Community Detection Using Time-Dependent Personalized PageRank

Haim Avron HAIMAV@US.IBM.COM
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

Lior Horesh LHORESH@US.IBM.COM
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598

Abstract

Local graph diffusions have proven to be valuable tools for solving various graph clustering problems. As such, there has been much interest recently in efficient local algorithms for computing them. We present an efficient local algorithm for approximating a graph diffusion that generalizes both the celebrated personalized PageRank and its recent competitor/companion - the heat kernel. Our algorithm is based on writing the diffusion vector as the solution of an initial value problem, and then using a waveform relaxation approach to approximate the solution. Our experimental results suggest that it produces rankings that are distinct and competitive with the ones produced by high quality implementations of personalized PageRank and localized heat kernel, and that our algorithm is a useful addition to the toolset of localized graph diffusions.

1. Introduction

In community detection problems (i.e., graph clustering problems), one seeks to identify sets of nodes in a graph that are both internally cohesive and well separated from the rest of the graph. Such sets are then referred to as communities or clusters. In one important variant, the goal is to build a community around a given seed node or set of seed nodes. That is, the algorithm is given, as an input, a node (or nodes) in the graph, and the goal is to find a cluster in which it is a member.

One popular technique for identifying communities using seed nodes is to use local graph diffusions (Andersen et al., 2006; Chung, 2007). The general framework is as follows. Using the seed, a diffusion vector is computed. The diffusion vector is reweighted based on the degrees, and the nodes are sorted according to their magnitude in the reweighted diffusion vector. The community is found by making a sweep over the nodes according to their rank, selecting the prefix that minimizes (or maximizes) some scoring function. If the input vector is sparse, most of the entries in the diffusion vectors tend to be tiny, in which case a sparse approximation is viable (the exact diffusion vector is generally dense), and this is allows algorithms that work even on massive graphs. Algorithms that find sparse approximations to diffusion vectors are *local graph diffusion algorithms.*

One widely used scoring function is *conductance*. The conductance of a subset of nodes S is given by

$$\phi(S) \equiv \frac{\partial S}{\min(\mathbf{vol}\left(S\right), \mathbf{vol}\left(\bar{S}\right))},$$

where $\bar{S}$ is the complementary set of nodes, ∂S is the number of edges with one end-point in S and the other in $\bar{S}$, and $\mathbf{vol}\left(S\right)$ is the sum of degrees of nodes in S. Selecting sets with low conductance tends to produce high quality clusters (Yang & Leskovec, 2015), and the performance of the aforementioned algorithm with respect to minimizing the conductance can be rigorously analyzed for certain diffusions (Andersen et al., 2006; Chung, 2009). However, we stress that other scoring functions have been applied successfully (Yang & Leskovec, 2015).

The classical graph diffusion vector is the PageRank vector (Brin & Page, 1998). Assume, without loss of generality, that the set of nodes is $\{1, \ldots, n\}$. Let $\mathbf{A} \in \mathbb{R}^{n \times n}$ be the adjacency matrix of the graph, and let $\mathbf{D} \in \mathbb{R}^{n \times n}$ be a diagonal matrix with the degrees on the diagonal. Let $\mathbf{P} \equiv \mathbf{A}\mathbf{D}^{-1}$ be the random walk transition matrix. The PageRank vector is equal to

$$\mathbf{p} \equiv (1-\alpha)(\mathbf{I}_n - \alpha\mathbf{P})^{-1}\mathbf{s}$$

where $\alpha \in (0,1)$ and $\mathbf{s} \in \mathbb{R}^n$ are parameters. The vector $\mathbf{s}$ is typically referred to as the *teleportation vector.* The

Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37.

PageRank vector $\mathbf{p}$ can be interpreted as the stationary distribution of a random walk with restart, and there are also other equivalent formulations. When $\mathbf{s}$ is sparse this vector is often referred to as a *personalized PageRank vector*. Andersen et al. (2006) described an efficient efficient local algorithm for personalized PageRank.

Recently, the heat kernel diffusion vector (Chung, 2007) has received attention in the literature. The heat kernel vector is equal to

$$\mathbf{h} \equiv \exp\left\{-\gamma(\mathbf{I}_n - \mathbf{P})\right\}\mathbf{s}$$

where $\gamma > 0$ is some parameter. Kloster & Gleich (2014) recently described an efficient local algorithm for the heat kernel, and experimentally showed that the heat kernel tends to produce smaller and more realistic communities than the ones produced using the PageRank diffusion, with only a modest increase in conductance.

In this paper we consider a special case of the dynamical PageRank diffusion recently introduced by Gleich & Rossi (2014). Fix parameters $\gamma > 0$ and $\alpha \in [0, 1]$. The diffusion vector $\mathbf{x}$, which we refer to as the *time-dependent PageRank vector*, is equal to $\mathbf{x}(\gamma)$ where $\mathbf{x}(\cdot)$ is the solution to the initial value problem

$$\begin{aligned}\mathbf{x}'(t) &= (1-\alpha)\mathbf{s} - (\mathbf{I}_n - \alpha\mathbf{P})\mathbf{x}(t) \\ \mathbf{x}(0) &= \mathbf{s}, \quad t \in [0, \gamma]\,. \end{aligned} \tag{1}$$

It can be shown that (Gleich & Rossi, 2014)

$$\begin{aligned}\mathbf{x} \;=\; & (1-\alpha)(\mathbf{I}_n - \alpha\mathbf{P})^{-1}\mathbf{s} + \\ & \exp\left\{-\gamma(\mathbf{I}_n - \alpha\mathbf{P})\right\}\left(\mathbf{s} - (1-\alpha)(\mathbf{I}_n - \alpha\mathbf{P})^{-1}\mathbf{s}\right).\end{aligned} \tag{2}$$

Time-dependent PageRank is a generalization of *both* PageRank and heat kernel. If we fix α and let γ go to infinity, the time-dependent PageRank vector $\mathbf{x}$ converges to the PageRank vector $\mathbf{p}$. If we set $\alpha = 1.0$, time-dependent PageRank vector $\mathbf{x}$ is exactly equal to the heat kernel vector $\mathbf{h}$. Consequently, one can hope that certain non-degenerate combinations of α and γ (perhaps ones that are dependent on the application and/or the graph) will produce better diffusion vectors for downstream use.

The main contribution of this paper is an efficient local algorithm for approximating the time-dependent personalized PageRank vector $\mathbf{x}$. As such, our algorithm can also be viewed as a new algorithm for both the heat kernel diffusion and the personalized PageRank diffusion. The proposed algorithm is deterministic and simple (we give a detailed pseudo-code in the supplementary material). Our algorithm tends to access the out-links much less than existing algorithms for personalized heat kernel and PageRank, and as such more suitable than those algorithms when out-link access is somewhat expensive. We experimentally compare communities produced using our algorithm with communities produced using the heat kernel and using PageRank. Our algorithm tends to generate smaller communities than both heat kernel and PageRank, with roughly the same conductance. In our experimental setup, it had similar performance to heat kernel in detecting ground-truth communities.

An open-source implementation of the algorithm is available through the libSkylark library (`http://xdata-skylark.github.io/libskylark/`).

2. Preliminaries

2.1. Notation

We denote scalars using Greek letters or using $t, s, \ldots$. Vectors are denoted by $\mathbf{x}, \mathbf{y}, \ldots$ and matrices by $\mathbf{A}, \mathbf{B}, \ldots$. We use the convention that vectors are column-vectors. We denote vector-valued functions from $[0, \gamma]$ (or any other interval implied by the text) to $\mathbb{R}^n$ by $\mathbf{x}(\cdot), \mathbf{y}(\cdot), \ldots$, while $\mathbf{x}(t), \mathbf{y}(t)$ denotes their evaluation (which is a vector in $\mathbb{R}^n$) at a specific point t. Accordingly, $x(\cdot), y(\cdot), \ldots$ denote scalar-valued functions, and $x(t), y(t), \ldots$ their evaluation at t. For a vector (or vector-valued function), lower case letter with an index denotes the value of coordinate i of the vector (resp. vector-valued function). For example, x_i (resp. $x_i(\cdot)$) denotes the ith entry of $\mathbf{x}$ (resp. $\mathbf{x}(\cdot)$). Note that x_i is different from $\mathbf{x}_i$; the former is a scalar, while the latter is a vector indexed by i.

2.2. Diffusion Coefficients

Since both PageRank and heat kernel are a special cases of time-dependent PageRank, any rigorous theoretical guarantee on either one can be mimicked using the time-dependent PageRank. Consequently, one can hope that some non-degenerate combinations of α and γ will offer better results. While there is no rigorous analysis supporting this assessment, inspection of diffusion coefficients can provide insight into possibly favorable behaviors of time-dependent PageRank.

Graph diffusion vectors, like PageRank and heat kernel, can be written as an infinite series

$$\mathbf{f} = \sum_{k=0}^{\infty} \alpha_k \mathbf{P}^k \mathbf{s}$$

where $\sum_{k=0}^{\infty} \alpha_k = 1$. The terms α_k are the *diffusion coefficients*, which serve as weights to the distribution of random walks of the corresponding lengths.

For PageRank, the diffusion expansion is

$$\mathbf{p} = (1-\alpha)\sum_{k=0}^{\infty} \alpha^k \mathbf{P}^k \mathbf{s}\,,$$

and for the heat kernel it is

$$\mathbf{h} = e^{-\gamma} \sum_{k=0}^{\infty} \frac{\gamma^k}{k!} \mathbf{P}^k \mathbf{s} \,.$$

Based on (2), it is easy to show that

$$\mathbf{x} = \sum_{k=0}^{\infty} \left[(1-\alpha)\alpha^k \left(1 - e^{-\gamma} \sum_{r=0}^{k} \frac{\gamma^r}{r!} \right) + e^{-\gamma} \frac{\alpha^k \gamma^k}{k!} \right] \mathbf{P}^k \mathbf{s} \,.$$

The diffusion coefficients of PageRank decay at a fixed rate. Thus, if the decay for low-indices is slow (i.e. short paths have roughly the same coefficient), the decay is very slow overall, and excessively long paths will have rather high coefficients, thereby encouraging large communities. Slow decay also poses an algorithmic challenge, since the algorithm needs to allocate significant weights to very long paths. If we reduce α so that long paths have very small coefficients, we ran the risk of having a strong decay for the initial coefficients, perhaps resulting in unduly small communities.

As for the heat kernel, while the coefficients eventually decay to zero very quickly (since $k!$ grows much faster than γ^k), there is an initial phase in which they actually *grow* (this is because $k!$ is indepenent of γ, and thus γ^k initially grows much faster than $k!$). Thus, the coefficients of the heat kernel display an hump in the lower coefficients, which might cause short paths to get very small weight.

Inspecting the diffusion coefficients of time-dependent PageRank, we see that for small indices, the behavior is quite similar to that of PageRank (as long as α is small enough), but for large k the decay is similar to that of heat kernel. Thus, by carefully choosing α and γ, we can entertain a slow initial decay, followed by very rapid decay beyond some point. This is illustrated in Figure 1 (left).

Figure 1 (right) reports the cumulative sum of the diffusion coefficients as the index increases. Once the cumulative sum gets very close to 1.0, the weights of remaining long paths are too small for them to affect the final diffusion vector. Thus, when the cumulative sum quickly approaches 1.0 there is more hope for very fast algorithms.

2.3. Relaxation Methods and Waveform Relaxation

Relaxation is a general technique for solving linear equations of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$. In the most general form, the matrix is split $\mathbf{A} = \mathbf{B}_k - \mathbf{C}_k$ (here k is an iteration index), and the iteration is written as $\mathbf{x}^{(k+1)} = \mathbf{B}_k^{-1}(\mathbf{C}_k \mathbf{x}^{(k)} + \mathbf{b})$. By carefully choosing $\mathbf{B}_k$ and $\mathbf{C}_k$, one can ensure that $\mathbf{x}^{(k+1)}$ is identical to $\mathbf{x}^{(k)}$ in all but one coordinate. Such methods are called *coordinate relaxation methods*, and they are closely related to the Gauss-Seidel and Gauss-Southwell iteration. Several competitive local algorithms for personalized PageRank and heat kernel employ a coordinate relaxation approach (Kloster & Gleich, 2014).

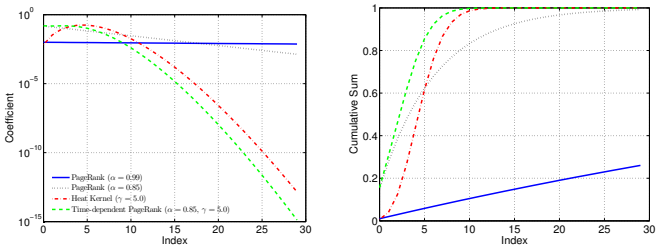

Figure 1. Diffusion coefficients (α_ks) for different diffusions. The left graph plots the coefficients themselves, while the right graph shows a cumulative sum of the coefficients.

Waveform relaxation, originally developed at the beginning of the 1980s for the simulation of electronic circuits, is a generalization of relaxation to solving initial value problems of the form $\mathbf{x}'(t) = F(t, \mathbf{x}(t))$. The basic idea is to introduce new functions $G_k(\cdot,\cdot,\cdot)$ such that $F(t,\mathbf{x}) = G_k(t,\mathbf{x},\mathbf{x})$. To obtain $\mathbf{x}^{(k+1)}(\cdot)$ from $\mathbf{x}^{(k)}(\cdot)$ we solve the initial value problem $(\mathbf{x}^{(k+1)})'(t) = G_k(t, \mathbf{x}^{(k+1)}(t), \mathbf{x}^{(k)}(t))$. By choosing $G_k(\cdot,\cdot,\cdot)$ carefully we can, again, ascertain $\mathbf{x}^{(k+1)}(\cdot)$ to be identical to $\mathbf{x}^{(k)}(\cdot)$ in all but one coordinate. This form of waveform coordinate relaxation is the basis for our method.

3. Algorithm

As explained earlier, diffusion-based community detection algorithms start by computing a diffusion vector, which is used rank the vertices and perform a sweep. Our goal is to integrate the time-dependent personalized PageRank (2) vector in this framework, instead of the personalized PageRank vector or the heat kernel vector. This section describes our algorithm for computing (2).

The exact time-dependent personalized PageRank vector is completely dense for connected graphs. Computing it exactly is too expensive for the purpose of detecting local communities. It is common practice to find approximate vectors that are localized (i.e., have only a small number of non-zeros). Since the main purpose of the diffusion vector is to form a ranking of the nodes by sorting their degree reweighted diffusion values, a natural objective is to seek an approximate diffusion vector $\mathbf{y}$ such that

$$\|\mathbf{D}^{-1}(\mathbf{x} - \mathbf{y})\|_\infty < \epsilon \,.$$

Our algorithm constructs a vector-valued function that approximates the diffusion vector for all $t \in [0, \gamma]$. That is, the algorithm constructs a vector function $\mathbf{y}(\cdot)$ such that

$$\|\mathbf{D}^{-1}(\mathbf{x}(t) - \mathbf{y}(t))\|_\infty < \epsilon \tag{3}$$

for all $t \in [0, \gamma]$. In the above $\mathbf{x}(\cdot)$ is the solution to (1).

Remark. Some of the algorithms for approximate diffusion also guarantee that the error is one sided, i.e. $\mathbf{y} \leq \mathbf{x}$. That can be achieved by our algorithm by solving to $\epsilon/2$ approximation error, and then applying a non-uniform shift to $\mathbf{y}$. This shift does not affect the ranking of the nodes.

Let $\mathbf{y}(\cdot)$ be some candidate solution. Following Botchev et al. (2013) we define the *residual* $\mathbf{r}(\cdot)$ relative to $\mathbf{y}(\cdot)$ to be

$$\mathbf{r}(t) \equiv (1-\alpha)\mathbf{s} - (\mathbf{I}_n - \alpha\mathbf{P})\mathbf{y}(t) - \mathbf{y}'(t)\,.$$

The residual is a powerful tool for analyzing approximate solutions to initial value problems (Botchev et al., 2013). In particular, it can be used to certify that (3) is met by $\mathbf{y}(\cdot)$, as the following proposition shows (the proof, like all proofs, appears in the supplementary material):

Proposition 1. *Let d_i denote the ith diagonal entry of $\mathbf{D}$ (which is the degree of vertex i). Suppose that $\mathbf{y}(0) = \mathbf{s}$. If*

$$\|r_i(\cdot)\|_\infty < \frac{(1-\alpha)d_i\epsilon}{1-\exp((\alpha-1)\gamma)} \tag{4}$$

(if $\alpha = 1$ then substitute $1-\exp((\alpha-1)\gamma)/(1-\alpha)$ with γ) holds for all $i = 1,\ldots,n$, then (3) holds for all $t \in [0,\gamma]$.

Algorithm Outline. The key observation is that if we initialize $\mathbf{y}(t) = \mathbf{s}$ then most coordinates are not in violation of (4). Thus, if we use a simple waveform relaxation approach, in which we iteratively target coordinates in which (4) is violated, the iterates wo;; tend to stay localized. In particular, our algorithm maintains a queue of violating coordinates. In each iteration, a coordinate is extracted from the queue and the corresponding coordinate of $\mathbf{y}(\cdot)$ is perturbed so that the corresponding residual is small enough. This, in turn, can cause new coordinates to violate (4). Newly violating coordinates (if any) are then inserted into the queue. The algorithm terminates when the queue is empty. To allow these various steps to be implemented in standard computational models (e.g. RAM model), we restrict the coordinates of $\mathbf{y}(\cdot)$ to be polynomials of some fixed degree. We remark that our algorithm is similar in structure and spirit to "push" algorithms for personalized PageRank and heat kernel (Andersen et al., 2006; Kloster & Gleich, 2014).

3.1. Waveform Relaxation Approach

Proposition 1 suggests the following waveform relaxation approach: identify an index i for which $\|r_i(\cdot)\|_\infty$ is too large, modify $y_i(\cdot)$ so to force $\|r_i(\cdot)\|_\infty$ to be zero, while keeping $y_j(\cdot)$ for $j \neq i$ as-is. This leads to the following proto-algorithm:

1. Initialize $\mathbf{y}(t) = s$.
2. While there exists an i such that
$$\|r_i(\cdot)\|_\infty > \frac{(1-\alpha)d_i\epsilon}{1-\exp((\alpha-1)\gamma)}$$
   (a) Select such an i arbitrarily.
   (b) Set $y_i(\cdot)$ to the solution of the ODE
$$\begin{aligned} y'(t) &= -y(t) + \alpha\sum_{j=1}^{n}\mathbf{P}_{ij}y_j(t) + (1-\alpha)s_i \\ y(0) &= s_i \end{aligned} \tag{5}$$

The reason we use the term "proto-algorithm" is that in the above form, the proto-algorithm cannot be implemented in usual computational models (e.g. RAM model), as it abstractly operates on functions. We address this issue in the next subsection.

3.2. Restricting the Function Space

To turn the proto-algorithm into an actual algorithm, we restrict the functions to be polynomials of degree equal or smaller than some fixed degree N. That is, each coordinate of $\mathbf{y}(\cdot)$ is a polynomial of degree at most N. As a consequence, each coordinate of $\mathbf{r}(\cdot)$ is also a polynomial of degree at most N. How N is chosen is explained in the next subsection. For now, it suffices to keep in mind that N depend on γ and ϵ, and that it tends to grow slowly. For example, for $\gamma = 5.0$ and $\epsilon = 10^{-4}$, $N = 14$ suffices.

Let $\mathbb{P}_N$ be the space of polynomials of degree at most N. We use a pseudo-spectral approach of representing polynomials in $\mathbb{P}_N$ by their values at predefined grid points[1]. Let $t_0,\ldots,t_N$ be the result of scaling and shifting the Chebyshev points of the second kind from $[-1,1]$ to $[0,\gamma]$, i.e.

$$t_j = (\cos(j\pi/N)+1)\gamma/2, \quad j = 0,\ldots,N\,.$$

Given a set of values $y_0,\ldots,y_N$ there is unique polynomial $p(\cdot)$ which interpolates $y_j = p(t_j)$, which means we can use the vector $\left[\begin{array}{ccc} p(t_0) & \cdots & p(t_N)\end{array}\right]^T$ to represent $p(\cdot) \in \mathbb{P}_N$. We denote by $\mathbb{S}_N$ the operator mapping a scalar function to a $N+1$-dimensional vector of its point-samples at $t_0,\ldots,t_N$, i.e.

$$\mathbb{S}_N[f(\cdot)] \equiv \left[\begin{array}{ccc} f(t_0) & \cdots & f(t_N)\end{array}\right]^T\,.$$

Our algorithm uses $\mathbf{y}_i \equiv \mathbb{S}_N[y_i(\cdot)]$ to represent $y_i(\cdot)$. The derivative of a degree N polynomial is a degree $N-1$ polynomial, so $r_i(\cdot) \in \mathbb{P}_N$. Thus, we can use $\mathbf{r}_i \equiv \mathbb{S}_N[r_i(\cdot)]$ to represent $r_i(\cdot)$.[2]

[1] Our approach is inspired by the CHEBFUN library (Driscoll et al., 2014; Battles & Trefethen, 2004). In fact, early prototypes simply implemented the proto-algorithm using CHEBFUN.

[2] The reason we do not use the coefficients in the monomial basis is because that will not result in a stable algorithm. The reason we use non-equidistant grid points is that equidistant grid point will result in a non-stable algorithm as well due to the Runge phenomenon. See Trefethen (2000) and Boyd (1989).

We now show how the various operations in the proto-algorithm are implemented in our algorithm. These operations are:

- Taking derivative in order to compute residuals.
- Computing the infinity norm of residual entries for testing convergence.
- Solving the ODE (5).

3.2.1. Derivative

Derivative is a linear operation on $\mathbb{P}_N$, so there exists a matrix $\mathbf{\Xi} \in \mathbb{R}^{(N+1)\times(N+1)}$ such that for every $p(\cdot) \in \mathbb{P}_N$

$$\mathbb{S}_N[p'(\cdot)] = \mathbf{\Xi}\mathbb{S}_N[p(\cdot)]\,.$$

The following explicit formula for $\mathbf{\Xi}$ is based on known formulas for the $[-1,1]$ domain (Boyd, 1989)

$$\mathbf{\Xi}_{ij} = \begin{cases} \gamma(1+2N^2)/12 & i=j=0 \\ -\gamma(1+2N^2)/12 & i=j=N \\ \gamma x_j/(4-4x_j^2) & i=j;\ 0<j<N \\ (-1)^{i+j}p_i/(2p_jx_i - p_jx_j) & i\neq j \end{cases}$$

where $x_i = \cos(\pi i/N)$, $p_0 = p_N = 2$, and $p_j = 1$ otherwise.

3.2.2. Bounding $\|r_i(\cdot)\|_\infty$

Convergence is tested by inspecting the value of $\|r_i(\cdot)\|_\infty$. This raises the question whether we can compute $\|r_i(\cdot)\|_\infty$ using $\mathbf{r}_i$, or more generally, given $\mathbb{S}_N[p(\cdot)]$ can $\|p(\cdot)\|_\infty$ be computed for $p(\cdot) \in \mathbb{P}_N$? The answer is yes, and in a stable manner (Battles & Trefethen, 2004). However, this computation is rather expensive: $O(N^3)$ operations. Fortunately, we can bound $\|p(\cdot)\|_\infty$ using $\mathbb{S}_N[p(\cdot)]$ in only $O(N)$ operations, as the following proposition shows.

Proposition 2. *Let $p(\cdot) \in \mathbb{P}_N$. The following holds,*

$$\|p(\cdot)\|_\infty \leq (1+\frac{2}{\pi}\log N)\|\mathbb{S}_N[p(\cdot)]\|_\infty\,.$$

Thus, we use the following termination test:

$$\|\mathbf{r}_i\|_\infty < \frac{(1-\alpha)d_i\epsilon}{(1-\exp((\alpha-1)\gamma))(1+\frac{2}{\pi}\log N)} \tag{6}$$

(if $\alpha = 1$ then substitute $1-\exp((\alpha-1)\gamma)/(1-\alpha)$ with γ) for all $i = 1,\ldots,n$.

3.2.3. Solving the ODE

The solution to the ODE (5) is generally not a polynomial, and step 2 (b) cannot be implemented exactly if we restrict the solution space to $\mathbb{P}_N$. However, as N grows, better and better approximation to the solution can be found in $\mathbb{P}_N$. How to set N to be large enough so to find sufficiently accurate solution to the ODE is the subject of the next subsection. In this subsection, we explain how our algorithm finds an approximate solution to (5) within $\mathbb{P}_N$.

Writing $y(t) = y_i(t) + d(t)$ and rearranging (5), we can rewrite the ODE and the update step as

$$d'(t) + d(t) - r_i(t) = 0,\ \ d(0) = 0 \tag{7}$$

$$y_i(\cdot) \leftarrow y_i(\cdot) + d(\cdot)\,.$$

If we abandon solving (7) exactly so that we can require $d(\cdot) \in \mathbb{P}_N$, the update step (in the values-on-grid-points representation) then becomes

$$\mathbf{y}_i \leftarrow \mathbf{y}_i + \mathbf{d}$$

where $\mathbf{d} \equiv \mathbb{S}_N[d(\cdot)]$. After this step, the residual can be updated as well

$$\mathbf{r}_i \leftarrow \mathbf{r}_i - (\mathbf{\Xi} + \mathbf{I}_{N+1})\mathbf{d}\,.$$

The convergence criteria requires $\|\mathbf{r}_i\|_\infty$ to be small enough for all i, so it seems natural to attempt to find a $\mathbf{d}_i$ that will minimize $\|\mathbf{r}_i\|_\infty$ after the update. However, infinity norm minimization is rather expensive. Noting that $\|\mathbf{r}_i\|_\infty \leq \|\mathbf{r}_i\|_2$ we instead opt to minimize $\|\mathbf{r}_i\|_2$ after the update, which yields the following equality-constrained linear least-squares problem

$$\min_{\mathbf{d}} \|\mathbf{r}_i - (\mathbf{\Xi} + \mathbf{I}_{N+1})\mathbf{d}\|_2 \ \text{ s.t. } \ d_{N+1} = 0\,.$$

Writing

$$\mathbf{\Xi} + \mathbf{I}_{N+1} = \begin{pmatrix} \mathbf{\Xi}_1 & \mathbf{c} \end{pmatrix}$$

we get the following formula for $\mathbf{d}$:

$$\mathbf{d} = \begin{pmatrix} \mathbf{\Xi}_1^+\mathbf{r}_i \\ 0 \end{pmatrix}. \tag{8}$$

This leads to the following update formula for $\mathbf{r}_i$

$$\mathbf{r}_i \leftarrow (\mathbf{I}_{N+1} - \mathbf{\Xi}_1\mathbf{\Xi}_1^+)\mathbf{r}_i. \tag{9}$$

Let $\mathbf{R} \equiv \mathbf{I}_n - \mathbf{\Xi}_1\mathbf{\Xi}_1^+$. Since $\mathbf{\Xi}_1$ has full column rank, $\mathbf{R}$ is exactly a rank one matrix. In fact, $\mathbf{R} = \mathbf{u}\mathbf{u}^{\mathrm{T}}$ where $\mathbf{u}$ is the unique unit norm null vector of $\mathbf{\Xi}_1^{\mathrm{T}}$. This allows us to compute the update formula (9) in $O(N)$ time. Note that $\mathbf{\Xi}_1^+\mathbf{u} = 0_N$ and $\mathbf{R}\mathbf{u} = \mathbf{u}$, so setting $\mathbf{r}_i$ to 0 instead of using (9) will not change the result of any subsequent application of (8). However, it might affect the detection of convergence.

3.3. Choosing N

Recall that the update formula for the residual is $\mathbf{r}_i \leftarrow (\mathbf{I}_{N+1} - \mathbf{\Xi}_1\mathbf{\Xi}_1^+)\mathbf{r}_i$, and that to declare convergence $\|\mathbf{r}_i\|_\infty$

has to be small enough. It is crucial that N is set large enough so that $\|(\mathbf{I}_{N+1} - \mathbf{\Xi}_1\mathbf{\Xi}_1^+)\mathbf{r}_i\|_\infty$ is small enough for all $\mathbf{r}_i$s encountered by the algorithm. Observing that at the first iteration all residuals are multiple of the all-ones vector, we can bound the infinity error after the first iteration using the following lemma.

Lemma 3. *Let $\mathbf{1}_{N+1}$ be the $N+1$ dimensional all-ones vector. Provided that $\gamma \geq 1$ and $N \geq 10$ we have*

$$\|(\mathbf{I}_{N+1} - \mathbf{\Xi}_1\mathbf{\Xi}_1^+)\mathbf{1}_{N+1}\|_\infty \leq 20\exp(-\gamma/2)I_{N+1}(\gamma)(4/5)^{N+1}$$

where $I_0(\cdot), I_1(\cdot), \ldots$ are the modified Bessel functions of the first kind.

While we are unable to prove it rigorously, numerical experiments revealed that values of $\|(\mathbf{I}_{N+1} - \mathbf{\Xi}_1\mathbf{\Xi}_1^+)\mathbf{r}_i\|_\infty$ produced by the algorithm are of the same order of magnitude as that of the first iteration. Therefore, we base our criteria for choosing N on Lemma 3, and set N to be the minimum $N \geq 10$ that will guarantee

$$20\exp(-\gamma/2)I_{N+1}(\gamma)(4/5)^{N+1} \leq \frac{\epsilon}{\gamma(1+\frac{2}{\pi}\log N)}\,.$$

Our experience showed that this strategy for choosing N is extremely robust, never failing to allow the algorithm to converge in all our experiments (we stress that once the algorithm has detected convergence, the result is accurate enough regardless of how N is chosen).

An inequality due to Luke (1972) implies that $I_{N+1}(\gamma) \leq \gamma^{N+1}e^\gamma/2^{N+1}(N+1)!$ (we credit the answer to question 415834 of math stackexchange for this observation) which imply that $N = O(\gamma + \log(1/\epsilon))$.

3.4. Putting It All Together

The algorithm keeps a queue of indices which violate (6). The queue is initialized using the seed vector. In each iteration, the algorithm pops an index from the queue and operates on it. Suppose u is the popped index. The algorithm first updates $\mathbf{y}_u$ and $\mathbf{r}_u$ using formulas in subsection 3.2.3. It then updates $\mathbf{r}_v$ for all neighboring vs of u. If $\mathbf{r}_v$ violates (6), and it is not already in the queue, it is added to the queue. The algorithm terminates when the queue is empty.

During the execution of the algorithm, most of the $\mathbf{y}_i$s and $\mathbf{r}_i$s are zero. Our algorithm does not explicitly retain them in memory, and initializes $\mathbf{y}_i$ and $\mathbf{r}_i$ only when encountered first (that happens in three cases: as a seed vector, as a neighbor of a seed vector and as a neighbor of an index that was popped from the queue). We use an hash table for compact storage with fast access.

A detailed pseudo-code appears in the supplementary material. The only access our algorithm needs to $G = (V, E)$ is to query a vertex degree (i.e., given $v \in V$, return its degree $G.deg(v)$) and to query the set of vertex neighbors (i.e., given $v \in V$, return its neighbors $G.neighbors(v)$).

Once the algorithm terminates, the result $\mathbf{y}(\cdot)$ appear as a collection of $N+1$ dimensional vectors, which hold the values $\mathbf{y}(\cdot)$ at $N+1$ time points in the range $[0, \gamma]$. These vectors can be used for ranking. The value of $\mathbf{y}(t)$ for other values of $t \in [0, \gamma]$ can be computed efficiently using the barycentric formula (Berrut & Trefethen, 2004).

In terms of computational cost, we first note that while finding N and $\mathbf{\Xi}_1^+$ is expensive, this is a one-time operation given γ and ϵ, so their values can be cached for future uses. Thus, we do not include them in the analysis. Assuming hash operations take $O(1)$, the cost of a single iteration of the main loop is $O(N^2) = O((t + \log(1/\epsilon))^2)$. While we do not currently have a bound on the number of iterations, our experience shows that it is rather small.

4. Experimental Results

We experimentally evaluated our algorithm in the context of community detection using a seed vector (that is, we use the common sweep procedure after applying the diffusion, selecting the prefix with minimum conductance). It is not the purpose of this section to show that our algorithm always, or in most of the cases, produces better communities when compared to the baselines. Evaluating communities is very tricky, and it is unclear how to quantify which of two communities is better. For example, just comparing the conductance is questionable, since PageRank based sweeps tend to produce lower conductance but much larger (and less realistic) clusters than heat kernel based sweeps (Kloster & Gleich, 2014). Other metrics exist, and it is unclear which is best (Yang & Leskovec, 2015). It is not unreasonable to expect that the best metric is application dependent and might in fact be an ensemble of metrics, and thus for a given application the best strategy might be to work with an ensemble of diffusion vectors from which a collection of candidate clusters are generated.

Instead, we demonstrate that our algorithm produces diffusion vectors that are useful in the sense that they produces rankings that are distinct and competitive with the ones produced by high quality implementations of personalized PageRank and localized heat kernel. In competitive we mean that selecting the better community between the two is not a trivial task of selecting the smallest conductance, and in fact that task might be application dependent. As such, we aim to establish that our algorithm is a useful addition to the toolset of local graph diffusions.

We use two other codes as a baseline. The first is **hkgrow** (Kloster & Gleich, 2014)[3]. The second is **pprgrow,**

[3]The paper uses the name **hk-relax**, but the code uses **hkgrow**.

Table 1. Datasets

Graph	$\|V\|$	$\|E\|$
email-Enron	36,692	183,831
ca-AstroPh	18,771	198,050
soc-sign-epinions	131,580	711,210
soc-LiveJournal1	4,846,609	42,851,237
com-amazon	334,863	925,872
com-dblp	317,080	1,049,866
com-youtube	1,134,890	2,987,624
com-lj	3,997,962	34,681,189
com-orkut	3,072,441	117,185,083

an implementation of the personalized PageRank push algorithm (Andersen et al., 2006) by the authors of **hkgrow**. Similary, we call our algorithm **tpprgrow**. Note that we are comparing *algorithms* and not *diffusions*. The reason is that even when the underlying diffusion is the same (for example time-dependent PageRank with $\alpha = 1.0$ is identical to heat kernel), different algorithms often produce different approximate diffusion vector. Unless the accuracy parameter ϵ is set to be extremely small and well below common values, these different values result in different ranking between nodes. Indeed, it is common practice to use ϵ as another parameter, and to run the algorithm with different ϵ values and select the best cluster found, instead of driving ϵ to be low enough for the ranking to be fully determined (we remark that setting ϵ so that the exact ranking between all nodes is fully determined necessitates a dense vector, which is counter to the goal of having a local algorithm).

4.1. Conductance, Cluster Size and Runtime

In the first set of experiments we compare **hkgrow** with $\gamma = 5.0$ to **tpprgrow** with $\alpha = 1.0, \gamma = 5.0$ and **tpprgrow** with $\alpha = 0.85, \gamma = 5.0$. Note that the underlying diffusion for **tpprgrow** with $\alpha = 1.0, \gamma = 5.0$ and **hkgrow** with $\gamma = 5.0$ is exactly the same. So any difference in the observed results is due to finding a different approximate diffusion vector within the allowed error budget. We use the four widely used datasets at the top of Table 1. For each dataset, we randomly choose a seed node, and detect a community around it using the various algorithms. We repeat 1000 times for every dataset. We use $\epsilon = 0.001$.

The results are summarized in Figure 2, Table 2 and Table 3. Figure 2 shows scatter plots of the conductance versus cluster size of the cluster found by the two variants of **tpprgrow** using randomly selected seed nodes. The values are normalized according to the cluster found using **hkgrow**. We see that **tpprgrow** tends to produce clusters with slightly higher (but comparable) conductance, but often of smaller, and perhaps more realistic, size. It also occasionally produces clusters with lower conductance.

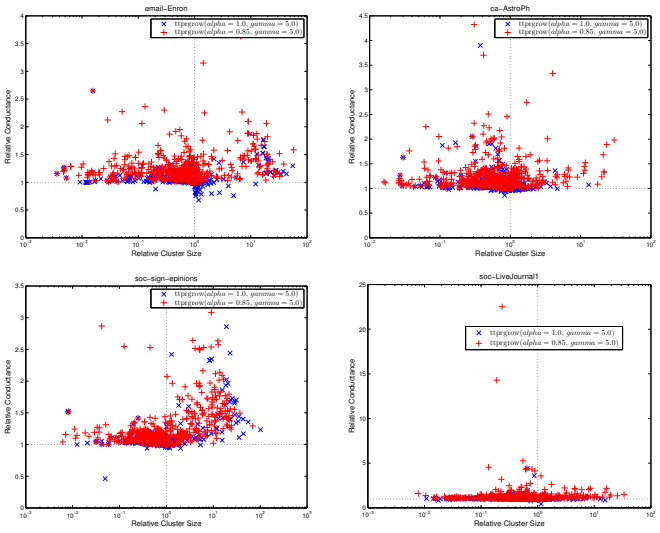

Figure 2. Scatter plot of conductance vs. community size of **tpprgrow** relative to **hkgrow**. Each point represent a single run from a random seed, with blue x's with $\alpha = 1.0$ and $\gamma = 5.0$, and red +'s for $\alpha = 0.85$ and $\gamma = 5.0$. The conductance and community size is relative to the values obtained by the cluster found by the **hkgrow** with the same seed node.

Table 2 quantifies this further. In the table we compare the size and conductance of the cluster found between the algorithms. For each combination of algorithms, we find what percent of the runs resulted in clusters that are smaller, equal or larger of one algorithm versus the other. We repeat with regards to conductance. We see that not only does **tpprgrow** with $\alpha = 1.0, \gamma = 5.0$ tends to produce smaller sized clusters, but also with higher conductance. **tpprgrow** with $\alpha = 0.85, \gamma = 5.0$ tends to produce even smaller clusters and even higher conductance.

Table 3 compares the performance in terms of running time. The total time columns aggregates the total running time of all 1000 experiments. While **tpprgrow** tends to be somewhat slower than **hkgrow** when used with $\alpha = 1.0$ and $\gamma = 5.0$, it is faster than **hkgrow** when $\alpha = 0.85$ is used. This is probably due to fact that with $\alpha = 0.85$ the sum of diffusion coefficients converge to 1.0 faster (recall Figure 1). We also report the average number of iterations in the main loop. This corresponds to the average number of access to the edge list of a node in the graph. In some cases, this might be dominant cost (e.g., if the graph is too big to kept in memory and instead kept on disk, so each access to the neighbor list of a vertex is access to external storage). We see that **tpprgrow** tends to do less iterations.

The results make it clear that the clusters that can be found using **tpprgrow** are often distinct and competitive with the ones found by **hkgrow**.

Table 2. Comparison of cluster profile produced using different diffusions. **tpprgrow-1** refers to **tpprgrow** with $\alpha = 1.0, \gamma = 5.0$. **tpprgrow**-2 uses $\alpha = 0.85, \gamma = 5.0$. For **hkgrow** we use $\gamma = 5.0$. A triple "X%/Y%/Z%" means that between the algorithms compared, the first algorithm had a value smaller than the second algorithm's value for X% of the cases, equal value for Y% of the cases, and lager value for Z% of the cases. For example, the "36%/49%/15%" under "Size" for "**tpprgrow**-1 vs. **hkgrow**" means that in 36% of the cases **tpprgow** produced a cluster which was smaller than the one produced by **hkgrow**, 49% of the cases the cluster size was the same and in 15% of the cases the cluster was larger.

Dataset	**tpprgrow**-1 vs. **hkgrow**		**tpprgrow**-2 vs. **hkgrow**		**tpprgrow**-2 vs. **tpprgrow**-1	
	Size	Conductance	Size	Conductance	Size	Conductance
Enron	36%/49%/15%	9%/48%/43%	52%/29%/19%	1%/29%/70%	49%/32%/19%	1%/31%/68%
ca-AstroPh	40%/43%/17%	10%/42%/48%	59%/23%/18%	0%/22%/78%	58%/23%/19%	1%/22%/77%
soc-sign-epinions	27%/60%/12%	5%/ 60% / 35%	38%/38%/23%	0%/39%/61%	39%/ 41%/20%	1%/41%/58%
soc-LiveJournal1	42%/42%/16%	12%/41%/47%	61%/21%/18%	0%20%/80%	58%/ 22%/20%	1%/21%/78%

Table 3. Comparison between the total time and average number of iterations of the main loop for finding cluster around the same seeds between **hkgrow** and two **tpprgrow** configurations.

Dataset	**tpprgrow** with $\alpha = 1.0, \gamma = 5.0$		**tpprgrow** with $\alpha = 0.85, \gamma = 5.0$		**hkgrow** with $\gamma = 5.0$	
	Avg. #Its	Total Time (sec)	Avg. #Its	Total Time (sec)	Avg. #Its	Total Time (sec)
Enron	358.3	2.26	163.8	0.70	594.0	0.93
ca-AstroPh	124.4	1.47	69.8	0.58	272.7	0.93
soc-sign-epinions	111.9	1.16	64.4	0.56	259.3	0.80
soc-LiveJournal1	134.1	1.86	72.7	0.51	304.8	1.00

Table 4. Comparison between **tpprgrow**, **hkgrow** and **pprgrow** on finding real-world communities (datasets with ground-truth communities). In the "Statistics" column, the first item is the average F_1-score, the second item is the average conductance, and the third is the average size.

Dataset	**tpprgrow**		**hkgrow**		**pprgrow**	
	Statistics	Time (sec)	Statistics	Time (sec)	Statistics	Time (sec)
amazon	0.940/0.026/28	131	**0.943**/0.027/**28**	**26**	0.822/**0.019**/46	362
dblp	0.591/0.165/**33**	343	**0.592**/0.166/34	**138**	0.527/**0.139**/53	429
youtube	**0.185**/0.527/**61**	292	0.173/**0.490**/84	126	0.183/0.522/89	**108**
lj	**0.730**/0.170/**45**	505	0.719/0.153/51	**197**	0.655/**0.143**/63	555
orkut	0.357/0.778/**58**	6058	**0.363**/0.762/65	2814	0.360/**0.755**/61	**1673**

4.2. Datasets with Ground Truth

Next, we evaluate the different algorithms using datasets for which ground truth communities exists (Yang & Leskovec, 2015). These are the datasets at the bottom of Table 1. Our setup is quite similar to the one used by Kloster & Gleich (2014): for each dataset, we use only the list of 5000 top communities, and experiment on all communities with 10 or more members. Given a seed, we run **tpprgrow** with $\epsilon = 0.001$ and three different values of α: $0.85, 0.99, 1.0$. We use $\gamma = 5.0$, but examine the diffusion vector at three additional time points. Among the clusters found, we pick the one with minimum conductance. We repeat this for every member of the ground truth community, using it as a seed, choosing the community with maximum F_1-score. Similarly we run on **hkgrow** with the same four values of γ, and **pprgrow** with $\alpha = 0.85, 0.99$ ($\alpha = 1.0$ does not make sense for **pprgrow**).

Table 4 reports for each dataset and algorithm, the average F_1-score, average cluster size, average cluster conductance, and total running time. On average, **tpprgrow** and **hkgrow** find clusters of about the same quality. **hkgrow** is faster than our algorithm by a factor of x2-5 (however, our algorithm explores a much wider parameter space). **pprgrow** tends to produce larger clusters, with lower conductance, but they are less faithful to the ground truth. It also tends to be slower than **hkgrow** and **tpprgrow**.

5. Conclusions

We have described an efficient local algorithm for yet-another graph diffusion. Our algorithm is also yet-another local algorithm for PageRank and heat kernel. Our experiments suggest that the algorithm produces rankings that are distinct and competitive with the ones produced by high quality implementations of personalized PageRank and local heat kernel, and so is a useful addition to the toolset of local graph diffusions.

From a theoretical perspective, by selecting parameters so to behave like PageRank or heat kernel, the time-dependent PageRank diffusion can recover the same rigorous bounds that have been proven for those graph diffusions. However, it is unclear whether a careful choice of parameter can, in fact, produce stronger theoretical results. We propose that as an open question for future research.

Acknowledgments

We thank David Gleich and Kyle Kloster for useful discussions. Haim Avron acknowledges the support from the XDATA program of the Defense Advanced Research Projects Agency (DARPA), administered through Air Force Research Laboratory contract FA8750-12-C-0323.

References

Abramowitz, M. and Stegun, I. A. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, New York, Ninth Dover printing, Tenth GPO printing edition, 1964.

Andersen, R., Chung, F., and Lang, K. Local graph partitioning using pagerank vectors. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, pp. 475–486, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2720-5. doi: 10.1109/FOCS.2006.44. URL http://dx.doi.org/10.1109/FOCS.2006.44.

Battles, Z. and Trefethen, L. An extension of MATLAB to continuous functions and operators. *SIAM Journal on Scientific Computing*, 25(5):1743–1770, 2004. doi: 10.1137/S1064827503430126. URL http://dx.doi.org/10.1137/S1064827503430126.

Berrut, J. and Trefethen, L. Barycentric Lagrange interpolation. *SIAM Review*, 46(3):501–517, 2004. doi: 10.1137/S0036144502417715. URL http://dx.doi.org/10.1137/S0036144502417715.

Botchev, M. A., Grimm, V., and Hochbruck, M. Residual, restarting and Richardson iteration for the matrix exponential. *SIAM journal on scientific computing*, 35(3): A1376–A1397, 2013. URL http://doc.utwente.nl/87651/.

Boyd, J. P. *Chebyshev and Fourier Spectral Methods*. Springer-Verlag, New York, 1989. 792 pp.

Brin, S. and Page, L. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998. ISSN 0169-7552. doi: 10.1016/S0169-7552(98)00110-X. URL http://dx.doi.org/10.1016/S0169-7552(98)00110-X.

Chung, F. The heat kernel as the pagerank of a graph. *Proceedings of the National Academy of Sciences*, 104(50): 19735–19740, December 2007. ISSN 1091-6490. doi: 10.1073/pnas.0708838104. URL http://dx.doi.org/10.1073/pnas.0708838104.

Chung, F. A local graph partitioning algorithm using heat kernel pagerank. In Avrachenkov, Konstantin, Donato, Debora, and Litvak, Nelly (eds.), *Algorithms and Models for the Web-Graph*, volume 5427 of *Lecture Notes in Computer Science*, pp. 62–75. Springer Berlin Heidelberg, 2009. ISBN 978-3-540-95994-6. doi: 10.1007/978-3-540-95995-3_6. URL http://dx.doi.org/10.1007/978-3-540-95995-3_6.

Driscoll, T. A., Hale, N., and Trefethen, L. N. (eds.). *Chebfun Guide*. Pafnuty Publications, Oxford, UK, 1st edition, 2014.

Gleich, D. F. and Rossi, R. A. A dynamical system for PageRank with time-dependent teleportation. *Internet Mathematics*, 10(1–2):188–217, June 2014. doi: 10.1080/15427951.2013.814092.

Kloster, K. and Gleich, D. F. Heat kernel based community detection. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pp. 1386–1395, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2956-9. doi: 10.1145/2623330.2623706. URL http://doi.acm.org/10.1145/2623330.2623706.

Laforgia, A. Bounds for modified Bessel functions. *Journal of Computational and Applied Mathematics*, 34(3):263 – 267, 1991. ISSN 0377-0427. doi: http://dx.doi.org/10.1016/0377-0427(91)90087-Z. URL http://www.sciencedirect.com/science/article/pii/037704279190087Z.

Luke, Y. L. Inequalities for generalized hypergeometric functions. *Journal of Approximation Theory*, 5(1):41 – 65, 1972. ISSN 0021-9045. doi: http://dx.doi.org/10.1016/0021-9045(72)90028-7. URL http://www.sciencedirect.com/science/article/pii/0021904572900287.

Paris, R. An inequality for the Bessel function $J_\nu(\nu x)$. *SIAM Journal on Mathematical Analysis*, 15(1):203–205, 1984. doi: 10.1137/0515016. URL http://dx.doi.org/10.1137/0515016.

Trefethen, L. N. *Spectral Methods in MATLAB*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000. ISBN 0-89871-465-6.

Yang, J. and Leskovec, J. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015. ISSN 0219-1377. doi: 10.1007/s10115-013-0693-z. URL http://dx.doi.org/10.1007/s10115-013-0693-z.