Internet Mathematics Vol. 6, No. 3: 373–398

Speeding Up Algorithms on Compressed Web Graphs

Chinmay Karande, Kumar Chellapilla, and Reid Andersen

Abstract. A variety of lossless compression schemes has been proposed to reduce the storage requirements of web graphs. One successful approach is virtual-node compression [Buehrer and Chellapilla 08], in which often-used patterns of links are replaced by links to virtual nodes, creating a compressed graph that succinctly represents the original. In this paper, we show that several important classes of web graph algorithms can be extended to run directly on virtual-node-compressed graphs, such that their running times depend on the size of the compressed graph rather than on that of the original. These include algorithms for link analysis, estimating the size of vertex neighborhoods, and a variety of algorithms based on matrix–vector products and random walks. Similar speedups have been obtained previously for classical graph algorithms such as shortest paths and maximum bipartite matching. We measure the performance of our modified algorithms on several publicly available web graph data sets, and demonstrate significant empirical speedups that nearly match the compression ratios.

1. Introduction

Compression schemes can significantly reduce the number of bits per edge required to losslessly represent web graphs [Boldi and Vigna 04b, Boldi and Vigna 04a]. One approach to implementing algorithms on compressed graphs is to decompress the graph on the fly, so that a client algorithm does not need to know how the underlying graph is compressed. Another approach is to design specialized algorithms that can directly use the compressed representation. It can be

1542-7951/09 $0.50 per page

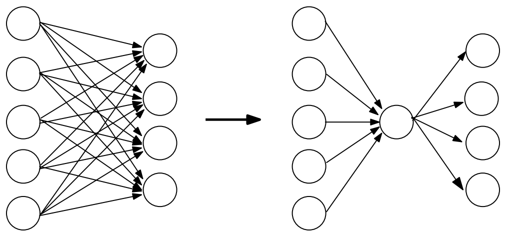

Figure 1. The bipartite clique–star transformation.

shown that for certain compression schemes, such algorithms can be made to run faster on the compressed graph than on the original [Feder and Motwani 95].

In virtual-node compression, a succinct representation of the graph is constructed by replacing dense subgraphs by sparse ones [Buehrer and Chellapilla 08]. In particular, a directed bipartite clique on the vertex set K is replaced by a star centered at a new "virtual" node, with nodes in K being the leaves (see Figure 1). Applying this transformation repeatedly leads to a compressed graph with significantly fewer edges and a relatively small number of additional nodes.

It is shown in [Feder and Motwani 95] that several classical graph algorithms can be sped up using a similar type of virtual-node compression, in which an undirected clique is transformed into a star. The authors showed that algorithms for all-pairs shortest paths, bipartite matching, and edge and vertex connectivity can be modified so their running times depend on the size of the compressed graph rather than on that of the original. They also showed that dense graphs can be significantly compressed by virtual-node compression; they gave an algorithm that finds, for any graph with $\Omega(n^2)$ edges, a compressed graph with $O(n^2/\log n)$ edges. This result, combined with their sped-up algorithms, improved the worst-case running time bounds for all-pairs shortest paths and bipartite matching.

Recently, it was demonstrated in [Buehrer and Chellapilla 08] that virtual-node compression can achieve high compression ratios for web graphs. The authors introduced a frequent-pattern mining algorithm for finding directed bipartite cliques, and showed that their algorithm achieves compression ratios of four to eight on a variety of page-level web graphs, which is comparable to state-of-the-art compression methods based on gap coding [Boldi and Vigna 04b, Boldi and Vigna 04a]. This high compression ratio reflects the frequent occurrence of bipartite cliques in web graphs, which was observed earlier in the context of community-finding [Kumar et al. 99].

In this paper, we show that a large class of web graph algorithms can be extended to run on virtual-node-compressed graphs, with running time speedups proportional to the compression ratio. As a fundamental tool, we first show that multiplication by the adjacency matrix of the graph can be performed in time proportional to the size of the compressed graph. Using this matrix multiplication routine as a black box, we obtain significant speedups for numerous popular web graph algorithms, including PageRank, HITS, and SALSA, and various algorithms based on random walks. This multiplication routine can be implemented in the sequential file access model, and can be implemented on a distributed graph using a small number of global synchronizations.

We then consider a second approach to speeding up PageRank and SALSA, this time using the computation of stationary vectors as a black box. We show that by computing an appropriately modified PageRank directly on the compressed graph, we can perform a simple transformation of the result to obtain the PageRank of the original graph. With this approach, one can achieve a speedup on the compressed graph using an existing PageRank implementation. We discuss several tradeoffs between these two approaches, including the number of iterations required for convergence and the number of synchronizations required in a distributed implementation.

We tested the performance of both of these approaches for PageRank and SALSA on large publicly available web graphs, which we compressed using techniques described in [Buehrer and Chellapilla 08]. For these graphs the compression ratios are roughly 4 to 6, and the speedup achieved by our algorithms is roughly 2.5 to 4.5 times over the uncompressed versions. It is expected that the speedup is close to the compression ratio but does not exactly match it, since various operations that require $O(|V|)$ time are not sped up.

The paper is organized as follows. In Section 2 we define virtual-node compression schemes, describe algorithms for computing compressed representations of graphs, and introduce notation. In Section 3 we present an algorithm for performing fast matrix–vector multiplication on compressed graphs. In Section 4 we describe an alternative approach to speeding up the computation of PageRank and SALSA. In Section 5 we discuss additional applications, and in Section 6 we present experimental results. A conference version of this paper [Karande et al. 09] appeared earlier. We have since expanded the discussion of the virtual-node compression scheme in Section 2.2 as well as the experimental results in Section 6.

2. Background

2.1. Graph Compression Using Virtual Nodes

We now describe how virtual-node compression is applied to a directed graph $G(V, E)$. This scheme is based on a graph transformation that replaces a directed bipartite clique by a directed star. A directed bipartite clique (or biclique) $\langle S, T\rangle$ is a pair of disjoint vertex sets S and T such that for each $u \in S$ and $v \in T$, there is a directed link from u to v in G. Given a biclique $\langle S, T\rangle$, we form a new compressed graph $G'(V', E')$ by adding a new vertex w to the graph, removing all the edges in $\langle S, T\rangle$, and adding a new edge $uw \in E'$ for each $u \in S$ and a new edge $wv \in E'$ for each $v \in T$. This transformation is depicted in Figure 1. Note that the number of vertices increases by 1, while the number of edges decreases, since $|S| \times |T|$ edges in E are replaced by $|S| + |T|$ edges in E'. We remark that this is a directed bipartite version of the clique–star transformation from [Feder and Motwani 95].

We call the node w a *virtual* node as opposed to the *real* nodes already present in G. Note that the biclique–star transformation essentially replaces an edge uv in G with a unique path $u \to w \to v$ in G' that acts as a placeholder for the original edge. We will call such a path a *virtual* edge.

The biclique–star transformation may be performed again on G'. Virtual nodes are allowed to be reused, so the bipartite clique $\langle S', T'\rangle$ found in G' may contain the virtual node w. In this case, the virtual edge path between u and v in the resulting graph G'' may be extended to $u \to w \to w' \to v$ or $u \to w' \to w \to v$. We can now define a virtual edge more formally: a path $u \to w_1 \to \cdots \to w_l \to v$ in the compressed graph is a virtual edge if and only if $w_1, \ldots, w_l$ are virtual nodes and u, v are real nodes with an edge from u to v in the original graph G. Note that the definition of a virtual node encompasses those edges $uv \in E$ that remain in G'.

To obtain significant compression, the biclique–star transformation can then be repeated many times. A graph obtained by this process is called a *compression* of G.

More generally, given two digraphs $G(V, E)$ and $G'(V', E')$, we say that G' is a compression of G if it can be obtained by applying a series of bipartite clique–star transformations to G. We will denote this relation by $G' \prec G$. Any compression $G' \prec G$ satisfies the following properties, which are straightforward to verify and were proved in [Buehrer and Chellapilla 08]. We provide sketches of those proofs here.

Lemma 2.1. *There is a one-to-one correspondence between the set of edges in G and the set of virtual edges in G'.*

Proof. If an edge $uv \in E$ exists in G', then the one-to-one correspondence is clear, since uv is part of the set of virtual edges in G'. Otherwise, we look at the biclique–star transformation that splits $u \to v$ into $u \to w \to v$. If both the edges uw and wv exist in G', then uv corresponds to the virtual edge $u \to w \to v$. If not, we can continue the splitting of edges recursively to find a unique virtual edge corresponding to uv. □

Lemma 2.2. *There exists no cycle in G' consisting entirely of virtual nodes.*

Proof. We will prove the lemma by recursion. Graph G did not contain such a cycle to start with. Now consider the last biclique–star transformation that changed the graph from G'' to G'. If G' contains a cycle on virtual nodes, then either the same cycle existed in G'' or the transformation produced two adjacent edges $u \to w \to v$ of that cycle. Since both u, v are virtual nodes, a smaller cycle on virtual nodes must exist in G''. □

We use the following notation for a compressed graph G'. The set of real nodes in G' is denoted by ${}_rV'$, and the set of virtual nodes is ${}_vV'$.

2.2. Finding Virtual Nodes Using Frequent Itemset Mining

The algorithms we describe in this paper can be applied to any compression of G, but their performance depends on the properties of the compression. The most important property is the number of edges and nodes in the compressed graph. We will refer to the quantity $|E|/|E'|$ as the compression ratio. In addition, we want to bound the maximum length of any virtual edge. Clearly, longer virtual edges are undesirable, since one can access the original edge only after discovering the entire virtual edge.

An algorithm was introduced in [Buehrer and Chellapilla 08] that produces compressions of web graphs with high compression ratio and small depth. This algorithm finds collections of bicliques using techniques from frequent itemset mining, and runs in time $O(|E|\log(|V|))$. We will provide an overview of the algorithm and highlight the properties that are important from our point of view. Nodes in the graph are viewed as items, and the set of out-neighbors of a node as an itemset. Therefore, a large frequent itemset corresponds to a large biclique in the graph. First, the nodes are sorted by in-degree so that the most frequent singleton item appears first. The nodes are then added to a prefix tree such that every path from root to leaf in this tree corresponds to the set of out-neighbors of a node in the graph. In fact, every subset of vertices on such a path in the tree corresponds to a biclique in the graph. In each phase, the algorithm constructs this prefix tree, heuristically identifies many disjoint bicliques as paths in the tree, and performs the biclique–star transformation on all of them. The process is then repeated. Virtual nodes added to the graph in one phase can be part of the bicliques found in the next phase.

Note that this compression scheme contains several heuristics that work well in practice, but does not provide any guarantees with regard to the quality of the compression. Nonetheless, it has a few desirable properties worth noting for our purposes. It is reported in [Buehrer and Chellapilla 08] that the resulting compressed graphs contain five to ten times fewer edges than the original, for a variety of page-level web graphs. To obtain this compression, one typically requires four to five phases of the algorithm. We will call the number of phases used to compress G into G' the *depth* of compression. Our bounds on the precision of our algorithm depend on the depth of compression. In our experiments, our algorithm operates at a much higher precision than that guaranteed by our theoretical bounds, and hence does not seem to be dependent on this property.

Henceforth, we will assume the use of this compression scheme when referring to *compressed graphs*, and we assume that the depth of the compression is bounded by a small constant. Note that if the depth of compression is k, then the worst case is that a virtual edge in the compressed graph may have length at most 2^k.

We remark that approximation algorithms for finding the best virtual-node compressions were considered in [Feder et al. 03]. There, it is shown that finding the optimal compression is NP-hard, but a good approximation algorithm exists for the restricted problem of finding the best compression obtained from a collection of vertex-disjoint cliques.

2.3. Combination of Structural and Representational Compression

Some existing graph compression techniques [Boldi and Vigna 04b, Boldi and Vigna 04a] are representational. They choose an optimal encoding that requires only a small number of bits to represent the graph. However, this provides only storage reduction. Since the graph represented by this encoding is still the original graph, any algorithm will require the same amount of computation. On the other hand, the virtual-node compression scheme yields a smaller graph, and we design algorithms that use the smaller graph to provide a speedup, apart from the storage reduction. In fact, the representational compression schemes of [Boldi and Vigna 04b, Boldi and Vigna 04a] can be used on top of virtual-node compression to boost the storage reduction effect.

2.4. Notation

We consider directed graphs $G(V, E)$ with no loops or parallel edges. We denote the set of in-neighbors and out-neighbors of node v by $\delta^G_{\text{in}}(v)$ and $\delta^G_{\text{out}}(v)$ respectively.

We overload the symbol E to denote the adjacency matrix of the graph, where

$$E[u,v] = \begin{cases} 1 & \text{If edge } uv \in E, \\ 0 & \text{otherwise.} \end{cases}$$

When talking about probability distributions on the vertices of G, we will use boldface letters such as $\mathbf{p}$ to denote a column vector of dimension $|V|$, unless mentioned otherwise. When M is a matrix, we will use $M[u]$ to represent the row corresponding to vertex u and $M[u,v]$ to represent the entry in row u and column v.

Since we will be concerned with random walks on the Markov chain on the underlying graph G, we will denote the probability of transition from u to v by $\Pr(u,v)$. By W we will denote the random-walk matrix obtained by normalizing each row of E to sum to 1. It is then clear that if $\mathbf{p_0}$ is the starting probability distribution, then $\mathbf{p_1} = W^T\mathbf{p_0}$ is the distribution resulting from a single step of the uniform random walk on the graph.

3. Speeding Up Matrix–Vector Multiplication

A large class of graph algorithms can be expressed succinctly and efficiently in terms of multiplication by the adjacency matrix. Here we show that the multiplication of a vector by the adjacency matrix of a graph can be carried out in time proportional to the size of the graph's compressed representation. This matrix multiplication routine can be used as a black box to obtain efficient compressed implementations.

3.1. Adjacency Matrix Multiplication

Proposition 3.1. *Let G be a graph with adjacency matrix E, and let $G' \prec G$ be a compression of G. Then for any vector $\mathbf{x} \in \mathbb{R}^{|V|}$, the matrix–vector product $E^T\mathbf{x}$ can be computed in time $O(|E'| + |V'|)$. This computation needs only sequential access to the adjacency list of G' and does not require the original graph G.*

Proof. First let us explore what the computation $\mathbf{y} = E^T\mathbf{x}$ looks like when the uncompressed graph G is accessible.

Algorithm 1 performs a series of what are popularly called "push" operations: The value stored at node u in $\mathbf{x}$ is "pushed" along the edge uv. This algorithm simply encodes the following definition of $\mathbf{y}$:

$$\mathbf{y}[v] = \sum_{uv \in E} \mathbf{x}[u]. \tag{3.1}$$

Algorithm 1. (Multiply($\boldsymbol{E}$, $\mathbf{x}$))

```
forall $v \in V$ do
    $\mathbf{y}[v] = 0$;
end
forall Nodes $u \in V$ do
    forall Edges $uv \in E$ do
        $\mathbf{y}[v] = \mathbf{y}[v] + \mathbf{x}[u]$;
    end
end
```

We extend this definition to compressed graphs, by extending the vector $\mathbf{x}$ onto virtual nodes in the following fashion: For a virtual node v, we expand $\mathbf{x}[v]$ as

$$\mathbf{x}[v] = \sum_{uv \in E'} \mathbf{x}[u]. \tag{3.2}$$

Armed with the above definition, we now provide the equation that computes $\mathbf{y}$ using the compressed graph G':

$$\mathbf{y}[v] = \sum_{uv \in E'} \mathbf{x}[u]. \tag{3.3}$$

We claim that definitions (3.1) and (3.3) of $\mathbf{y}$ are equivalent. This follows easily from Lemma 2.2. Hence, using the recursive definition (3.2), we can expand the terms corresponding to virtual nodes on the right side of equation (3.3) to obtain exactly equation (3.1).

Although definitions (3.1) and (3.3) are equivalent, their implementations are not. Note that the input vector $\mathbf{x}$ is not defined on virtual nodes. Moreover, due to the recursive definition (3.2), these values have dependencies. For illustration, consider the example in Figure 2, where w is a virtual node:

$$\mathbf{y}[v] = \mathbf{x}[u_1] + \mathbf{x}[u_2] + \underline{\mathbf{x}[u_3] + \mathbf{x}[u_4] + \mathbf{x}[u_5]} = \mathbf{x}[u_1] + \mathbf{x}[u_2] + \mathbf{x}[w].$$

Hence, although the value of $\mathbf{y}[u]$ is encoded correctly by definitions (3.2) and (3.3), it depends on $\mathbf{x}[w]$, which itself needs to be computed before the "push" operation on edge wv is performed. The problem then simply becomes that of arranging the "push" operations on edges incident upon virtual nodes.

Consider a virtual node v all of whose in-links originate from real nodes. Lemma 2.2 guarantees the existence of such nodes. Clearly, when the push operations on all the out-links of all the real nodes are finished, $\mathbf{x}[v]$ has been computed. Now we can go ahead and "push" scores along all the out-links of v,

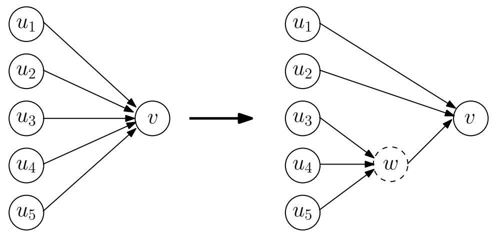

Figure 2. Push operations on a compressed graph.

which may in turn help complete the computation of $\mathbf{x}[w]$ for some other virtual node w.

We now formalize by assigning a rank $R(v)$ to each virtual node v using the following recursive definition:

- If u is real for all $uv \in E'$, then $R(v) = 0$.
- Otherwise, $R(v) = 1 + \max_{u \in \delta_{\text{in}}(v) \cap \ _vV} R(u)$.

We now reorder the rows of the adjacency list representation of G' in the following manner:

1. Adjacency lists of real nodes appear before those of virtual nodes.
2. For two virtual nodes u and v, if $R(u) < R(v)$ then the adjacency list of u appears before that of v.

This reordering now imparts the following property to the adjacency list of G': For every virtual node v and any u such that $uv \in E'$, the adjacency list of u appears before that of v. Therefore $\mathbf{x}[v]$ can be computed *before* we begin to push scores along out-links of v. Algorithm 2 computes $\mathbf{y} = E^t\mathbf{x}$ using the adjacency matrix E' of G' reordered as above.

Finally, note that the reordering can be performed during preprocessing by computing the ranking function R using a simple algorithm that requires $O(|E'|+|V'|)$ time. $\square$

Note that we can also speed up the computation of $z = E\mathbf{x}$ in a similar manner, by compressing the in-link graph rather than the out-link graph. The same collection of virtual nodes can be used for both the in-link graph and the out-link graph, leading to compressed in-link and out-link graphs with the same values of $|V'|$ and $|E'|$. However, the in-links of virtual nodes in the compressed graph must be stored separately, and they require a different ordering of virtual nodes.

Algorithm 2. (Compressed-multiply($\boldsymbol{E}$, $\mathbf{x}$))

```
forall Real nodes $v$ do
    $\mathbf{y}[v] = 0$;
end
forall Virtual nodes $v$ do
    $\mathbf{x}[v] = 0$;
end
forall Nodes $u \in V'$ do
    forall Edges $uv \in E'$ do
        if $v$ is real then
            $\mathbf{y}[v] = \mathbf{y}[v] + \mathbf{x}[u]$;
        else
            $\mathbf{x}[v] = \mathbf{x}[v] + \mathbf{x}[u]$;
        end
    end
end
```

3.2. Applications to Computation of Importance Metrics

The approaches described above can be used to speed up the canonical link-analysis algorithms PageRank [Brin and Page 98, Page et al. 98], HITS [Kleinberg 99], and SALSA [Lempel and Moran 00]. Here we briefly describe implementations of these algorithms using black-box compressed multiplication to provide context for the following section. We defer the discussion of other applications of adjacency matrix multiplication to Section 5.

These algorithms essentially perform several iterations of the power method, for different graph-related matrices. Each iteration requires $\Theta(|E| + |V|)$ operations on an uncompressed graph G. Given a compressed graph G', each iteration can be sped up to $\Theta(|E'|+|V'|)$ operations using compressed-multiply. Typically $|V'|$ is 20 to 40% larger than $|V|$, so the performance boost observed is determined mainly by the ratio $|E'|/|E|$. Alternative compressed implementations of these algorithms will be described in more detail in the following section.

- *PageRank:* Given a graph G with adjacency matrix E, PageRank can be computed by the following power method step:

$$\mathbf{x}_{i+1} = (1-\alpha)E^T(D^{-1}\mathbf{x}_i) + \alpha\mathbf{j}\,,$$

  where α is the jump probability and $\mathbf{j}$ is the jump vector. Refer to Section 4.1 for a detailed definition of PageRank.

- *HITS and SALSA:* The HITS algorithm [Kleinberg 99] assigns a separate hub score and authority score to each web page in a query-dependent graph, equal to the top eigenvector of EE^T and E^TE. SALSA can be viewed as a normalized version of HITS, where the authority vector $\mathbf{a}$ and hub vector $\mathbf{h}$ are the top eigenvectors of $W_{\mathrm{r}}^T W_{\mathrm{c}}$ and $W_{\mathrm{c}} W_{\mathrm{r}}^T$, where W_{r} and W_{c} are the row and column normalized versions of E.

4. Stochastic Algorithms on Compressed Graphs

In this section, we consider an alternative method for computing the stationary vectors for PageRank and SALSA using compressed graphs. We show that the stationary vector in the original graph can be computed by computing the stationary vector of a Markov chain running on the compressed graph, then projecting and rescaling. This allows us to compute PageRank or SALSA on the original graph by running an existing implementation of the algorithm directly on the compressed graph. The methods discussed above are specific to these algorithms—we cannot apply them to compute the steady state of a general stochastic metric. Refer to Section 5 for more detailed discussion of this point.

4.1. PageRank on Compressed Graphs

PageRank (introduced in [Brin and Page 98, Page et al. 98]) models a uniform random walk on the Web graph performed by a so-called *random surfer*. The matrix W as defined in Section 2.4 represents the underlying Markov chain. To ensure ergodicity, we assume that the surfer clicks on a random link on a page with probability $1-\alpha$, $0 < \alpha < 1$. With probability α, she jumps to any page in the graph, which she then chooses from the probability distribution $\mathbf{j}$. Here $\mathbf{j}$ is a vector of positive entries called the *jump vector*. This modification makes the Markov chain ergodic, and hence, the equation governing the steady state becomes

$$\mathbf{p} = \left((1-\alpha)W^T + \alpha J\right)\mathbf{p} = L^T\mathbf{p},$$

where J is simply the square matrix containing a copy of $\mathbf{j}$ in each column.

The power method can be efficiently applied to approximate the steady state of this Markov chain in $\Theta(r(|E|+|V|))$ operations given an adjacency-list representation of E, by multiplying the current distribution vector $\mathbf{p_i}$ by $(1-\alpha)W^T$ and then adding the vector $\alpha\mathbf{j}$ to it. Here, r is the number of power iterations performed.

Our goal then is to run an algorithm similar to the above on a compression $G' \prec G$ such that just restricted to nodes in V, it *models* the jump-adjusted

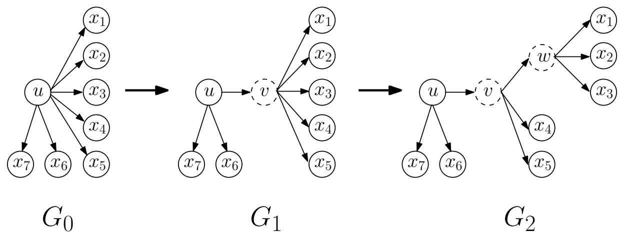

Figure 3. Illustration of the Δ function.

uniform random walk on G. From Lemma 2.1, if $u \in G$, then starting from u, the walk can reach exactly the set $\delta^G_{\text{out}}(u)$ using virtual edges. Let p_{uv} be the probability that v is the first real node visited by the random walk on G' when starting from u. If we tweak the transition probabilities on G' so as to have p_{uv} equal to the probability of the transition $u \to v$ in G, then we have a good model of the original uniform random walk on G.

With this in mind, we now define some required notation. For a graph G (compressed or otherwise), we define $\Delta_G(u)$ as follows:

$$\Delta_G(u) = \begin{cases} 1 & \text{if } u \text{ is real,} \\ \sum_{w \in \delta^G_{\text{out}}(u)} \Delta_G(w) & \text{if } u \text{ is virtual.} \end{cases}$$

For example, in graph G_2 in Figure 3, we have $\Delta_{G_2}(v) = 5$ even though $|\delta^{G_2}_{\text{out}}(v)| = 3$, because the following five virtual edges leading to real nodes $x_1, \ldots, x_5$ pass through v: (1) $(u \to v \to w \to x_1)$, (2) $(u \to v \to w \to x_2)$, (3) $(u \to v \to w \to x_3)$, (4) $(u \to v \to x_4)$, and (5) $(u \to v \to x_5)$.

This configuration can be formed from the original graph G_0 when a bipartite clique involving u and $x_1, \ldots, x_3$ is replaced by virtual node w and subsequently, a bipartite clique involving u and w, x_4, x_5 was replaced by virtual node v. Refer to Figure 3 for illustration.

We will assume that the values of this function are supplied to us along with the compressed graph. Indeed, the value $\Delta_G(v)$ is readily available to the compressor algorithm (see [Buehrer and Chellapilla 08]) when it introduces the virtual node v, and hence it needs to record only this entry associated with node v. This increases the storage requirement for the compressed graph, but not by more than a factor of 2, which itself is a generous, worst-case estimate since the proportion of virtual nodes is very small. In practice, the extra storage required is close to 3 to 5% [Buehrer and Chellapilla 08]. Given the function $\Delta_G(u)$, we define the real out-degree of u in G:

$$\Gamma_G(u) = \sum_{w \in \delta^G_{\text{out}}(u)} \Delta_G(w).$$

For a real node u, $\Gamma_G(u)$ is nothing but the number of real nodes in G reachable from u using a virtual edge. For a virtual node v, $\Gamma_G(v) = \Delta_G(v)$.

It is easy to verify that if $G' \prec G$, then for a node $u \in G$, $\Gamma_{G'}(u) = \Gamma_G(u)$. Moreover, if G is an uncompressed graph, then $\Gamma_{G'}(u) = |\delta^G_{\text{out}}(u)|$.

How are the functions Δ_G and Γ_G relevant? Consider the edge uv in graph G_2 in Figure 3: $\Delta_{G_2}(v) = 5$ and $\Gamma_{G_2}(u) = 7$. Hence, virtual edges passing through the virtual node v *capture* or *encode* five real out-neighbors of u in the original graph G. Common sense tells us that to accurately model the uniform random walk on G, the probability of the transition $u \to v$ must be $5/7$.

With this background, we can now define a random walk on a graph G' compressed from G that exhibits the desired modeling behavior:

1. The random walk on G' is not uniform (unlike that on G). For example, for the compressed graph G_2 in Figure 3, we ensure that $\Pr(u, v) = 5 \cdot \Pr(u, x_6)$, since v captures the virtual edges to five real neighbors of u. Similarly, we keep $\Pr(v, w) = 3 \cdot \Pr(v, x_4)$.

2. We ensure that the jump vector has zeros in entries corresponding to virtual nodes. Similarly, transitions made from virtual nodes have zero jump probability. This ensures that the Markov chain models exactly the uniform random walk on G.

Given graphs $G' \prec G$, jump probability α, and the jump vector $\mathbf{j}$, we define the random walk on G' as follows:

1. Let X be the matrix of dimension $|V'| \times |V'|$ such that
$$X[u, v] = \frac{\Delta_{G'}(v)}{\Gamma_{G'}(u)}.$$

2. We obtain Y from X by making adjustments for the jump probability:
$$Y[u, v] = \begin{cases} (1-\alpha)X[u, v] & \text{if } u \text{ is real,} \\ X[u, v] & \text{if } u \text{ is virtual.} \end{cases}$$

3. Pad the jump vector $\mathbf{j}$ with zeros to obtain a jump vector $\mathbf{j'}$ for G'. This assigns the probability of a jump transition into a virtual node to be zero. Let J' be the jump matrix containing copies of $\mathbf{j'}$ in each column.

4. The desired Markov chain is given by the transition matrix $\text{MC}(G') = Z = (Y + \alpha J'^T)$.

```
Algorithm 3. (ComputePageRank($G$, $G'$, $\alpha$, $\mathbf{j}$))
1  Compute $Z = \mathrm{MC}(G')$
2  Compute the steady state $\mathbf{p}'$ of the Markov chain represented by $Z$.
3  Project $\mathbf{p}'$ onto the set of real nodes to form $\mathbf{p}''$. Discard the values for
   virtual nodes.
4  Scale $\mathbf{p}''$ up to unit $L_1$ norm to obtain $\mathbf{p}$, which is the desired vector of
   PageRank values on $G$.
```

Just like $\mathrm{MC}(G)$, the irreducibility and aperiodicity of $\mathrm{MC}(G')$ is ensured by the jump vector $\mathbf{j}'$. It makes the set of real nodes strongly connected, and since every virtual node has a path to and from a real node, the resulting Markov chain is ergodic. Hence it makes sense to talk about the steady state of $\mathrm{MC}(G')$.

Algorithm 3 takes as input a graph G, its compressed representation G', jump probability α, and the jump vector $\mathbf{j}$ to compute PageRank on vertices of G strictly using the graph G'.

From the schematic, it is clear that Algorithm 3 can be implemented to run in time $\Theta(r(|E'| + |V'|))$, where r is the desired number of power iterations. We prove correctness of the algorithm in Theorem 4.1.

Theorem 4.1. *The vector* $\mathbf{p}$ *computed by Algorithm 3 satisfies*

$$\mathbf{p} = \left((1-\alpha)W^T + \alpha J\right)\mathbf{p}.$$

That is, $\mathbf{p}$ *is the steady state of the jump-adjusted uniform random walk* $\mathrm{MC}(G)$*.*

Proof. Although Algorithm 3 is not recursive, our proof will be. The recursion will be based on the phases of compression mentioned in Section 2.1.

Let $G_i(V_i, E_i)$ for $0 \le i \le k$ be a series of graphs

$$G' = G_k \prec G_{k-1} \prec \cdots \prec G_1 \prec G_0 = G$$

such that G_{i+1} is obtained from G_i by one phase of the clique–star transformations, i.e., by replacing many edge-disjoint bipartite cliques with virtual nodes. The following property follows from this procedure: For $uv \in E_i$, if $uv \notin E_{i+1}$, then there exists a unique virtual edge $u \to w \to v$ in G_{i+1}. This bound helps us expand the equations governing the steady state of $\mathrm{MC}(G_{i+1})$.

Let $\mathbf{j}_i$ be the padded jump vector associated with $\mathrm{MC}(G_i)$. Let $\mathbf{p}_i$ be the steady state of $\mathrm{MC}(G_i)$. The following claim is crucial to the proof.

Claim 4.2. *For all* $0 \le i < k$ *and* $u \in V_i$*,* $\mathbf{p}_{i+1}[u] = \beta_i \mathbf{p}_i[u]$*, where* β_i *is a constant depending only on* i*.*

Proof of Claim 4.2. For better readability, let

$$a_i(v) = \begin{cases} (1-\alpha) & \text{if } v \in_r V_i, \\ 1 & \text{if } v \in_v V_i, \end{cases}$$

be the jump multiplier. Then for any $u \in V_i$, the steady-state equation governing $\mathbf{p}_{i+1}[u]$ can be written as

$$\mathbf{p}_{i+1}[u] = \alpha \mathbf{j}_{i+1}[u] + \sum_{v \in \delta_{\text{in}}^{G_{i+1}}(u)} a_{i+1}(v) \frac{\Delta_{G_{i+1}}(u)}{\Gamma_{G_{i+1}}(v)} \mathbf{p}_{i+1}[v]. \tag{4.1}$$

The first term in equation (4.1) is the contribution made by the jump vector to $\mathbf{p}_{i+1}[u]$. But since $\mathbf{j}_{i+1}$ is obtained from $\mathbf{j}_i$ by simple padding, $\mathbf{j}_{i+1}[u] = \mathbf{j}_i[u]$.

To analyze the summation in equation (4.1), we split $\delta_{\text{in}}^{G_{i+1}}(u)$ into two parts: Let $Q = V_i \cap \delta_{\text{in}}^{G_{i+1}}$ be the set of in-neighbors of u already present in G_i. Let $Q' = \delta_{\text{in}}^{G_{i+1}} - Q$ be the set of virtual in-neighbors of u that were added during the transformation from G_i to G_{i+1}. For $v \in Q$, the edge vu already existed in G_i; hence we have

$$\sum_{v \in Q} a_{i+1}(v) \frac{\Delta_{G_{i+1}}(u)}{\Gamma_{G_{i+1}}(v)} \mathbf{p}_{i+1}[v] = \sum_{v \in Q} a_i(v) \frac{\Delta_{G_i}(u)}{\Gamma_{G_i}(v)} \mathbf{p}_{i+1}[v]. \tag{4.2}$$

Nodes in Q' are "fresh" virtual nodes. Therefore, for $v \in Q'$, we have $a_{i+1}(v) = 1$. We can expand the term $\mathbf{p}_{i+1}[v]$ as follows:

$$\begin{aligned} &\sum_{v \in Q'} a_{i+1}(v) \frac{\Delta_{G_{i+1}}(u)}{\Gamma_{G_{i+1}}(v)} \mathbf{p}_{i+1}[v] \qquad (4.3) \\ &\qquad = \sum_{v \in Q'} \frac{\Delta_{G_{i+1}}(u)}{\Gamma_{G_{i+1}}(v)} \left(\sum_{w \in \delta_{\text{in}}^{G_{i+1}}(v)} a_{i+1}(w) \frac{\Delta_{G_{i+1}}(v)}{\Gamma_{G_{i+1}}(w)} \mathbf{p}_{i+1}[w] \right). \end{aligned}$$

Recall the following properties for $v \in Q'$:

1. From the definition of Q', the in-neighbors of v in G_{i+1} are in fact in-neighbors of u in G_i.

2. From the fact that edge-disjoint cliques are chosen for transformation from G_i to G_{i+1} (see Section 2.2), the sets $\delta_{\text{in}}^{G_{i+1}}(v)$ are disjoint over $v \in Q'$.

3. From Lemma 2.1, in G_{i+1} we have

$$\bigcup_{v \in Q'} \delta_{\text{in}}^{G_{i+1}}(v) = \delta_{\text{in}}^{G_i}(u) - Q.$$

4. For v a virtual node, $\Delta_{G_{i+1}}(v) = \Gamma_{G_{i+1}}(v)$.

Using these properties, we can now write equation (4.3) as

$$\sum_{v \in Q'} a_{i+1}(v) \frac{\Delta_{G_{i+1}}(u)}{\Gamma_{G_{i+1}}(v)} \mathbf{p}_{i+1}[v] = \sum_{w \in \delta_{\text{in}}^{G_i}(u) - Q} a_i(w) \frac{\Delta_{G_i}(u)}{\Gamma_{G_i}(w)} \mathbf{p}_{i+1}[w]. \tag{4.4}$$

Substituting (4.2) and (4.4) in equation (4.1), we get

$$\mathbf{p}_{i+1}[u] = \alpha \mathbf{j}_i[u] + \sum_{v \in \delta_{\text{in}}^{G_i}(u)} a_i(v) \frac{\Delta_{G_i}(u)}{\Gamma_{G_i}(v)} \mathbf{p}_{i+1}[v].$$

Compare this with the steady-state equation governing $\mathbf{p}_i[u]$:

$$\mathbf{p}_i[u] = \alpha \mathbf{j}_i[u] + \sum_{v \in \delta_{\text{in}}^{G_i}(u)} a_i(v) \frac{\Delta_{G_i}(u)}{\Gamma_{G_i}(v)} \mathbf{p}_i[v].$$

We conclude that the vector $\mathbf{p}_{i+1}$ when restricted to nodes in V_i satisfies the same steady-state equations satisfied by $\mathbf{p}_i$. Since these equations uniquely determine $\mathbf{p}_i$ up to scaling, we arrive at the statement of Claim 4.2. $\square$

Successively using the statement of Claim 4.2, we see that there is a constant β such that for all $u \in V_0$, we have $\mathbf{p}'[u] = \mathbf{p}_k[u] = \beta \mathbf{p_0}[u]$. Hence $\mathbf{p}$ as computed by Algorithm 3 satisfies

$$\mathbf{p} = \left((1-\alpha)W^T + \alpha J\right) \mathbf{p}.$$

The scaling ensures that $\mathbf{p}$ has unit L_1 norm, and hence is the desired PageRank vector. $\square$

Although Theorem 4.1 completes the theoretical analysis of our method, one can begin to see a possible practical difficulty in the implementation of Algorithm 3. If the value of the constant β is very small, the computed values of $\mathbf{p}'$ will contain very few bits of accuracy, and the subsequent scaling up will only maintain this precision. In what follows, we prove a lower bound on β.

Theorem 4.3. *Let*

$$G' = G_k \prec G_{k-1} \prec \cdots \prec G_1 \prec G_0 = G$$

be any sequence of graphs as in the proof of Theorem 4.1. Let $\beta = \|\mathbf{p}''\|_1 / \|\mathbf{p}\|_1$ *be the scaling factor between* $\mathbf{p}''$ *and* $\mathbf{p}$ *in Algorithm 3. Then* $\beta \geq 2^{-k}$.

Proof. Using definitions from the proof of Theorem 4.1, let β_i be the scaling factor between $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. Then we shall prove that $\beta_i \geq \frac{1}{2}$. By telescoping this bound, we will prove the theorem. Recall that any node $v \in V_{i+1} - V_i$ is a freshly added virtual node. Hence, the only contributions to $\mathbf{p}_{i+1}[v]$ come from nodes in V_i. Moreover, any node u can contribute at most $\mathbf{p}_{i+1}[u]$ to the steady-state values of other nodes. Therefore,

$$\sum_{v \in V_{i+1} - V_i} \mathbf{p}_{i+1}[v] \leq \sum_{u \in V_i} \mathbf{p}_{i+1}[u].$$

Adding $\sum_{u \in V_i} \mathbf{p}_{i+1}[u]$ to the above equation, we have

$$\begin{aligned}\sum_{v \in V_{i+1} - V_i} \mathbf{p}_{i+1}[v] + \sum_{u \in V_i} \mathbf{p}_{i+1}[u] &\leq 2 \sum_{u \in V_i} \mathbf{p}_{i+1}[u] \sum_{v \in V_{i+1}} \mathbf{p}_{i+1}[v] \\ &\leq 2 \sum_{u \in V_i} \beta_i \mathbf{p}_i[u] 1 \leq 2\beta_i.\end{aligned}$$

Hence, $\beta = \prod_{i=0}^{k-1} \beta_i \geq 2^{-k}$. □

How does Theorem 4.3 help us? Note that it holds for any valid sequence of transformations. We can then use the sequence of graphs G_i such that G_i is the graph after i phases of edge-disjoint clique–star transformations as described in [Buehrer and Chellapilla 08]. Since only four or five phases are required in practice to obtain nearly the best possible compression, the above theorem then concludes that we lose only four to five bits of floating-point accuracy when using Algorithm 3.

4.2. SALSA on Compressed Graphs

SALSA [Lempel and Moran 00] is a link-analysis algorithm similar to HITS that assigns each web page a separate *authority* score and *hub* score. Let $G(V, E)$ be the query-specific graph under consideration, with W_r and W_c being the row and column normalized versions of E respectively. Then the authority vector $\mathbf{a}$ and hub vector $\mathbf{h}$ are the top eigenvectors (corresponding to the eigenvalue 1) of $W_\mathrm{r}^T W_\mathrm{c}$ and $W_\mathrm{c} W_\mathrm{r}^T$ respectively, satisfying the following recursive definition:

$$\mathbf{a} = W_\mathrm{r}^T \mathbf{h}, \quad \mathbf{h} = W_\mathrm{c}\, \mathbf{a}. \tag{4.5}$$

We can view the above as the following single eigenvalue computation:

$$\begin{bmatrix} \mathbf{a} \\ \mathbf{h} \end{bmatrix} = M \begin{bmatrix} \mathbf{a} \\ \mathbf{h} \end{bmatrix},$$

where M is the $2|V| \times 2|V|$ matrix encoding the equations in (4.5).

Under reasonable assumptions that are described in [Lempel and Moran 00], the solutions $\mathbf{a}$ and $\mathbf{h}$ to the above system are unique and with nonnegative entries. As with PageRank, the power method can be employed to compute these eigenvalues.

We will provide a method to run the algorithm directly on a compressed graph G' to compute authority and hub scores on the original graph G. As expected, we will start with the function Δ_G. However, since SALSA involves pushing authority scores back over in-links to a node, we also need the in-link counterpart of Δ_G. We define this function Λ_G in a manner analogous to Δ_G:

$$\Lambda_G(u) = \begin{cases} 1 & \text{if } u \text{ is real,} \\ \sum_{w \in \delta_{\text{in}}^G(u)} \Lambda_G(w) & \text{if } u \text{ is virtual.} \end{cases}$$

As noted in the case of Δ_G, the values of Λ_G can be precomputed during the operation of the compression algorithm. Similarly, we define the in-degree analogue of Γ_G as

$$\Phi_G(u) = \sum_{w \in \delta_{\text{in}}^G(u)} \Lambda_G(w).$$

The reader can predict that analogous to our scheme for PageRank, we can now design a modeling Markov chain on the compressed graph G' by assigning the probability of forward transition along the edge $uv \in E'$ to be $\Delta_{G'}(v)/\Gamma_{G'}(u)$ and that of reverse transition to be $\Lambda_{G'}(u)/\Phi_{G'}(v)$. This is indeed the case. However, since we deal with two different scores in the case of SALSA, we run into a subtle issue even after these adjustments.

To understand the subtleties involved, let us view the directed graphs as flow networks. Consider an edge $uv \in E$ in the graph G and the corresponding virtual edge $u \to w \to v$ in the compressed graph G'. In the case of PageRank, only one commodity—the PageRank score—flows through the network. Hence the virtual nodes in compressed graphs merely *delay* the flow of PageRank between real nodes. For example, $\mathbf{p}'[u]$ contributes to $\mathbf{p}'[w]$, which in turn contributes to $\mathbf{p}'[v]$ as desired. In case of SALSA, the situation is different.

In the original graph G, the hub score from node u is pushed along a forward edge ($uv \in E$) into the *authority* score bucket of node v, whereas the authority score of node v is pushed along the reverse edge into the *hub* score of node u.

If we attempt to run the SALSA power iterations (albeit with weight adjustments as noted above) unchanged on G', $\mathbf{h}'[u]$ will contribute to $\mathbf{a}'[w]$ but never to $\mathbf{a}'[v]$. This clearly is erroneous modeling of the flow of scores in the original graph, and it stems from the *alternating* behavior of authority and hub scores.

To tackle this issue, we need to draw upon our abstract idea that virtual nodes merely "delay" the flow of scores within the network and hence must not participate in the alternating behavior. (Recall that in the case of PageRank, we barred virtual nodes from jump transitions.) Specifically, for a virtual edge

$u \to w \to v$, we must push the hub score $\mathbf{h}'[u]$ into the hub score $\mathbf{h}'(v)$, which subsequently will contribute to $\mathbf{a}'[v]$ as desired. Indeed, this modification to the definitions—formulated in the equations below—does the trick.

Definition 4.4. (SALSA on an uncompressed graph.)

$$\mathbf{a}_{i+1}[u] = \sum_{v \in \delta_{\text{in}}^{G}(u)} \frac{1}{|\delta_{\text{out}}^{G}(v)|} \mathbf{h}_i(v),$$
$$\mathbf{h}_{i+1}[u] = \sum_{v \in \delta_{\text{out}}^{G}(u)} \frac{1}{|\delta_{\text{in}}^{G}(v)|} \mathbf{a}_i(v).$$

Definition 4.5. (SALSA on a compressed graph.)

$$\mathbf{a}'_{i+1}[u] = \begin{cases} \sum_{v \in \delta_{\text{in}}^{G'}(u)} \frac{\Delta_{G'}(u)}{\Gamma_{G'}(v)} \mathbf{h}'_i(v) & \text{if } u \text{ is real,} \\ \sum_{v \in \delta_{\text{out}}^{G'}(u)} \frac{\Lambda_{G'}(u)}{\Phi_{G'}(v)} \mathbf{a}'_i(v) & \text{if } u \text{ is virtual,} \end{cases}$$

$$\mathbf{h}'_{i+1}[u] = \begin{cases} \sum_{v \in \delta_{\text{out}}^{G'}(u)} \frac{\Lambda_{G'}(u)}{\Phi_{G'}(v)} \mathbf{a}'_i(v) & \text{if } u \text{ is real,} \\ \sum_{v \in \delta_{\text{in}}^{G'}(u)} \frac{\Delta_{G'}(u)}{\Gamma_{G'}(v)} \mathbf{h}'_i(v) & \text{if } u \text{ is virtual.} \end{cases}$$

As a sanity check, observe that our modifications do not alter the operation of SALSA on uncompressed graphs; they simply extend it.

It is a matter of detail now to arrange the above equations into matrix form and to implement power iterations to compute eigenvalues $\mathbf{a}'$ and $\mathbf{h}'$. For ease of exposition, we can view this as computing the eigenvector $\left[\begin{smallmatrix}\mathbf{a}' \\ \mathbf{h}'\end{smallmatrix}\right]$ of the $2|V'| \times 2|V'|$ matrix that encodes the above equations. Let us call this matrix M'.

In contrast to PageRank, the irreducibility and aperiodicity of this Markov chain is not immediately obvious. Aperiodicity can be obtained by introducing a nonzero probability α of nontransition on real nodes, i.e., modifying the equations to

$$\mathbf{a}'_{i+1}[u] = \alpha \mathbf{a}'_i[u] + (1-\alpha) \sum_{v \in \delta_{\text{in}}^{G'}(u)} \frac{\Delta_{G'}(u)}{\Gamma_{G'}(v)} \mathbf{h}'_i(v),$$
$$\mathbf{h}'_{i+1}[u] = \alpha \mathbf{h}'_i[u] + (1-\alpha) \sum_{v \in \delta_{\text{out}}^{G'}(u)} \frac{\Lambda_{G'}(u)}{\Phi_{G'}(v)} \mathbf{a}'_i(v).$$

The irreducibility of M' follows from the irreducibility of M. We now give an outline of the proof. Consider the support graph G_M of matrix M. This graph on $2|V|$ vertices is identical to the graph G' constructed in [Lempel and Moran 00], and it follows that it is bipartite. However, since M contains each edge $uv \in E$ in both directions, with the connectivity assumptions stated in

[Lempel and Moran 00], G_M has a single strongly connected component. Now the irreducibility of M' follows from that of M by the observations that every path between real nodes is kept intact during the compression and that every virtual node has a path to and from a real node.

The following theorem proves the correctness of our solution. We omit the proof, which is almost identical to that of Theorems 4.2 and 4.3.

Theorem 4.6. *Let* $\left[\begin{smallmatrix}\mathbf{a}'\\ \mathbf{h}'\end{smallmatrix}\right]$ *and* $\left[\begin{smallmatrix}\mathbf{a}\\ \mathbf{h}\end{smallmatrix}\right]$ *be top eigenvectors of* M' *and* M *respectively. Then,*

(1) $\mathbf{a}'[u] = \beta\mathbf{a}[u]$ *and* $\mathbf{h}'[u] = \beta\mathbf{h}[u]$ *for all* $u \in V(G)$*;*

(2) *if* k *is the depth of compression of* G'*, then* $\beta \geq 2^{-k}$*.*

4.3. Comparison of the Two Approaches

We now summarize the advantages and disadvantages of computing PageRank and SALSA with the *black-box multiplication* algorithms of Section 3, and the *Markov chain* algorithms from Section 4.

Although the Markov chain algorithms from Section 4 converge to eigenvectors that are similar to the corresponding eigenvectors on the uncompressed graph, the number of iterations required may change. Since the compression via virtual nodes introduces longer paths in the graph, it may require a larger number of power iterations to converge to the desired accuracy. We remark that the number of iterations required may increase by at most a factor of the longest virtual edge.

The black-box methods from Section 3 simply speed up each individual iteration, so the number of iterations required is identical. As a result, the black-box methods usually result in better speedup ratios.

The number of iterations required by the Markov chain algorithm and the overall comparison in speedup ratios is examined experimentally in Section 6.

Since the Markov chain methods involve only changing transition probabilities, an existing implementation of say PageRank can be run directly on the compressed graph, with appropriately modified weights, to compute PageRank in the original uncompressed graph. This allows us to take advantage of existing optimized implementations and heuristics.

Both methods can be efficiently parallelized. Black-box multiplication requires that certain sets of virtual nodes be pushed before others, requiring a small number of global synchronizations in each iteration. For the Markov chain method, any parallel algorithm for computing PageRank or SALSA can be used, some of which require few if any global syncs [McSherry 05, Kempe and McSherry 08]. In a large-scale parallel implementation, the cost of global syncs can be prohibitive, so in this case the Markov chain method may be preferable.

We remark that the Markov chain methods are not directly applicable to HITS, because the scaling step involved after every iteration destroys correctness.

Finally, the black-box method for SALSA needs lists of in-links of virtual nodes and separate orderings on virtual nodes with respect to in-links and out-links. This adds to the storage required for the compressed graph, apart from slowing the algorithm down to a small extent.

5. Applications

Many algorithms can be sped up using compressed graphs, but they require techniques different from those described in this paper. Several examples were considered in [Feder and Motwani 95], including algorithms for computing breadth-first search and other shortest-path algorithms.

Here we describe a few examples of algorithms that can be written in terms of adjacency matrix multiplication, and thus can be sped up using compressed-multiply as a subroutine. Many of these algorithms perform several iterations, and each iteration is dominated by the time required to compute the matrix–vector product.

Random walk distributions. The task is to compute the distribution of a random walk after T steps, starting from the initial distribution p_0. This can be done in T iterations by computing $p_{t+1} = E^T D^{-1} p_t$, where D is the diagonal matrix such that $D(i,i)$ is the out-degree of vertex i. Given p_t, we first compute $D^{-1}p_t$ in time $O(|V|)$, and then use compressed-multiply to compute $p_{t+1} = E^T(D^{-1}p_t)$. The time per iteration is $O(|V|) + O(|E'| + |V'|) = O(|E'| + |V'|)$.

Eigenvectors and spectral methods. The largest eigenvectors of the adjacency matrix E can be computed using the power method, which requires repeatedly multiplying an initial vector by E. In each iteration we must also subtract the projections onto the larger eigenvectors and normalize, which can be done in $O(|V|)$ time per iteration, so the time required per iteration is $O(|E'|+|V'|)$. The power method can also be used to compute the few smallest eigenvectors of the Laplacian matrix $L = D-E$, which are useful for spectral partitioning [Chung 97] and transductive learning on graphs [Zhou et al. 07].

Top singular vectors. The top singular vectors of E, which are the top eigenvectors of E^TE and EE^T, can also be computed using the power method. A single iteration requires first multiplying by E^T using the compressed out-link graph and then multiplying by E using the compressed in-link graph. Since the compressed in-link graph and out-link graph have the same values of $|E'|$ and $|V'|$, the time per iteration is $O(|E'| + |V'|)$.

As an application, [Kannan and Vinay 99] introduced an algorithm for finding dense subgraphs of directed graphs, whose main step is computing the top singular vectors of E.

Estimating the size of neighborhoods. An algorithm is introduced in [Becchetti et al. 06] for estimating the number of nodes within r steps of each node in a graph, based on probabilistic counting. Each node stores a k-bit vector initialized to all zeros. Initially, some randomly chosen bit positions are flipped to ones. The algorithm then performs r iterations, and in each iteration each node's bit vector becomes the bitwise OR of its own bit vector and the bit vectors of its neighbors. This iteration can be viewed as multiplication by the adjacency matrix, where the sum operation is replaced by bitwise OR.

Multiplying a sparse vector by an adjacency matrix. Given a sparse vector $\mathbf{x}$, where S is the set of vertices with nonzero entries in $\mathbf{x}$, we can compute $E \cdot \mathbf{x}$ using the method from Section 3, except we need to push only from real nodes with nonzero values, and through the virtual edges incident on those nodes. This requires time proportional to $L \cdot \text{out-degree}(S)$, where L is an upper bound on the length of a virtual edge. Similarly, we can compute $A^T x$ in time proportional to $L \cdot \text{in-degree}(S)$. These operations require random access to the adjacency information of the compressed graph, as opposed to the algorithms in earlier sections that require only sequential disk access to the compressed graph. Using sparse vector multiplication as a primitive, we can implement algorithms that examine only a portion of the entire graph, including algorithms for finding communities [Andersen and Lang 06] and computing personalized PageRank [Haveliwala 03, Jeh and Widom 03].

Potential extension of the stochastic methods. We would like to point out that the stochastic methods from Section 4 are specific to PageRank and SALSA. We cannot apply them to compute the steady state of a general stochastic matrix. In the case of PageRank and SALSA, the virtual-node compression preserves a memoryless property that allows us to construct a Markov chain on the compressed graph that simulates the original stochastic process.

6. Experiments

We implemented the methods discussed in Sections 3 and 4 for PageRank and SALSA on web graphs compressed using techniques described in [Buehrer and Chellapilla 08]. We compared them against standard versions of PageRank and SALSA running on uncompressed graphs.

	Uncompressed		Compressed		Ratio
	# Nodes	# Edges	# Nodes	# Edges	
eu-2005	862,664	19,235,140	1,196,536	4,429,375	4.34
uk-2005	39,459,925	936,364,282	47,482,140	151,456,024	6.18

Table 1. Data sets.

	Uncompressed	Black-Box	Markov Chain
Time/Iteration (sec)	5.37	1.58	1.50
No. of Iterations	19	19	50
Speedup	1	3.40	1.36

Table 2. PageRank, eu-2005.

System. We ran the algorithms on a standard workstation with 16 GB RAM and a quad-core Intel Xeon processor at 3.0 GHz. Only one of the available cores was used, since the implementation is single-threaded. This does not limit the generality of the performance boost, since as discussed in Section 4.3, the Markov chain methods are highly parallelizable and the black-box multiplication methods require only a small amount of synchronization between threads.

Implementation. Our programs strictly followed the sequential file access paradigm, wherein the graph files are stored only on disk in the adjacency list format. We used $O(|V|)$ bits of random access memory to hold the intermediate score vectors.

Data sets. We used the public data sets eu-2005 and uk-2005 hosted by *Laboratory for Web Algorithmics*[1] at *Università degli Studi di Milano*. Many of these web graphs were generated using *UbiCrawler* [Boldi et al. 04] by various labs in the search community. Statistics for these two data sets appear in Table 1. The comparative performance of PageRank algorithms is tabulated in Tables 2 and 3. Note that the speedup ratios consider the total time required, as opposed to the time per iteration, since the numbers of iterations differ.

Both the black-box and Markov chain methods show an improvement in the time per iteration over the uncompressed versions of the algorithms. However, as described in Section 4.3, the Markov chain method requires more iterations to converge to the same accuracy, bringing down the net performance boost. This is due to the introduction of longer paths in the graph during compression. Also note that the overall speedup ratios do not exactly match the reduction in the number of edges. This is due to the fact that both these algorithms perform some bookkeeping operations such as zeroing the variables, which require time

[1]See http://law.dsi.unimi.it.

	Uncompressed	Black-Box	Markov Chain
Time/Iteration (sec)	263.52	60.80	60.06
No. of Iterations	21	21	53
Speedup	1	4.33	1.74

Table 3. PageRank, uk-2005.

	Uncompressed	Black-Box	Markov Chain
Time/Iteration (sec)	5.48	2.37	1.97
No. of Iterations	91	91	100
Speedup	1	2.31	2.70
Storage Reduction	1	2.36	3.21

Table 4. SALSA; eu-2005.

proportional to the number of nodes. These parts of the algorithm are not sped up, and in fact require slightly more operations in the compressed graphs due to the increased number of nodes.

Results for SALSA are depicted in Tables 4 and 5. Again, the algorithms achieve significant speedup over the uncompressed versions. We observe that in the case of SALSA, the Markov chain method performs better than the black-box method. This appears to be due to two reasons:

1. The difference in the number of iterations required between the two methods is smaller for SALSA than for PageRank. This is because the convergence rate of SALSA is less sensitive to path lengths.

2. To compute SALSA, we need to perform separate passes over out-links and in-links of the virtual nodes. As explained in Section 4.3, the black-box algorithm for SALSA requires distinct orderings of the virtual nodes for the in-link graph and out-link graph. The Markov chain method has a slight advantage here because it can use the same ordering of virtual nodes.

We remark that the Markov chain method for SALSA also requires slightly less storage on disk, since it needs to store only one ordering of the virtual nodes.

Precision of the Markov chain methods. In practical terms, Theorems 4.3 and 4.6 guarantee that at most k bits of precision are lost during the run of PageRank or SALSA on

	Uncompressed	Black-Box	Markov Chain
Time/Iteration (sec)	265.94	84.22	68.80
No. of Iterations	104	104	124
Speedup	1	3.16	3.24
Storage Reduction	1	3.47	4.54

Table 5. SALSA; uk-2005.

a compressed graph, where k is the depth of the compression. However, during the empirical studies outlined in this section, we found that the scaling constant β as defined in Theorem 4.3 was always greater than $1/2$, which corresponds to the loss of at most one bit of information on average. Therefore, loss of precision does not seem to be an issue when these methods are used in practice.

References

[Andersen and Lang 06] Reid Andersen and Kevin J. Lang. "Communities from Seed Sets." In *Proceedings of the 15th international conference on World Wide Web*, pp. 223–232. New York: ACM, 2006.

[Becchetti et al. 06] Luca Becchetti, Carlos Castillo, Debora Donato, Stefano Leonardi, and Ricardo Baeza-Yates. "Using Rank Propagation and Probabilistic Counting for Link-Based Spam Detection." In *Proceedings of the Workshop on Web Mining and Web Usage Analysis (WebKDD)*. New York: ACM Press, 2006.

[Boldi and Vigna 04a] Paolo Boldi and Sebastiano Vigna. "The Webgraph Framework I: Compression Techniques." In *Proceedings of the 13th International Conference on World Wide Web*, pp. 595–602. New York: ACM Press, 2004.

[Boldi and Vigna 04b] Paolo Boldi and Sebastiano Vigna. "The Webgraph Framework II: Codes for the World-Wide Web." In *Proceedings of the Conference on Data Compression*, p. 528. Washington, DC: IEEE Computer Society, 2004.

[Boldi et al. 04] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. "Ubicrawler: A Scalable Fully Distributed Web Crawler. *Software: Practice & Experience* 34:8 (2004), 711–726.

[Brin and Page 98] Sergey Brin and Lawrence Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine." *Computer Networks and ISDN Systems* 30:1–7 (1998), 107–117.

[Buehrer and Chellapilla 08] Gregory Buehrer and Kumar Chellapilla. "A Scalable Pattern Mining Approach to Web Graph Compression with Communities." In *WSDM*, pp. 95–106, 2008.

[Chung 97] Fan R. K. Chung. *Spectral Graph Theory*, CBMS Regional Conference Series in Mathematics 92. Providence: American Mathematical Society, 1997.

[Feder and Motwani 95] Tomás Feder and Rajeev Motwani. "Clique Partitions, Graph Compression and Speeding-Up Algorithms." *J. Comput. Syst. Sci.* 51:2 (1995), 261–272.

[Feder et al. 03] Tomás Feder, Adam Meyerson, Rajeev Motwani, Liadan O'Callaghan, and Rina Panigrahy. "Representing Graph Metrics with Fewest Edges." In *STACS 2003: 20th Annual Symposium on Theoretical Aspects of Computer Science Berlin, Germany, February 27–March 1, 2003, Proceedings*, Lecture Notes in Computer Science 2607, pp. 355–366. Berlin: Springer, 2003.

[Haveliwala 03] Taher H. Haveliwala. "Topic-Sensitive Pagerank: A Context-Sensitive Ranking Algorithm for Web Search." *IEEE Trans. Knowl. Data Eng.* 15:4 (2003), 784–796.

[Jeh and Widom 03] Glen Jeh and Jennifer Widom. "Scaling Personalized Web Search." In *Proceedings of the 12th International Conference on World Wide Web*, pp. 271–279. New York: ACM, 2003.

[Kannan and Vinay 99] R. Kannan and V. Vinay. "Analyzing the Structure of Large Graphs." Manuscript, 1999.

[Karande et al. 09] Chinmay Karande, Kumar Chellapilla, and Reid Andersen. "Speeding Up Algorithms on Compressed Web Graphs." In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, pp. 272–281. New York: ACM, 2009.

[Kempe and McSherry 08] David Kempe and Frank McSherry. "A Decentralized Algorithm for Spectral Analysis." *J. Comput. Syst. Sci.* 74:1 (2008), 70–83.

[Kleinberg 99] Jon M. Kleinberg. "Authoritative Sources in a Hyperlinked Environment." *Journal of the ACM* 46:5 (1999), 604–632.

[Kumar et al. 99] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. "Trawling the Web for Emerging Cyber-Communities." *Computer Networks* 31:11–16 (1999), 1481–1493.

[Lempel and Moran 00] R. Lempel and S. Moran. "The Stochastic Approach for Link-Structure Analysis (SALSA) and the TKC Effect." *Computer Networks (Amsterdam, Netherlands: 1999)* 33:1–6 (2000), 387–401.

[McSherry 05] Frank McSherry. "A Uniform Approach to Accelerated PageRank Computation." In *Proceedings of the 14th International Conference on World Wide Web*, pp. 575–582. New York: ACM, 2005.

[Page et al. 98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. "The Pagerank Citation Ranking: Bringing Order to the Web." Technical report, Stanford Digital Library Technologies Project, 1998.

[Zhou et al. 07] Dengyong Zhou, Christopher J. C. Burges, and Tao Tao. "Transductive Link Spam Detection." In *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*, pp. 21–28, New York: ACM, 2007.

Chinmay Karande, Georgia Institute of Technology, College of Computing, 266 Ferst Drive, Atlanta, GA 30332 (ckarande@cc.gatech.edu)

Kumar Chellapilla, Microsoft, One Microsoft Way, Redmond, WA 98052 (kumarc@microsoft.com)

Reid Andersen, Microsoft, One Microsoft Way, Redmond, WA 98052 (reidan@microsoft.com)

Received June 30, 2009; accepted June 3, 2010.