage of our approach compared to this is that we can also find components above the single large component rather than combining them into a single even larger component as well as finding sets of components which can be computed in parallel.

The total number of levels in the Web graph was 28, with the majority being located right at the very top or right after the large central component. Because of their small size it might be desirable to merge some of these components if this proves to happen consistently. If no merging of 1-vertex CACs was done (using the ordinary SCC partitioning) the number of levels was increased to 34 levels instead.

Since PageRank of different SCCs converge in varying amounts of iterations it is also of interest to see how the number of iterations for different components varies, as well as how it compares to the number of iterations needed by the basic algorithm where we calculate PageRank as if the graph was a single component. The number of iterations for all SCCs of more than 2 vertices of the Web graph with $c = 0.85$ and tol $= 10^{-9}$ can be seen in Figure 3.

In Figure 3 we can see a couple of things, first the number of iterations over any component is less than the number of iterations that would be needed if we calculated PageRank of the graph as if it were a single huge component. The average number of iterations per edge was 148, which can be compared to the number of iterations for the graph as a single component which was 168; this gives an improvement of approximately 12 %. This might appear small looking at the figure, but remember that the largest components on each level are put first on their level and the size of components approximately follows a power law, hence large parts of the figure represent relatively few vertices. It should also be noted that a significant number of edges (approximately 26 %) lies either between components or within CACs, both of which are not counted for here since they don't use the iterative method and are instead used only once either to modify weights between levels or as part of the DFS when calculating PageRank of CACs.
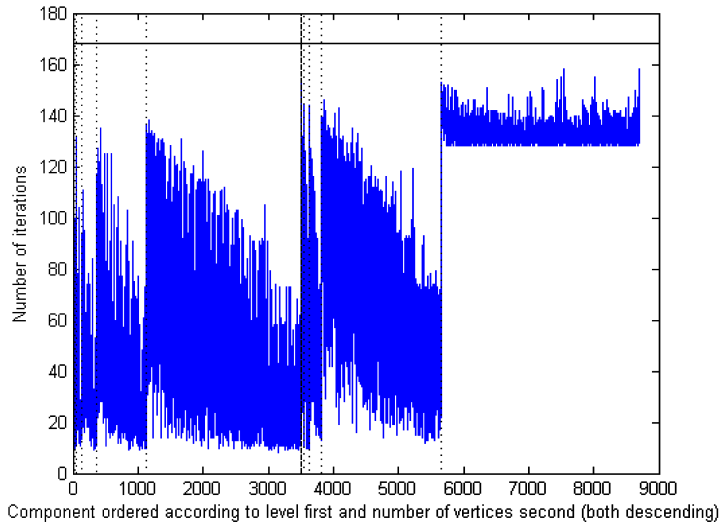
FIGURE 3. The number of iterations needed per SCC ordered according to their level first and number of vertices second (both descending order). The dotted vertical lines denote where one level ends and the next one starts, while the horizontal line denotes the result where the whole graph is considered a single component. $c = 0.85$, tol $= 10^{-9}$.

The second point of interest is that there is a clear difference between components at the last level compared to those of a higher level. Any SCC on the last level is by definition a stochastic matrix (before multiplication with c) since they have no edges to any vertex in any other component. This gives a lower bound on the number of iterations equal to $\log \text{tol}/\log c \approx 128$ easily seen from the relation $c^{\text{itr}} \leq \text{tol}$, where itr is the number of iterations. However those components of higher level are by definition a sub-stochastic matrix (before multiplication with c) since there is at least one edge to some other component. This is equivalent to some vertices having a lower c value, and the algorithm can therefore converge faster.

The third observation is that a large component generally needs more iterations than a smaller component. This makes sense if we consider that as long as most vertices in the component do not contain edges to other components, as the component grows in size at least some part of the component will behave similar to those in the last level and we thus need a larger number of iterations. For large components the estimated number of iterations is usually quite good, while for small components it usually gives too high of an estimate (unless it is part of the last level).

The running time in seconds for the Web graph for the three different methods for different values of error tolerance from 10^{-1} to 10^{-20} can be seen in Figure 4a. From this it is clear that our method adds a significant amount of overhead, especially the parallel one. While all three methods need a longer time if the error tolerance is smaller, our method shows a significantly smaller increase compared to the basic method. The break here even seems to lie at around 10^{-5} for the sequential algorithm and at around 10^{-9} for the parallel algorithm because of additional overhead. If the overhead in particular for the parallel algorithm could be further reduced, this breakpoint could potentially be significantly earlier. Note that because of limits in machine precision we might not have the correct rank down to the last 20 decimals at the lowest tolerance. However, since
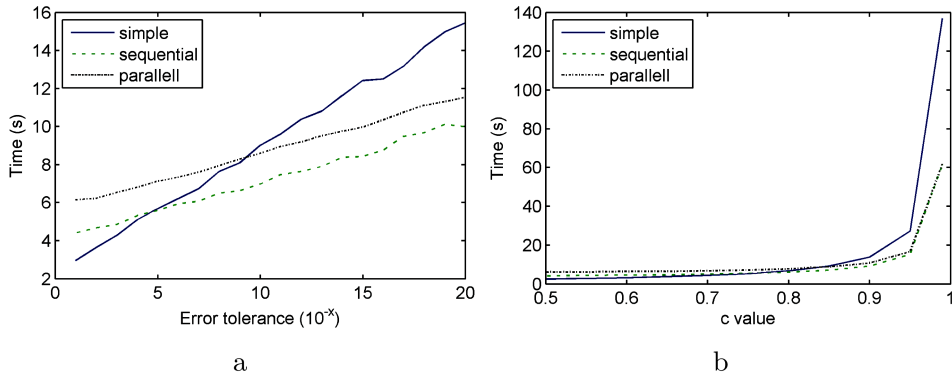
a   b

FIGURE 4. Running time needed to calculate PageRank for 3 differ-
ent methods (a) depending on error tolerance using $c = 0.85$ and (b)
depending on c between 0.5 and 0.99 using tol $= 10^{-10}$ on the Web
graph

we sum over successively smaller parts we still get a good approximation of the actual
computation time.

The running time when we let c vary between 0.5 and 0.99 with a constant error
tolerance (10^{-10}) can be seen in Figure 4b. Overall the results of our approach are
promising; we have gotten better results the bigger the graph is as well as the lower
the error tolerance is as compared to the basic approach. Our implementation of the
parallel method had significantly more overhead than the sequential method. However
this difference could likely be reduced significantly if the algorithm where implemented
in full (with parallelization in mind) in for example c++, where we have more control
over how it can be implemented.

4. CHANGES IN A GRAPH

Whenever the graph changes, such as the addition or removal of edger or vertices,
PageRank in the graph will change as well. When recalculating PageRank after such
changes you would like to use the previous PageRank as much as possible to find the
new rank after the change.

Given a PageRank algorithm similar to the one in Subsection 2.1 where we have
partitioned the graph into SCCs (and CACs or other useful parts) there are many types
of specific changes in the graph that can be handled more efficiently.

In this section we will consider some examples of such changes. Much of the discussion
is based on [12].

4.1. **Changes is personalization vector.** Changes in the personalization vector are
comparatively easy to handle since it does not change any paths in the graph.

Lemma 4.1. *Consider a graph with PageRank $\vec{R}^1$ and weight vector $\vec{w}^1$. Then the new
PageRank $\vec{R}^2$ given a new personalization vector $\vec{w}^2 = \vec{w}^1 + \triangle\vec{w}$ can be written as*

$$(7) \qquad R_j^2 = R_j^1 + \left(\triangle w_j + \sum_{v_i \in S, v_i \neq v_j} \triangle w_i P_{ij} \right) \left(\sum_{k=0}^{\infty} (P_{jj})^k \right).$$

Proof. The proof is very straightforward using the definition and factoring out the old rank:

$$R_j^2 = \left(w_j^1 + \triangle w_j + \sum_{v_i \in S, v_i \neq v_j} (w_i^1 + \triangle w_i) P_{ij} \right) \left(\sum_{k=0}^{\infty} (P_{jj})^k \right)$$

$$= \left(w_j^1 + \sum_{v_i \in S, v_i \neq v_j} w_i^1 P_{ij} \right) \left(\sum_{k=0}^{\infty} (P_{jj})^k \right)$$

$$+ \left(\triangle w_j + \sum_{v_i \in S, v_i \neq v_j} \triangle w_i P_{ij} \right) \left(\sum_{k=0}^{\infty} (P_{jj})^k \right). \qquad \square$$

While making changes to the personalization vector might not be a very large change made in a graph, it is important to note that something similar happens to other unchanged components when you make localized changes in one component. When a change is made to a single high level component, then this can be seen as a change in the personalization vector in all lower level components which can be reached from this component.

Since these changes do not affect the component structure of the graph at all, it means that a previously found graph partition could be utilized, saving time in that part of the algorithm. Even in a single component with a personalization vector change we can expect the method used to calculate PageRank to converge slightly faster than it would otherwise, assuming the change is small in relation to the original personalization vector.

4.2. **Add or remove edges between components.**

(1) The only change to the graph is the addition and/or removal of one or more edges from a single "source" vertex to one or more "target" vertices belonging to other components.
(2) New edges do not create any cycle in the underlying DAG.
(3) The personalization vector remains unchanged in the "source" component.

The first condition is slightly more general than only allowing single edge additions or removals while still limiting the amount of potential change in rank of the source component. The second condition is required in order to make sure that any change in rank only depends on the source and target components and to avoid dependence on the rank of the the target components for the rank of the source component. The second condition is most easily achieved by only allowing new edges to be formed from the source component to lower or same level components. By not allowing changes of the personalization vector for the source component we again limit the amount of change in rank we get in this component. Obviously changes in personalization vector of the source component could be handled by first considering only the change in personalization vector as outlined in the previous section, and then continue as if there was no change in the personalization vector.

For convenience we introduce the following notation:

- $P_{\to j} = w_j + \sum_{v_i \in V, v_i \neq v_j} w_i P_{ij}$.
- $P_{ab}(c)$ is the probability to reach v_b starting in v_a after passing through v_c at least once.
- $P_{ab}(\bar{c})$ is the probability to reach v_b starting in v_a without ever passing through v_c.

Note that using this notation PageRank can also be written as

$$(8) \qquad\qquad R_a = \frac{P_{\to a}}{1 - P_{aa}} .$$

Before looking at the actual problem we will start by formulating the following lemma which states that PageRank can be decomposed into two parts representing all paths that go through or don't go through some vertex v_a.

Lemma 4.2. *PageRank of a single vertex v_b can be written as*

$$
(9) \qquad R_b = \frac{P_{\to b}(\bar{a})}{1 - P_{bb}(\bar{a})} + \frac{R_a P_{ab}(\bar{a})}{1 - P_{bb}(\bar{a})} \ .
$$

This can be seen as a decomposition of paths; the left expression can be seen as the sum of all paths which does not go through vertex v_a and the right side being the sum of all paths going through vertex v_a at least once.

Proof. We start by rewriting PageRank as a sum of all visits to v_b before any visits to v_a + all visits to v_b after 1 visit to v_a but before the second visit to v_a, and so on:

$$
R_b = \frac{P_{\to b}(\bar{a})}{1 - P_{bb}(\bar{a})} + \frac{P_{\to a}P_{ab}(\bar{a})}{1 - P_{bb}(\bar{a})} + \frac{P_{\to a}P_{aa}P_{ab}(\bar{a})}{1 - P_{bb}(\bar{a})} + \frac{P_{\to a}(P_{aa})^2 P_{ab}(\bar{a})}{1 - P_{bb}(\bar{a})} + \cdots .
$$

The second and later expressions can be identified as a geometric sum resulting in

$$
R_b = \frac{P_{\to b}(\bar{a})}{1 - P_{bb}(\bar{a})} + \frac{\sum_{k=0}^{\infty} P_{\to a}(P_{aa})^k P_{ab}(\bar{a})}{1 - P_{bb}(\bar{a})}.
$$

Solving the geometric sum and using (8) completes the proof:

$$
R_b = \frac{P_{\to b}(\bar{a})}{1 - P_{bb}(\bar{a})} + \frac{P_{\to a}P_{ab}(\bar{a})}{(1 - P_{bb}(\bar{a}))(1 - P_{aa})} = \frac{P_{\to b}(\bar{a})}{1 - P_{bb}(\bar{a})} + \frac{R_a P_{ab}(\bar{a})}{1 - P_{bb}(\bar{a})}. \qquad \square
$$

Adding or removing edges between components from a single vertex can be seen as a personalization vector change for the target components as outlined in the previous section. The remaining problem is how this effects the source component itself since changing the number of outgoing edges of a vertex changes the weight on each of those edges as well. This leads us to consider the problem of changing the weights on all outgoing edges from a single vertex as seen in the following theorem.

Theorem 4.1. *Consider a graph with PageRank R^1, and let e_a^1 be the weight of all edges going out of vertex v_a. After changing edge weights on edges out of v_a to e_a^2, then PageRank R_a^2 of vertex v_a and PageRank R_b^2 of any other vertex $v_b \neq v_a$ after the change can be written as*

$$
R_a^2 = \frac{P_{\to a}^1}{1 - P_{aa}^1 \frac{e_a^2}{e_a^1}},
$$

$$
R_b^2 = R_b^1 + \frac{\left(R_a^2 \frac{e_a^2}{e_a^1} - R_a^1 \right) P_{ab}^1(\bar{a})}{1 - P_{bb}^1(\bar{a})}.
$$

Proof. The first statement is easily shown to be correct. First of all $P_{\to a}^1 = P_{\to a}^2$ since edges going out of v_a have no effect on the first hitting probability of v_a in a random walk on the graph. Similarly it is easy to see that the probability of return paths only changes by the new edge weight e_a^2; this gives $P_{aa}^2 = P_{aa}^1 e_a^2/e_a^1$. Putting both of these together proves the first statement.

For the second part we start by decomposing PageRank using Lemma 4.2:

$$
R_b^2 = \frac{P_{\to b}^2(\bar{a})}{1 - P_{bb}^2(\bar{a})} + \frac{R_a^2 P_{ab}^2(\bar{a})}{1 - P_{bb}^2(\bar{a})} = \frac{P_{\to b}^1(\bar{a})}{1 - P_{bb}^1(\bar{a})} + \frac{R_a^2 P_{ab}^1(\bar{a}) \frac{e_a^2}{e_a^1}}{1 - P_{bb}^1(\bar{a})} \ .
$$

Here the second equality is found by realising that $P_{\to b}^2(\bar{a}) = P_{\to b}^1(\bar{a})$ and $P_{bb}^2 = P_{bb}^1$ since we skip the paths through v_a when calculating these and $P_{ab}^2(\bar{a}) = P_{ab}^1(\bar{a})e_a^2/e_a^1$ since all

these paths go through v_a exactly once and therefore need to be scaled by the new edge weight.

Next we apply Lemma 4.2 again in reverse on the left-hand side which completes the proof:

$$R_b^2 = R_b^1 - \frac{R_a^1 P_{ab}^1(\bar{a})}{1 - P_{bb}^1(\bar{a})} + \frac{R_a^2 P_{ab}^1(\bar{a})\frac{e_a^2}{e_a^1}}{1 - P_{bb}^1(\bar{a})} = R_b^1 + \frac{\left(R_a^2 \frac{e_a^2}{e_a^1} - R_a^1\right) P_{ab}^1(\bar{a})}{1 - P_{bb}^1(\bar{a})} \ . \qquad \square$$

While Theorem 4.1 considers a whole graph, this graph change is identical to the one we would get in an SCC if we added or removed edges from one of the vertices in the component to other (lower level) components. This means we can use Theorem 4.1 for the source component and Lemma 4.1 for the changes in rank this induces in the target and other lower level components.

4.3. **Computations in practice.** To be able to use Theorem 4.1 in practice we would like to first rewrite it in a way that can be computed efficiently. Fortunately it is easy to rewrite it using a power series as seen below.

We start by defining the vector $\vec{Q}_b^a$ which can be seen as the expected number of hits to each vertex in the random walk on the graph where we start in v_a and then after leaving it remove all edges out of v_a (ensuring we do not count any paths going through v_a):

$$\vec{Q}_b^a = \begin{cases} P_{aa}^1, & b = a, \\ \frac{P_{ab}^1(\bar{a})}{1 - P_{bb}^1(\bar{a})}, & b \neq a. \end{cases}$$

To calculate $\vec{Q}_b^a$ we define a new vector $\vec{U}$ as the 1-step probabilities of a random walk starting in v_a before the edge weight change:

$$\vec{U}_b = \begin{cases} e_{ab}, & (a, b) \in E, \\ 0, & (a, b) \notin E. \end{cases}$$

Let B be equal to the scaled adjacency matrix after removing all outgoing edges from v_a; this gives the following power series to calculating $\vec{Q}^a$:

$$\vec{Q}^a = \sum_{k=0}^{\infty} (cB)^k \vec{U}.$$

In vector form this gives

$$R_a^2 = \frac{P_{\to a}^1 \left(1 - P_{aa}^1\right)}{\left(1 - P_{aa}^1\right)\left(1 - P_{aa}^1 \frac{e_a^2}{e_a^1}\right)} = \frac{R_a^1 (1 - Q_a^a)}{1 - Q_a^a \frac{e_a^2}{e_a^1}},$$

$$\vec{R}_{\bar{a}}^2 = \vec{R}_{\bar{a}}^1 + \left(R_a^2 \frac{e_a^2}{e_a^1} - R_a^1\right) \vec{Q}_{\bar{a}}^a.$$

So we see that to find the new rank after the change we need to solve a similar problem as the original problem (calculating it from scratch) but with a slightly different starting vector and matrix to work with. This approach has some potential advantages. First of all we can expect the sum to converge at least as fast as the original problem due to B being identical to A with a couple of removed elements. Another potential advantage is that initially $\vec{U}$ will be sparse, hence making it possible to work with a sparse vector as well as matrix for the first couple of iterations.

5. Bipartite graphs

For some specific types of graph structures it is possible to find explicit expressions for the PageRank of the vertices in the graph. One such graph is the complete bipartite graph, a graph with two sets of vertices with no edges within each set but the maximum amount of edges between the two sets as seen in Figure 5.
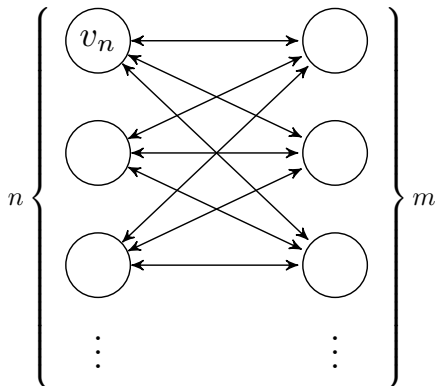

FIGURE 5. Bipartite graph with n vertices on one side and m vertices on the other.

Theorem 5.1. *Consider a complete bipartite graph $G = G_n \cup G_m \cup E(n,m)$, where G_n and G_m are made up of n and m vertices respectively and zero edges and $E(n,m)$ is the set of edges containing all possible edges between pairs of vertices in G_n and G_m. Then given a weight vector $\vec{w}$, PageRank R_{a^*} for a vertex $v_{a^*} \in G_n$ can be written as*

$$
R_{a^*} = \left(w_{a^*} \left(n - (n-1)c^2 \right) + \sum_{v_i \in G_m} w_i c + \sum_{\substack{v_j \in G_n \\ v_j \neq v_{a^*}}} w_j c^2 \right) \frac{1}{n - nc^2} .
$$

To get the rank of a vertex $v_j^* \in G_m$ on the other side, simply swap n and m with each other.

Proof. Using Definition 1.2 and taking the sum of the weights over vertices in G_n and G_m separately, we get

$$
R_{a^*} = \left(w_{a^*} + \sum_{v_i \in G_m} w_i P_{ia^*} + \right) \left(\sum_{k=0}^{\infty} P_{a^* a^*} \right) + \left(\sum_{\substack{v_j \in G_n \\ v_j \neq v_a^*}} w_j P_{ia^*} \right) \left(\sum_{k=0}^{\infty} P_{a^* a^*} \right) .
$$

Using different weights on vertices obviously does not change the probabilities calculated for the uniform weight vector example earlier. By combining previously obtained results

we get

$$R_{a^*} = \left(w_{a^*} + \sum_{v_i \in G_m} \frac{w_i c}{(n - (n-1)c^2)} \right) \left(\frac{n - (n-1)c^2}{n - (n-1)c^2 - c^2} \right)$$

$$+ \left(\sum_{\substack{v_j \in G_n \\ v_j \neq v_{a^*}}} \frac{w_j c^2}{(n - (n-1)c^2)} \right) \left(\frac{n - (n-1)c^2}{n - (n-1)c^2 - c^2} \right)$$

$$= \left(w_{a^*} \left(n - (n-1)c^2 \right) + \sum_{v_i \in G_m} w_i c + \sum_{\substack{v_j \in G_n \\ v_j \neq v_{a^*}}} w_j c^2 \right) \frac{1}{n - nc^2}. \qquad \square$$

We note that Theorem 5.1 can also be used to find the rank of the vertices in a bipartite graph even if it is part of a larger graph with some other incoming edges from vertices outside the bipartite graph. This is possible only if we assume the bipartite graph itself does not have any edges to the outside; in particular, the return probability $\sum_{k=0}^{\infty} (P_{ii})^k$ for vertices v_i in the bipartite graph will remain unchanged even if we add such edges. If we know PageRank $\vec{R}_{out}$ for the vertices outside the bipartite graph we can find new weights $\vec{w}_i$ for vertices in the bipartite graph by calculating

$$w_i^{\text{new}} = w_i^{\text{old}} + \sum_{\substack{j \to i \\ v_j \notin G_P}} \frac{c}{n_j} R_j \ .$$

Here $j \to i$ denotes that there is an edge from vertex e_j to vertex e_i, n_j is the number of edges from e_j and G_p denotes the bipartite graph.

Using Theorem 5.1 we notice a few things. First of all increasing one's own weight (through links from the outside or by itself) will always give a more or less linear increase in rank (more if n is small). Apart from that, we can also see that increasing the weight of said vertex gives a higher increase in rank for those on the other side than those on the same side (assuming $n \approx m$). We can also see that the number of vertices on the other side have no influence on the rank, only their combined weight. On the other hand the number of vertices on the same side is very important for the rank, the fewer vertices on the same side the higher the rank.

Similar results can be found for many other types of graph structures such as complete N-partite graphs [11], and different combinations of a complete graph and a line of vertices [13–16].

6. Conclusions and open problems

In this work we have seen how a partitioning of a graph into components can be used to calculate PageRank in a large network and how such a partitioning can be used to re-calculate PageRank as the network changes. Although we have considered the problem of calculating PageRank, it is worth noting that the same partitioning method could be used when working with Markov chains in general or the solving of linear systems as long as the method used for solving a single component is chosen appropriately.

Below follows some open problems related to this paper.

- Which other types of graph changes can be handled efficiently, in particular changes involving multiple graph changes at the same time (such as the removal or addition of multiple edges)? While we have shown how some changes between components can be handled here, and in [13] it was shown how some limited

forms of edge addition/removal inside a component can be handled, there is still no satisfactory method in the more general case.

- How can the component structure be maintained efficiently over multiple changes? The addition or removal of edges between components does not pose a huge problem since you can then find the new partitioning simply by traversing the graph made up by the graph components in the original graph. A bigger problem is edge removals in which a single component might be split into multiple smaller components. This is especially problematic given the presence of a "giant component", often containing more than half the vertices in the graph.
- For small graphs or graphs with very high symmetry it is possible to find analytical expressions for PageRank similar to what we did in Section 5 for complete bipartite graphs. Intuitively such results should be useful for finding approximations of PageRank in larger more complex graphs or as building blocks when calculating PageRank for larger graphs. An open problem is in deciding which small set of types of subgraphs could be used in such a way as well as how one would go about implementing such a method in practice.
- The computational algorithms' efficiency and approximations of PageRank in large evolving networks and their dependence of damping parameter, when considered in the framework of Markov chains, can be studied using asymptotic expansions and phase space transformations for approximations of stationary distribution vectors and other important functionals of perturbed Markov chains [3, 23, 28–30]. Further detailed development of applications of asymptotic expansions for perturbed Markov chains and semi-Markov processes to computation and approximations of PageRank for large evolving networks is an interesting open problem.

BIBLIOGRAPHY

[1] J. Leskovec, *Google web graph. Google programming contest*, http://snap.stanford.edu/data/web-Google.html.

[2] F. Andersson and S. Silvestrov, *The mathematics of Internet search engines*, Acta Appl. Math. **104** (2008), 211–242. MR2443279

[3] K. E. Avrachenkov, J. A. Filar, and P. G. Howlett, *Analytic Perturbation Theory and Its Applications*, SIAM, Philadelphia, PA, 2013. MR3137641

[4] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin, *Pagerank computation and the structure of the web: Experiments and algorithms*, Proceedings of the Eleventh International Conference on World Wide Web, Alternate Poster Tracks, 2002.

[5] L. Becchetti and C. Castillo, *The distribution of pagerank follows a power-law only for particular values of the damping factor*, Proceedings of the 15th international conference on World Wide Web, ACM Press, 2006.

[6] R. Bellman, *Introduction to Matrix Analysis*, Classics in Applied Mathematics, Society for Industrial and Applied Mathematics, 1970. MR1312360

[7] A. Berman and R. Plemmons, *Nonnegative Matrices in the Mathematical Sciences*, Classics in Applied Mathematics, Society for Industrial and Applied Mathematics, 1994. MR1298430

[8] S. Brin and L. Page, *The anatomy of a large-scale hypertextual web search engine*, Proceedings of the Seventh International Conference on World Wide Web, Amsterdam, The Netherlands, 1998, Elsevier Science Publishers B.V., pp. 107–117.

[9] D. Dhyani, S. S. Bhowmick, and W.-K. Ng, *Deriving and verifying statistical distribution of a hyperlink-based web page quality metric*, Data Knowl. Eng. **46** (2003), no. 3, 291–315.

[10] C. Engström and S. Silvestrov, *A componentwise pagerank algorithm*, Applied Stochastic Models and Data Analysis (ASMDA 2015), The 16th Conference of the ASMDA International Society, 2015, pp. 186–199.

[11] C. Engström and S. Silvestrov, *Non-normalized pagerank and random walks on n-partite graphs*, 3rd Stochastic Modeling Techniques and Data Analysis International Conference (SMTDA2014), 2015, pp. 193–202.

[12] C. Engström and S. Silvestrov, *Using graph partitioning to calculate pagerank in a changing network*, 4th Stochastic Modeling Techniques and Data Analysis International Conference (SMTDA2016), 2016, pp. 155–164.

[13] C. Engström and S. Silvestrov, *Calculating pagerank in a changing network with added or removed edges*, International Conference in Nonlinear Problems in Aviation and Aerospace (ICNPAA), AIP Conference Proceedings, vol. 1798, 2017.

[14] C. Engström and S. Silvestrov, *Generalisation of the damping factor in PageRank for weighted networks*, Modern Problems in Insurance Mathematics (D. Silvestrov and A. Martin-Löf, eds.), EAA Series, Springer, Cham, 2014, pp. 313–333. MR3330692

[15] C. Engström and S. Silvestrov, *PageRank, a look at small changes in a line of nodes and the complete graph*, Engineering Mathematics II. Algebraic, Stochastic and Analysis Structures for Networks, Data Classification and Optimization (S. Silvestrov and M. Rančić, eds.), Springer Proc. Math. Stat., vol. 179, Springer, Heidelberg, 2016, pp. 223–247. MR3630582

[16] C. Engström and S. Silvestrov, *PageRank, connecting a line of nodes with a complete graph*, Engineering Mathematics II. Algebraic, Stochastic and Analysis Structures for Networks, Data Classification and Optimization (S. Silvestrov and M. Rančić, eds.), Springer Proc. Math. Stat., vol. 179, Springer, Heidelberg, 2016, pp. 249–274. MR3630582

[17] F. Gantmacher, *The Theory of Matrices*, Chelsea, 1959. MR0107649

[18] M. Gyllenberg and D. S. Sivestrov, *Quasi-Stationary Phenomena in Nonlinearly Perturbed Stochastic Systems*, De Gruyter Exp. Math., vol. 44, Walter de Gruyter, Berlin, 2008. MR2456816

[19] T. Haveliwala and S. Kamvar, *The second eigenvalue of the google matrix*, Technical Report 2003-20, Stanford InfoLab, 2003.

[20] H. Ishii, R. Tempo, E.-W. Bai, and F. Dabbene, *Distributed randomized pagerank computation based on web aggregation*, Decision and Control, 2009, Held jointly with the 2009 28th Chinese Control Conference, CDC/CCC 2009, Proceedings of the 48th IEEE Conference, 3026–3031.

[21] S. Kamvar and T. Haveliwala, *The condition number of the pagerank problem*, Technical Report 2003-36, Stanford InfoLab, June 2003.

[22] S. Kamvar, T. Haveliwala, and G. Golub, *Adaptive methods for the computation of pagerank*, Linear Algebra Appl. **386** (2004), 51–65. MR2066607

[23] V. S. Koroliuk and N. Limnios, *Stochastic Systems in Merging Phase Space*, World Scientific, Singapore, 2005. MR2205562

[24] P. Lancaster, *Theory of Matrices*, Academic Press, 1969. MR0245579

[25] A. N. Langville and C. D. Meyer, *A reordering for the pagerank problem*, SIAM J. Sci. Comput. **27** (2005), no. 6, 2112–2120. MR2211442

[26] C. P. Lee, G. H. Golub, and S. A. Zenios, *A two-stage algorithm for computing pagerank and multistage generalizations*, Internet Mathematics **4** (2007), 299–327. MR2522947

[27] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, `http://snap.stanford.edu/data`, June 2014.

[28] D. Silvestrov and S. Silvestrov, *Asymptotic expansions for stationary distributions of perturbed semi-Markov processes*, Engineering Mathematics II. Algebraic, Stochastic and Analysis Structures for Networks, Data Classification and Optimization (S. Silvestrov and M. Rančić, eds.), Springer Proc. Math. Stat., vol. 179, Springer, Heidelberg, 2016, pp. 151–222. MR3630580

[29] D. Silvestrov and S. Silvestrov, *Asymptotic expansions for stationary distributions of nonlinearly perturbed semi-Markov processes. I, II* (2016). Part I: arXiv:1603.04734, Part II: arXiv:1603.04743.

[30] D. Silvestrov and S. Silvestrov, *Nonlinearly Perturbed Semi-Markov Processes*, SpringerBriefs Probab. Math. Stat., 2017. MR3700359

[31] R. Tarjan, *Depth first search and linear graph algorithms*, SIAM J. Comput. **1** (1972), no. 2, 146–160. MR0304178

[32] Q. Yu, Z. Miao, G. Wu, and Y. Wei, *Lumping algorithms for computing google's pagerank and its derivative, with attention to unreferenced nodes*, Information Retrieval **15** (2012), no. 6, 503–526.

Division of Applied Mathematics, The School of Education, Culture and Communication, Mälardalen University, Box 883, 721 23 Västerås, Sweden
  Email address: `christopher.engstrom@mdh.se`

Division of Applied Mathematics, The School of Education, Culture and Communication, Mälardalen University, Box 883, 721 23 Västerås, Sweden
  Email address: `sergei.silvestrov@mdh.se`