EFFICIENT ALGORITHMS FOR PERSONALIZED PAGERANK

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Peter Lofgren
December 2015

Abstract

We present new, more efficient algorithms for estimating random walk scores such as Personalized PageRank from a given source node to one or several target nodes. These scores are useful for personalized search and recommendations on networks including social networks, user-item networks, and the web. Past work has proposed using Monte Carlo or using linear algebra to estimate scores from a single source to every target, making them inefficient for a single pair. Our contribution is a new bidirectional algorithm which combines linear algebra and Monte Carlo to achieve significant speed improvements. On a diverse set of six graphs, our algorithm is 70x faster than past state-of-the-art algorithms. We also present theoretical analysis: while past algorithms require $\Omega(n)$ time to estimate a random walk score of typical size $\frac{1}{n}$ on an n-node graph to a given constant accuracy, our algorithm requires only $O(\sqrt{m})$ expected time for an average target, where m is the number of edges, and is provably accurate.

In addition to our core bidirectional estimator for personalized PageRank, we present an alternative algorithm for undirected graphs, a generalization to arbitrary walk lengths and Markov Chains, an algorithm for personalized search ranking, and an algorithm for sampling random paths from a given source to a given set of targets. We expect our bidirectional methods can be extended in other ways and will be useful subroutines in other graph analysis problems.

Acknowledgements

I would like to acknowledge the help of the many people who helped me reach this place. First I thank my PhD advisor Ashish Goel for his help, both in technical advice and in mentoring. Ashish recognized that PPR was an important problem where progress could be made, helped me get started by suggesting sub-problems, and gave excellent advice at every step. I'm always amazed by the breadth of his understanding, from theoretical Chernoff bounds to systems-level caching to the business applications of recommender systems. He also helped me find excellent internships, was very gracious when I took a quarter off to try working at a start-up, and generously hosted parties at his house for his students.

I also thank my other reading committee members, Hector and Jure. Hector gave great advice on two crowd-sourcing papers (independent of my thesis) and is remarkably nice, making the InfoLab a positive community and hosting frequent movie nights at his house. Jure taught me much when I CAed his data mining and social networking courses and has amazing energy and enthusiasm.

I thank postdoc (now Cornell professor) Sid Banerjee for amazing collaboration in the last two years. Sid brought my research to a completely new level, and everything in this thesis was co-developed with Sid and Ashish. I thank my other collaborators during grad school for helping me learn to do research and write papers: Pankaj Gupta, C. Seshadhri, Vasilis Verroios, Qi He, Jaewon Yang, Mukund Sundararajan, and Steven Whang. I thank Dominic Hughes for his excellent advice and mentorship when I interned at his recommendation algorithms start-up and in the years after.

Going back further, I thank my undergraduate professors and mentors, including Nick Hopper, Gopalan Nadathur, Toni Bluher, Jon Weissman, and Paul Garrett.

Going back even further, I thank all the teachers who encouraged me in classes and math or computer science competitions, including Fred Almer, Brenda Kellen, and Brenda Leier.

I thank my family for raising me to value learning and for always loving me unconditionally. I thank all my friends (you know who you are) who made grad school fun through hiking, camping, playing board games, acting, dancing, discussing politics, discussing grad school life, and everything else. I look forward to many more years enjoying life with you.

Finally I thank everyone I forgot to list here who also gave me their time, advice, help, or kindness. It takes a village to raise a PhD student.

Contents

Abstract iv

Acknowledgements v

1 Introduction 1

1.1 Motivation from Personalized Search . . . 1
1.2 Other Applications . . . 3
1.3 Preliminaries . . . 3
1.4 Defining Personalized PageRank . . . 5
1.5 Other Random Walk Scores . . . 7
1.6 Problem Statements . . . 9
1.7 High Level Algorithm . . . 11
1.8 Contributions . . . 12

2 Background and Prior Work 16

2.1 Other Approaches to Personalization . . . 16
2.2 Power Iteration . . . 17
2.3 Reverse Push . . . 17
2.4 Forward Push . . . 20
2.5 Monte Carlo . . . 21
2.6 Algorithms using Precomputation and Hubs . . . 22

3 Bidirectional PPR Algorithms 24

3.1 FAST-PPR . . . 24

3.2 Bidirectional-PPR . . . 26
3.2.1 The `BidirectionalPPR` Algorithm . . . 26
3.2.2 Accuracy Analysis . . . 29
3.2.3 Running Time Analysis . . . 32
3.3 Undirected-Bidirectional-PPR . . . 35
3.3.1 Overview . . . 35
3.3.2 A Symmetry for PPR in Undirected Graphs . . . 36
3.3.3 The `UndirectedBiPPR` Algorithm . . . 37
3.3.4 Analyzing the Performance of `UndirectedBiPPR` . . . 39
3.4 Experiments . . . 42
3.4.1 PPR Estimation Running Time . . . 42
3.4.2 PPR Estimation Running Time on Undirected Graphs . . . 43

4 Random Walk Probability Estimation **47**
4.1 Overview . . . 47
4.2 Problem Description . . . 48
4.2.1 Our Results . . . 49
4.2.2 Existing Approaches for MSTP-Estimation . . . 50
4.3 The Bidirectional MSTP-estimation Algorithm . . . 52
4.3.1 Algorithm . . . 52
4.3.2 Some Intuition Behind our Approach . . . 54
4.3.3 Performance Analysis . . . 55
4.4 Applications of MSTP estimation . . . 59
4.5 Experiments . . . 60

5 Precomputation for PPR Estimation **64**
5.1 Basic Precomputation Algorithm . . . 64
5.2 Combining Random Walks to Decrease Storage . . . 66
5.3 Storage Analysis on Twitter-2010 . . . 67
5.4 Varying Minimum Probability to Reduce Storage . . . 72

6 Personalized PageRank Search **73**
6.1 Personalized PageRank Search . . . 73
6.1.1 Bidirectional-PPR with Grouping . . . 76
6.1.2 Sampling from Targets Matching a Query . . . 77
6.1.3 Experiments . . . 84
6.2 Sampling Paths Conditioned on Endpoint . . . 87
6.2.1 Introduction . . . 87
6.2.2 Path Sampling Algorithm . . . 89

7 Conclusion and Future Directions **96**
7.1 Recap of Thesis . . . 96
7.2 Open Problems in PPR Estimation . . . 98
7.2.1 The Running-Time Dependence on $\bar{d}$ . . . 98
7.2.2 Decreasing Pre-computation Storage or Running Time . . . 98
7.2.3 Maintaining Pre-computed Vectors on Dynamic graphs . . . 98
7.2.4 Parameterized Worst-Case Analysis for Global PageRank . . . 99
7.2.5 Computing Effective Resistance or Hitting Times . . . 99
7.3 Open Problems in Personalized Search . . . 100
7.3.1 Decreasing Storage for Text Search . . . 100
7.3.2 Handling Multi-Word Queries . . . 100
7.3.3 User Study of Personalized Search Algorithms for Social Networks 100

Bibliography **102**

List of Tables

4.1 Datasets used in experiments 61

List of Figures

1.1 A comparison of PPR with a production search algorithm on Twitter in 2014. . . . 2
1.2 Academic Search Example . . . 4
1.3 Example Graph . . . 7
1.4 PPR Values on Example Graph . . . 8

2.1 `ReversePush` Example . . . 19

3.1 FAST-PPR Algorithm . . . 25
3.2 An example graph. We will estimate $\pi_s[t]$. . . . 28
3.3 An example graph. We will estimate $\pi_s[t]$. . . . 28
3.4 The empirical probability of stopping at each node after 77 walks from s for some choice of random seed. Note there is an implicit self loop at v. . . . 29
3.5 Running-time Plots . . . 46

4.1 Visualizing a sequence of REVERSE-PUSH-MSTP operations: Given the Markov chain on the left with S as the target, we perform REVERSE-PUSH-MSTP operations $(S, 0), (H, 1), (C, 1), (S, 2)$. . . . 53
4.2 Estimating heat kernels: Bidirectional MSTP-estimation vs. Monte Carlo, Forward Push. For a fair comparison of running time, we choose parameters such that the mean relative error of all three algorithms is around 10%. Notice that our algorithm is 100 times faster per (source, target) pair than state-of-the-art algorithms. . . . 63

5.1 Mean reverse residual size (number of non-zeros) as $r^{\mathrm{r}}_{\max}$ varies, on Twitter-2010. The dotted line is $y = \frac{0.5}{r^{\mathrm{r}}_{\max}}$. 70
5.2 Mean forward residual size (number of non-zeros) as $r^{\mathrm{f}}_{\max}$ varies on Twitter-2010. The dotted line is $y = \frac{10}{r^{\mathrm{f}}_{\max}}$. 71

6.1 Example 79
6.2 Comparing PPR-Search Algorithms 85
6.3 Pokec Running Time 87

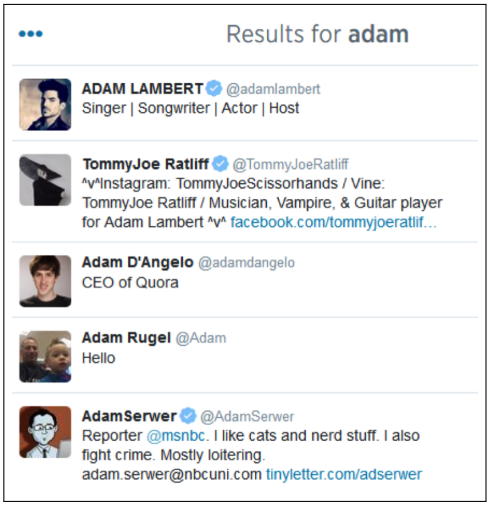
Chapter 1

Introduction

1.1 Motivation from Personalized Search

As the amount of information available to us grows exponentially, we need better ways of finding the information that is relevant to us. For general information queries like "What is the population of India?" the same results are relevant to any user. However, for context-specific queries like "Help me connect with the person named Adam I met at the party yesterday," or "Where is a good place to eat lunch?" giving different results to different users (personalization) is essential. There are many approaches to personalization, including using the user's search history, their attributes like location, or their social network, which we discuss in Section 2.1. In this thesis we focus on one approach to personalization, using random walks as described in Sections 1.4 and 1.5, which has proven useful in a variety of applications listed in Section 1.2. We present new, dramatically more efficient algorithms for computing random walk scores, and for concreteness we focus on computing the most well-known random walk score, Personalized PageRank.

Personalized PageRank (PPR) [45] is the personalized version of the PageRank algorithm which was important to Google's initial success. On any graph, given a starting node s whose point of view we take, Personalized PageRank assigns a score to every node t of the graph. This score models how much the user s is in interested in t, or how much s trusts t. More generally we can personalize to a



(a) Ranked results on twitter.com in 2014 for query "Adam" from a logged-in user who followed people in technology and politics. Notice that results appear un-personalized.

Name	Description
Adam Messinger	CTO twitter
Adam D'Angelo	CEO of Quora
Adam Satariano	Technology Reporter, Bloomberg News
Adam Steltzner	Rocket scientist, intermittent gardener, master of mars, and dangerous dinner guest. Co-founder of Adam and Trisha's dog and baby farm.
Adam Rugel	Hello

(b) Results for the same query, re-ranked by Personalized PageRank from the searching user's account. Notice that the top results are now in technology, reflecting the user's personalized interests.

Figure 1.1: A comparison of PPR with a production search algorithm on Twitter in 2014.

distribution over starting nodes, for example in web search we can create a distribution with equal probability mass on the web pages the searching user has bookmarked. If we personalize to the uniform distribution over all nodes, the score is no longer personalized, and we recover the standard (global) PageRank score.

As an example of how PPR can personalize search results, in Figure 1.1(a) we show a search for "Adam" done on Twitter in 2014 by the author, who was signed in to an account which followed people in technology and politics. However, the top result is a singer-songwriter, and the second result is a guitar player–results with much global popularity, but which are not personalized based on who the searcher follows. If we re-rank these results using Personalized PageRank from the same account, we get the results in Figure 1.1(b). Now the top two results are both in technology, and the results are more relevant overall.

As an example of how changing the source s of the PPR algorithm results in different rankings, we consider personalized search on a citation graph. On a citation graph provided by Cite-seer, we created a demo which given a keyword and

researcher name, finds all papers containing that keyword, and ranks them from the point of view of the given researcher. For a given researcher, we find all papers by that researcher, and define a source distribution s giving equal weight to those papers. We then use Personalized PageRank from s on the citation graph to rank papers matching the given keyword. As an example, the keyword "entropy" means different things to different researchers, so we compare the top results for keyword "entropy" from different points of view. In Figure 1.2 we show how the search results change when personalized for different researchers: a networking researcher (McKeown), a database/data-mining researcher (Garcia-Molina), and a learning-theory researcher (Valiant). These different rankings demonstrate that by changing the source distribution, we can use Personalized PageRank to understand the importance of nodes in a graph from different points of view.

1.2 Other Applications

The importance of Personalized PageRank extends beyond search on social networks. PPR has found use in many other domains, including friend recommendation on Facebook [6], who to follow on Twitter [25], graph partitioning [2], community detection [57], and other applications [54]. On the web graph there is significant work on using Personalized PageRank to rank web pages (e.g. [28, 27]). PPR is also useful on heterogeneous graphs; for example it can be used to rank items in a bipartite user-item graph, in which there is an edge from a user to an item if the user has liked that item. Random walks on a user-item graph have proven useful for YouTube when recommending videos [8]. Gleich's survey [22] lists a variety of applications of PageRank beyond the web, in fields from biology to chemistry to civil engineering.

1.3 Preliminaries

We work on a directed graph $G = (V, E)$ with $n = |V|$ nodes and $m = |E|$ edges. Our algorithms all generalize naturally to weighted graphs with weight matrix W having entry $w_{u,v} \geq 0$ for each edge $(u, v) \in E$, although we sometimes present algorithms

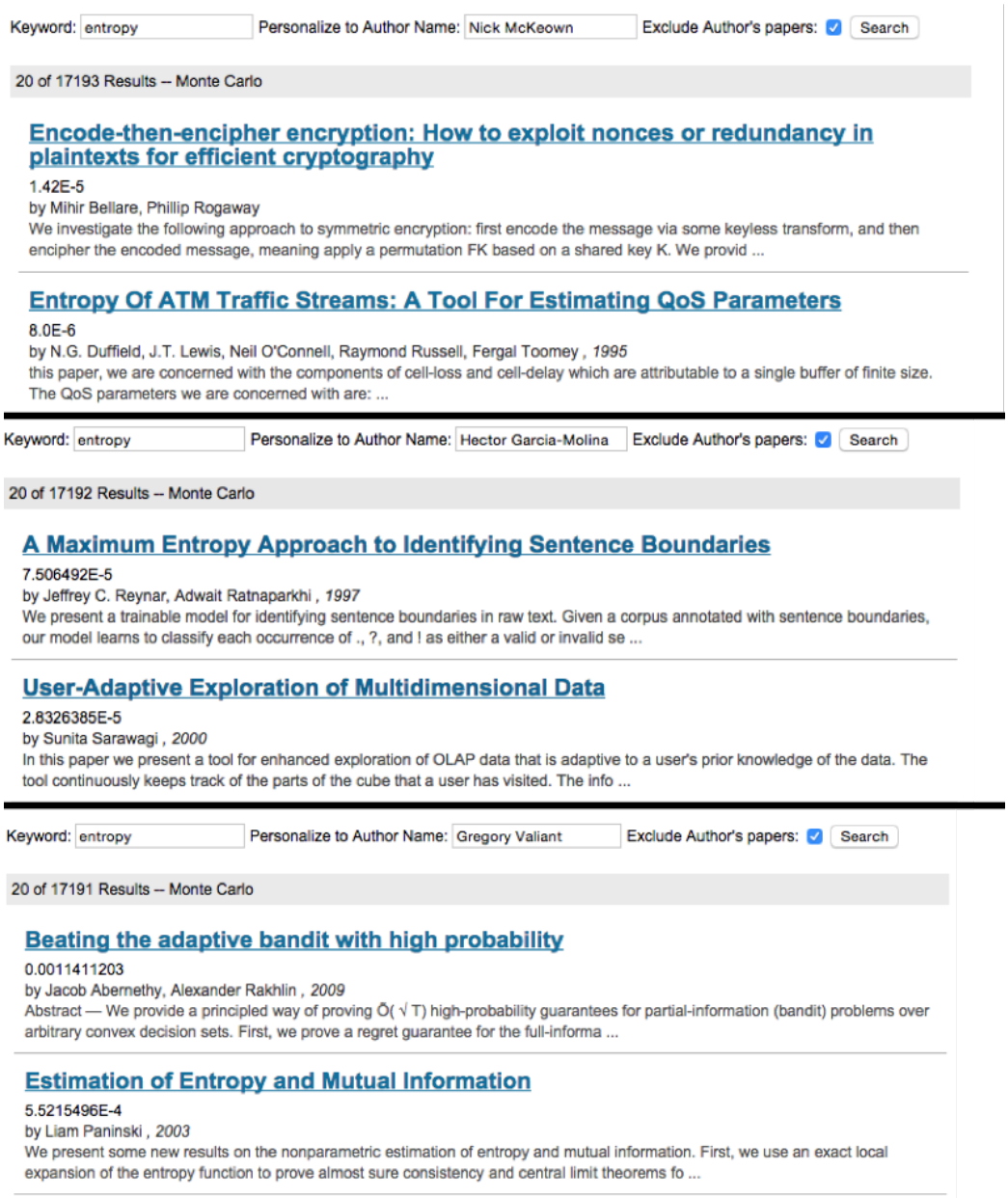


Figure 1.2: The top two computer science papers containing the multi-faceted keyword "entropy" personalized to three different researchers' point of view: a networking researcher, a database/data-mining researcher, and a learning-theory researcher. Notice how the ranking is very different for different researchers, and PPR on the citation graph captures the interests of each researcher.

and proofs for the unweighted case to simplify the exposition. For simplicity we assume the weights are normalized such that for all u, $\sum_v w_{u,v} = 1$.

Define the out-neighbors of a node u by $\mathcal{N}^{out}(u) = \{v : (u, v) \in E\}$ and let $d^{out}(u) = |\mathcal{N}^{out}(u)|$; define $\mathcal{N}^{in}(u)$ and $d^{in}(u)$ similarly. For example, in the graph in Figure 1.3, we have $\mathcal{N}^{out}(c) = \{v, x, w\}$ and $\mathcal{N}^{in}(c) = \{s, x, w\}$. Define the average degree of nodes $\bar{d} = \frac{m}{n}$.

For a vector $x \in \mathbb{R}^n$, we denote the ith entry as $x[i] \in \mathbb{R}$.

As alluded to before, we denote the starting node or distribution by s. We overload notation to denote either a single node $s \in V$ or a distribution $s \in \mathbb{R}^n$ assigning weight $s[v] \geq 0$ to node $v \in V$. We can interpret a node $s \in V$ as a distribution giving probability mass 1 to s and 0 to other nodes. Our algorithms generalize naturally to a start distribution, but for exposition we usually assume s is a single node in V.

1.4 Defining Personalized PageRank

Personalized PageRank has two equivalent definitions. We define both here because they are both useful way of thinking about PPR. In the first definition, Personalized PageRank recursively models the importance of nodes on a directed graph. At a high level, given a start node s whose point of view we take, we say that s is important, and in addition we say a node is important if its in-neighbors are important. To keep the importance scores bounded, we normalize the importance given from a node u to a node v though an edge (u, v) by dividing by u's out-degree. In addition, we choose a decay term α, and transfer a fraction $1 - \alpha$ of each node u's importance to u's out-neighbors. Formally, given normalized edge weight matrix W (as defined earlier, entry $w_{u,v}$ is the weight of edge (u, v) and $\forall u \sum_v w_{u,v} = 1$), the Personalized PageRank vector π_s with respect to source node s is the solution to the recursive equation

$$\pi_s = \alpha e_s + (1 - \alpha)\pi_s W. \tag{1.1}$$

If s is a distribution (viewed as a column vector where entry $s[i]$ is the weight s gives to node i) the recursive equation becomes

$$\pi_s = \alpha s + (1-\alpha)\pi_s W.$$

An equivalent definition is in terms of the terminal node of a random walk starting from s. Let $(X_0, X_1, \ldots, X_L)$ be a random walk starting from $X_0 = s$ of length $L \sim Geometric(\alpha)$. Here by $L \sim Geometric(\alpha)$ we mean $\Pr[L = \ell] = (1-\alpha)^\ell \alpha$. This walk starts at s and does the following at each step: with probability α, terminate; and with the remaining probability $1-\alpha$, continue to a random out-neighbor of the current node. Here if the current node is u, the random neighbor $v \in \mathcal{N}^{out}(u)$ is chosen with probability $w_{u,v}$ if the graph is weighted or with uniform probability $1/d^{out}(u)$ if the graph is unweighted. Then the PPR of any node t is the probability that this walk stops at t:

$$\pi_s[t] = \mathbb{P}[X_L = t]. \tag{1.2}$$

Notice in this definition there is a single random walk[1].

The equivalence of these two definitions can be seen by solving Equation 1.1 for π_s and then using a power series expansion [5]:

$$\pi_s = (I - (\alpha e_s + (1-\alpha)W))^{-1} = \sum_{\ell=0}^{\infty} \alpha(1-\alpha)^\ell e_s W^\ell.$$

In this expression we see the probability a walk has length ℓ, $\alpha(1-\alpha)^\ell$, and the probability distribution after ℓ steps from s, $e_s W^\ell$. For understanding the algorithms in this thesis, the second definition ($\pi_s[t]$ is the probability of stopping at t on a single random walk from s) is the more useful one.

[1] Yet another definition of $\pi_s[t]$ to be aware of is the fraction of time spent at t in the stationary distribution of the following random process over nodes: at each step, with probability α we transition ("teleport") back to s, and with remaining probability we transition from the current node u to a random out-neighbor $v \in \mathcal{N}^{out}(u)$. Because this chain is ergodic there is a unique stationary distribution.

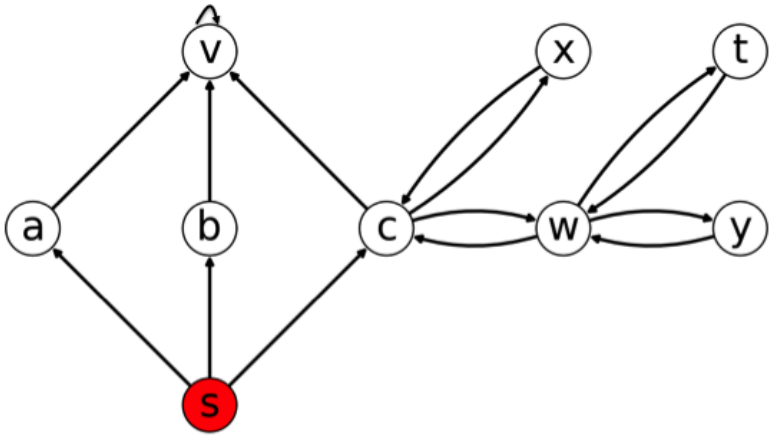


Figure 1.3: A graph with 9 nodes. We will compute PPR starting from s.

If some node u has no out-neighbors, different conventions are possible. For concreteness, we choose the convention of adding an artificial sink state to the graph, adding an edge from any such node u to the sink, and adding a self-loop from the sink to itself. In his survey [22], Gleich describes other conventions for dealing with such dangling nodes.

As an example of the scores personalized PageRank assigns, consider the graph in Figure 1.3. If we compute PPR values from s to all other nodes, we get the numbers shown in Figure 1.4

1.5 Other Random Walk Scores

Personalized PageRank uses a geometric distribution for random walk length, assigning weight $\alpha(1-\alpha)^\ell$ to walks of length ℓ for some parameter α, but there is nothing magical about that length distribution[2]. If we instead use a Poisson length distribution, giving weight $e^{-\alpha}\alpha^\ell/\ell!$ to paths of length ℓ, we get the the Heat Kernel which

[2]One unique property of a geometric random variable is that it is memoryless: if we have taken ℓ steps of a random walk with geometrically distributed length, then the expected number of remaining steps is the same as the initial number of steps. Expressed another way: if after each step of the walk we stop with probability α, we do not need to keep track of how long we've been walking, and can "forget" how many steps we've taken. This gives PPR the simpler recursive Equation 1.1.

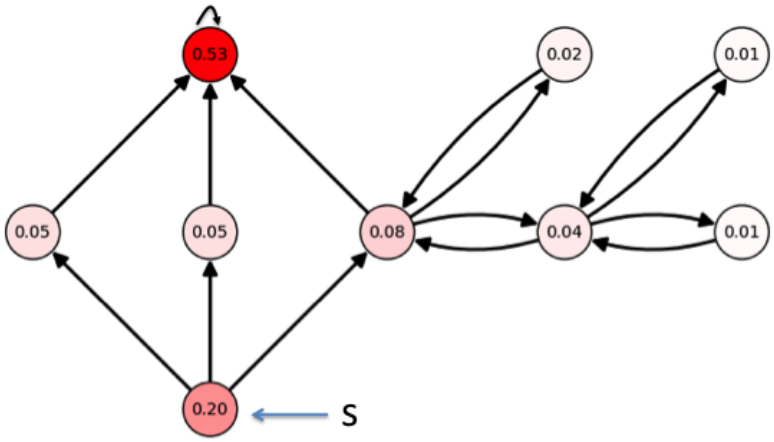


Figure 1.4: The PPR values from s to each other node in the graph of Figure 1.3. Nodes are also shaded by their PPR value. Notice that the top left node has the highest score. This is because all of s's out-neighbors point to t, and it matches our intuition that s should be interested in the top-left node, or trust it. On the other hand, the lowest score is for the two nodes on the right. This is because they are far from s and separated from s by several nodes with other neighbors, and it matches our intuition that s is likely less interested in these nodes.

is useful for community detection [31]. In general, we can choose any weights $(\alpha_\ell)_{l=0}^{\infty}$ such that $\sum_{\ell=0}^{\infty} \alpha_\ell = 1$ and assign weight ℓ to walks of length ℓ. If W is the random walk matrix, normalized so every row sums to 1, then there resulting score from s to t is

$$\sum_{\ell=0}^{\infty} \alpha_\ell s W^\ell e_t.$$

It is also not necessary to do a standard random walk to get a useful personalization score. In personalized SALSA [7] we alternate between taking forward and reverse walks along edges. For example, on Twitter if two users Alice and Bob both follow Barack Obama, and Bob follows an opposing political candidate like Jeb Bush, then an alternating random walk from Alice might go to Obama, then to Bob, then to Bush. On the other hand, a standard random walk would go from Alice to Obama to someone Obama follows (which might not include politicians of the opposite party). For an alternating random walk, even length walks tend to find similar users to the source user, since if two users both follow the same people they are likely similar.

Odd length random walks tend to find users the source might want to follow (since they are followed by users similar the to source user). See the section on SALSA in [25] for a fuller discussion of the meaning of SALSA. In an experiment at Twitter on real users, SALSA actually worked better than PPR at generating who-to-follow candidates which the user subsequently followed [25].

Our PPR estimation algorithms can also compute SALSA because SALSA on a graph G is actually equivalent to PPR on a transformed graph G'. To transform a given directed graph, for each node u in the original graph G, we create two nodes, a "consumer-node" u' and a "producer-node" u'' in G'. Any directed edge (u, v) in the original graph is then converted into an undirected edge (u', v'') from u's consumer node to v's producer node. Since personalized SALSA from s to t is defined as the probability that a random walk from s which alternates between following forwards and reverse edges stops at t, we see that personalized SALSA on the original graph is equal to PPR on the transformed graph.

1.6 Problem Statements

For personalized recommendations, such as who to follow on Twitter or friend recommendation on Facebook, a natural problem to solve is the δ-significant Personalized PageRank problem: Given source node or distribution s, find the the targets t with Personalized PageRank $\pi_s[t] \geq \delta$. For this problem, previous algorithms like Monte Carlo (see section 2.5) run in $O\left(\frac{1}{\delta}\right)$ time, and in the worst-case there are $\Omega\left(\frac{1}{\delta}\right)$ results to return, so there is not much improvement to be made.

In search-related applications, however, we are often interested in a small number of targets which are relevant to the search. This motivates the *Single Pair Personalized PageRank Estimation Problem*: Given source node (or distribution) s and target node t, estimate the Personalized PageRank $\pi_s[t]$ up to a small relative error. Since smaller values of $\pi_s[t]$ are more difficult to detect, we parameterize the problem by threshold δ, requiring small relative error only if $\pi_s[t] \geq \delta$.

We generalize the Single Pair PPR Estimation Problem to any given random walk length. Let a weighted, directed graph $G = (V, E)$ with normalized weight matrix W

be given (or equivalently let a Markov Chain with state space V and transition matrix W be given). The *ℓ-Step Transition Probability Estimation Problem* is the following: given an initial source distribution s over V, a target state $t \in V$ and a fixed length ℓ, compute the probability that a random walk starting at a state sampled from s stops at t after ℓ steps. This probability is equal to the tth coordinate of sW^ℓ. We parameterize by a minimum probability δ and require a small relative error if this probability is at least δ.

In a typical personalized search application, we are given a set of candidate results T for a query, and we want to find the top results ranked by Personalized PageRank. This leads us to study the *Personalized PageRank Search Problem*: Given

- a graph with nodes V (each associated with a set of keywords) and edges E (optionally weighted and directed),
- a keyword inducing a set of targets $T = \{t \in V : t \text{ is relevant to the keyword}\}$, and
- a node $s \in V$ or a distribution s over starting nodes, based on the user performing the search

return the top-k targets $t_1, \ldots, t_k \in T$ ranked by Personalized PageRank $\pi_s(t_i)$.

In future applications, it may be useful to not only score target nodes, but also compute paths to them. For example, in personalized search on a social network, we could show the user performing the search some paths between them and each search result. This motivates the *Random Path Sampling Problem*: Given a graph, a set of targets T, and a source node or distribution s, sample a random path from s of geometric length conditioned on the path ending in T.

Limitations of Prior Work Despite a rich body of existing work [28, 5, 2, 1, 7, 12, 47], past algorithms for PPR estimation or search are too slow for real-time search. For example, in Section 3.4.1, we show that on a Twitter graph with 1.5 billion edges, past algorithms take five minutes to compute PPR from a given source. Five minutes is far too long to wait for a user performing an interactive search. Analytically we can understand this slowness through the assymptotic running time of past algorithms:

On a graph with n nodes, they take $\Theta(n)$ time to estimate PPR scores of average size $1/n$. These algorithms requiring $\Theta(n)$ operations (more precicely, random memory accesses) are too slow for graphs with hundreds of millions of nodes.

1.7 High Level Algorithm

Our PPR estimators are much faster than previous estimators because they are bidirectional; they work from the target backwards and from the source forwards. For intuition, suppose we want to detect if there is a path of length ℓ between two nodes s and t in a given directed graph, and suppose for ease of analysis that every node has in-degree and out-degree d. Then finding a length-ℓ path using breadth first search forwards from s requires us to expand $O(d^\ell)$ neighbors, and similarly breadth first search backwards from t requires $O(d^\ell)$ time. In contrast, if we expand length $\ell/2$ from s and expand length $\ell/2$ from t and intersect the set of visited nodes, we can detect a length ℓ path in time $O(\sqrt{d^l})$. This is a huge difference; for example if $d = 100$ and $\ell = 4$ we are comparing order 10,000 node visits with order 100,000,000 node visits. Our algorithm achieves a similar improvement: past PPR estimation algorithms take time $\Theta(n)$ to estimate a PPR score of size $1/n$ within constant relative error with constant probability, while our algorithm on average takes time $O(\sqrt{m})$.

The challenge for us designing a bidirectional estimator is that in graphs with large degrees, doing a breadth first search of length $\ell/2$, where ℓ is the maximum walk length, will take too much time, so a bidirectional search based on path length is not fast enough. (The running time $O(\sqrt{d^\ell})$ is still exponential in ℓ, whereas our bidirectional algorithm takes time polynomial in ℓ.) Instead we go "half-way" back from the target in the sense of probability: if we're estimating a probability $\pi_s[t]$ of size $1/n$, we first find all nodes with probability greater than $\sqrt{\frac{\overline{d}}{n}}$ of reaching the target on a random walk, where $\overline{d} = m/n$ is the average degree. Then we take $O(\sqrt{m})$ random walks from s forwards. In Section 3.2 we describe how intersecting these random walks with the in-neighbors of nodes close to the target leads to a provably accurate estimator.

1.8 Contributions

The main result of this thesis is a bidirectional estimator `BidirectionalPPR` (Section 3.2) for the Single Pair Personalized PageRank Estimation Problem which is significantly faster than past algorithms in both theory and practice. In practice, we find that on six diverse graphs, our algorithm is 70x faster than past algorithms. For example, on the Twitter graph from 2010 with 1.5 billion edges, past algorithms take 5 minutes to estimate PPR from a random source to a random[3] target, while our algorithm takes less than three seconds. Our algorithm is not just a heuristic that seems to work well on some graphs, but we prove it is accurate on any graph with any edge weights.

We also analytically bound the running time of `BidirectionalPPR` on arbitrary graphs. First note that for a worst-case target, the target might have a large number of in-neighbors, making the reverse part of our bidirectional estimator very slow. For example, on Twitter, Barack Obama has more than 60 million followers, so even one reverse step iterating over his in-neighbors would take significant time. Analytically, if the target t has $\Theta(n)$ followers, then our algorithm will take $\Theta(n)$ time to estimate a score of size $1/n$. Since worst case analysis leads to weak running time guarantees for `BidirectionalPPR`, we present an average case analysis. In Theorem 2 of Section 3.2.3, we show that for a uniform random target, we can estimate a score of size at least $1/n$ to within a given accuracy ϵ with constant probability[4] in time

$$O\left(\frac{\sqrt{m}}{\epsilon}\right).$$

We prove this running time bound without making any assumptions on the graph. In contrast `Monte Carlo` (Section 2.5) takes $O\left(\frac{n}{\epsilon^2}\right)$ time for the same problem and `ReversePush` (Section 2.3) takes $O\left(\frac{n}{\epsilon}\right)$ time. To make these numbers concrete, on Twitter-2010 (a crawl of Twitter from 2010), there are $n = 40$ million nodes and

[3] This is for target sampled in proportion to their PageRank, since in practice users are more likely to search for popular targets. If targets are chosen uniformly at random, past algorithms take 2 minutes, while our algorithm takes just 30 milliseconds, and improvement of 3000x.

[4] We state results here for achieving relative error ϵ with probability 2/3, but we can guarantee relative error ϵ with probability $1 - p_{\text{fail}}$ by multiplying the running time by $\log \frac{1}{p_{\text{fail}}}$.

$m = 1.5$ billion edges, so the difference between n (past algorithms) and $\sqrt{m}$ (our algorithm) is a factor of 1000. In fact on this graph, when we choose targets uniformly at random and compare running times, our algorithm is 3000x faster than past algorithms, so this analysis explains why our algorithm is much faster than past algorithms. This estimator also applies to standard (global) PageRank, giving the best known running time for estimating the global PageRank of a single node.

For undirected graphs we propose an alternative estimator `UndirectedBiPPR` (Section 3.3) for the Single Pair PPR Estimation Problem which has worst-case running time guarantees. Before we state the running time, note that on undirected graphs, the natural global importance of a node t is $\frac{d_t}{2m}$, since this is the probability of stopping at t as the walk length goes to infinity, so only Personalized PageRank scores larger than $\frac{d_t}{2m}$ are interesting. `UndirectedBiPPR` can estimate a Personalized PageRank score $\pi_s[t]$ of size greater than $\frac{d_t}{2m}$ within relative error ϵ in expected time

$$O\left(\frac{\sqrt{m}}{\epsilon}\right).$$

This running time bound applies for a worst-case target t and worst-case undirected graph. There is a nice symmetry between `BidirectionalPPR` and `UndirectedBiPPR`, and in one preliminary experiment (Section 3.4.2) `UndirectedBiPPR` and `BidirectionalPPR` have about the same running time.

Our results are not specific to PPR, but actually generalize to a bidirectional estimator (Chapter 4) that solves the ℓ-Step Transition Probability Estimation Problem on any weighted graph (or Markov Chain) from any source distribution to any target. This estimator has relative error ϵ with constant probability. For a uniform random target, if the estimated probability is greater than δ, the expected running time is

$$O\left(\frac{\ell^{3/2}}{\epsilon}\sqrt{\overline{d}/\delta}\right)$$

where $\overline{d} = m/n$ is the average degree of a node. Compare this to the running time of Monte Carlo, $O\left(\frac{\ell}{\epsilon^2\delta}\right)$ and note that in practice ℓ, $1/\epsilon$, and $\overline{d}$ are relatively small, say less than 100, whereas δ is tiny, perhaps 10^{-8} if we are estimating a very small

probability, so the dependence on δ is crucial. This algorithm has applications in simulating stochastic systems and in estimating other random walk scores like the Heat Kernel which has been useful for community detection [31].

In practice, the `BidirectionalPPR` algorithm is slower when the target node is very popular (having many in-neighbors or high PageRank) than for average targets. In Chapter 5 we present a method of precomputing some data and distributing it among a set of servers to enable fast Single Pair PPR Estimation even when the target is popular. Analytically, with pre-computation time and space of $O\left(n\frac{\sqrt{m}}{\epsilon}\right)$ we can achieve worst-case expected running time

$$O\left(\frac{\sqrt{m}}{\epsilon}\right)$$

for any target, even targets with very large in-degree. In practice, the analysis in Section 5.3 estimates that the storage needed on the Twitter-2010 graph to pre-compute `BidirectionalPPR` is 8 TB, or 1.4 TB after an optimization. This is large, but not in-feasible, storage, and allows for very fast PPR computation for any target. For comparison, if we pre-computed Monte-Carlo walks from every source (as proposed in [21]), we would need $\Omega(n^2)$ storage, or 1600 TB.

When doing a personalized search, we typically do not have a single target, but have many candidate targets and want to find the most relevant ones, motivating the Personalized PageRank Search Problem. In Chapter 6 we describe a personalized search algorithm which can use precomputation to efficiently rank targets matching a query. In experiments on a 1.5 billion edge Twitter graph, this algorithm is more efficient than past algorithms, taking less than 0.2 seconds per query, while past algorithms take 10 seconds for some target set sizes. However, this improved running time comes at the expense of significant pre-computation, roughly 16 TB for the 2010 Twitter graph. Analytically, given $O(n\sqrt{m})$ precomputation time and space, this algorithm can respond to name search queries with rigorous approximate accuracy guarantees in time $O(\sqrt{m})$.

Finally, as another extension of these ideas, in Section 6.2 we present an algorithm for the Random Path Sampling Problem. Given a given start s and target t (or a set

of targets) this algorithm can sample a random path from s conditioned on stopping at t. Its average expected running time is

$$O\left(\sqrt{m}\right)$$

for a uniform random target. In contrast, Monte Carlo would take $\Omega(n)$ time for this task.

Chapter 2

Background and Prior Work

In this chapter we briefly consider other approaches to personalized search, and then, focusing on Personalized PageRank, we describe the prior algorithms for computing PPR which our bidirectional estimators build upon. Our bidirectional estimator uses these past estimators as subroutines, so understanding them is important for understanding our bidirectional algorithm.

2.1 Other Approaches to Personalization

The field of personalized search is large, but here we briefly describe some of the methods of personalizing search and cite a few representative papers. One method is using topic modeling based on the search history of a user [50] to improve the rank of pages whose topic matches the topic profile of a user. For example, if a user has a search history of computer science terms, while another has a search history of travel terms, that informs what to return for the ambiguous term "Java." A simpler method of using history, compared to topic-modeling methods in [17], is to boost the rank of results the user has clicked on in the past (for example, if the user has searched for "Java" before and clicked on the Wikipedia result for the language, then the Wikipedia result could be moved to the top in future searches for "Java" by that user), or to boost the rank of results similar users have clicked on in the past, where user similarity is measured using topic models. A related method is to personalize

based on a user's bookmarks [30].

Other past work has proposed using a user's social network to personalize searches. The authors of [13] score candidate results based not just on the searching user's history of document access, but also based on similar user's history of document access, where users are said to be similar if they use similar tags, share communities, or interact with the same documents. Considering the problem of name search on the Orkut social network, [55] proposes ranking results by shortest path distance from the searching user.

2.2 Power Iteration

The original method of computing PageRank and PPR proposed in [45] is known as power iteration. This method finds a fixed-point of the recursive definition Equation 1.1 by starting with an initial guess, say $p_s^0 = (\frac{1}{n}, \ldots, \frac{1}{n}) \in \mathbb{R}^n$, and repeatedly setting

$$p_s^{i+1} = \alpha e_s + (1 - \alpha) p_s^i W$$

until $\|p^{i+1} - p^i\|_\infty$ falls below some threshold. Standard power iteration analysis [26] shows that to achieve additive error δ, $O\left(\log \frac{1}{\delta}\right)$ iterations are sufficient, so the running time is $O\left(m \log \frac{1}{\delta}\right)$. On large graphs, m is more than a billion, so this technique works fine for computing global PageRank (which can be computed once and stored), but using it to personalize to every node is computationally impractical. This motivates local algorithms for PPR which focus on regions of the graph close to the source or target node and avoid iterating over the entire graph.

2.3 Reverse Push

One local variation on Power Iteration starts at a given target node t and works backwards, computing an estimate $p^t(s)$ of $\pi_s(t)$ from every source s to the given target. This technique was first proposed by Jeh and Widom [28], and subsequently improved by other researchers [21, 2]. The algorithms are primarily based on the

following recurrence relation for π_u:

$$\pi_s(t) = \alpha e_s + \frac{(1-\alpha)}{|\mathcal{N}^{out}(s)|} \cdot \sum_{v \in \mathcal{N}^{out}(s)} \pi_v(t). \tag{2.1}$$

Intuitively, this equation says that for s to decide how important t is, first s gives score α to itself, then adds the average opinion of its out-neighbors, scaled by $1-\alpha$.

Andersen et. al. [1] present and analyze a local algorithm for PPR based on this recurrence. This algorithm can be viewed as a message passing algorithm which starts with a message at the target. Each local push operation involves taking the message value (or "residual") $r^t[v]$ at some node v, incorporating $r^t[v]$ into an estimate $p^t[v]$ of $\pi_v[t]$, and sending a message to each in-neighbor $u \in \mathcal{N}^{in}(v)$, informing them that $p^t[v]$ has increased. Because we use it in our bidirectional algorithm, we give the full pseudo-code here as Algorithm 1. An example on a small four-node graph is shown

Algorithm 1 `ReversePush`$(t, r_{\max}, \alpha)$ [1]

Inputs: graph G with edge weights $(w_{u,v})_{u,v \in V}$, target node t, maximum residual $r_{\max}$, teleport probability α

Initialize (sparse) estimate-vector $p_t = \vec{0}$ and (sparse) residual-vector $r_t = e_t$ (i.e. $r_t(v) = 1$ if $v = t$; else 0)
while $\exists v \in V \; s.t. \; r_t(v) > r_{\max}$ **do**
  for $u \in \mathcal{N}^{in}(v)$ **do**
    $r_t(u) \mathrel{+}= (1-\alpha) w_{u,v} r_t(v)$
  end for
  $p_t(v) \mathrel{+}= \alpha r_t(v)$
  $r_t(v) = 0$
end while
return (p_t, r_t)

in Figure 2.1.

Andersen et. al. prove that after this algorithm completes, for all s the additive error $|\pi_s(t) - p^t(s)| \leq r_{\max}$. Their proof uses the following loop invariant (Lemma 1 in [1])

$$\pi_s[t] = p^t[s] + \sum_{v \in V} \pi_s[v] r^t[v]. \tag{2.2}$$

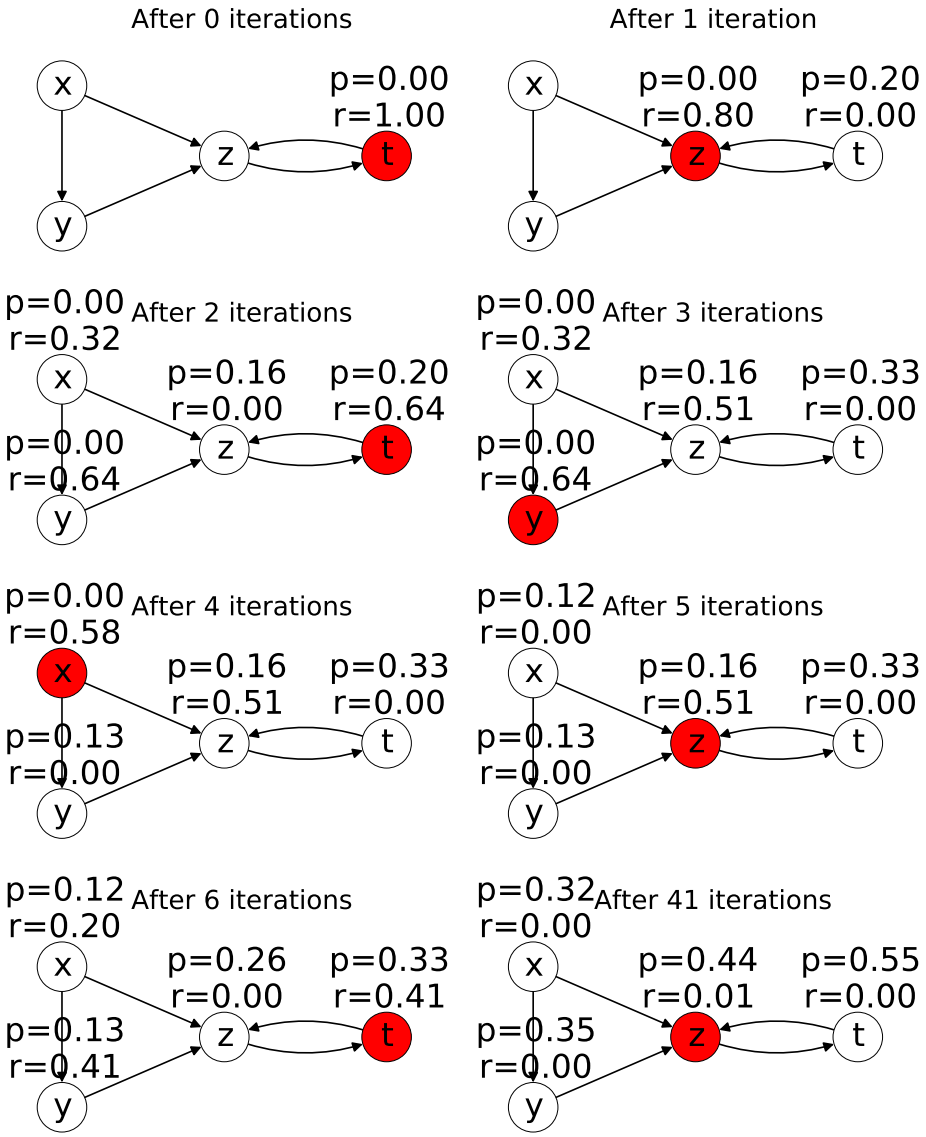


Figure 2.1: The first 7 iterations of the `ReversePush` algorithm to target t on a simple four-node graph. Each node v is annotated with its current residual $r^t[v]$ (mass that needs to be pushed back) and estimate $p^t[v]$ (estimate of $\pi_v[t]$). At each iteration, the node about to be pushed from is highlighted in red. We also show the result after 41 iterations, when all of the residuals have fallen below 0.01. If we set $r_{\max} = 0.01$ the algorithm would terminate after 41 iterations. At this point, all estimates $p^t[v] \in [\pi_v[t] - 0.01, \pi_v[t]]$.

This loop invariant can be interpreted as saying that the true value of $\pi_s[t]$ is equal to the estimate from messages processed at s plus the sum over all nodes v of their un-processed message value $r^t[v]$, weighted by their proximity to s, $\pi_s[t]$, since nearby messages are more relevant than distant messages. `ReversePush` terminates once every residual value $r^t[v] < r_{\max}$. Since $\sum_v \pi_s[v] = 1$, the convex combination $\sum_{v \in V} \pi_s[v] r^t[v] < r_{\max}$. Viewing this as an error term, Andersen et. al. observe that $p^t[s]$ estimates $\pi_s[t]$ from every $s \in V$ up to a maximum additive error of $r_{\max}$. For a given (s, t) pair, in Section 3.2 we will show how to use these residuals to get a more accurate estimate of $\pi_s(t)$.

Andersen et. al. also show that the number of push operations is at most $\frac{n\pi(t)}{r_{\max}}$, and the running time of pushing from some node v is proportional to its in-degree $d^{in}(v)$, so the total running time is proportional to $\frac{n\pi(t)}{r_{\max}}$ times the average in-degree of nodes pushed.

In [39], Lofgren and Goel show that if t is chosen uniformly at random, the average running time is

$$O\left(\frac{\bar{d}}{r_{\max}}\right)$$

where $\bar{d} = m/n$ is the average in-degree of a node. They also analyze the running time of a variant algorithm where a priority queue is used in the while loop, so the node v with the greatest residual value $r^t(v)$ is pushed on each iteration. For this variant, as $\delta \to 0$, the running time is $O\left(m \log \frac{1}{\delta}\right)$. Finally, they empirically measure the running time, and roughly find that for $\delta > \frac{1}{n}$, when running on a Twitter-based graph the running time roughly scales with $\frac{1}{\delta}$.

2.4 Forward Push

An alternative local version of power iteration starts from the start node s and works forward along edges. Variations on this were proposed in [10, 28] and others, but the variation most useful for our work is in Andersen et. al. [2] because of the analysis they give. Because we use it a variation of our bidirectional algorithm, we give the full pseudo-code here as Algorithm 2.

Algorithm 2 `ForwardPush`$(G, \alpha, s, r_{\max})$ [2]

Inputs: undirected graph G, teleport probability α, start node s, maximum residual $r_{\max}$

Initialize (sparse) estimate-vector $p_s = \vec{0}$ and (sparse) residual-vector $r_s = e_s$ (i.e. $r_s[v] = 1$ if $v = s$; else 0)
while $\exists u \in V$ *s.t.* $\frac{r_s[u]}{d_u} > r_{\max}$ **do**
  for $v \in \mathcal{N}[u]$ **do**
    $r_s[v]$ += $(1-\alpha)r_s[u]/d_u$
  end for
  $p_s[u]$ += $\alpha r_s[u]$
  $r_s[u] = 0$
end while
return (p_s, r_s)

To our knowledge, there is no clean bound on the error $\|p_s - \pi_s\|$ as a function of $r_{\max}$ for a useful choice of norm. The difficulty is illustrated by the following graph: we have n nodes, $\{s, t, v_1, \ldots, v_{n-2}\}$, and $2(n-2)$ edges, (s, v_i) and (v_i, t) for each i. If we run `ForwardPush` on this graph starting from s with $r_{\max} = \frac{1}{n-2}$, then after pushing from s, the algorithm halts, with estimate at t of $p_s(t) = 0$. However, $\pi_s(t) = \Omega(1)$, so the algorithm has a large error at t even though $r_{\max}$ is getting arbitrarily small as $n \to \infty$.

The loop invariant in [2] does give a bound on the error of `ForwardPush`, but it is somewhat subtle, involving the personalized PageRank personalized to the resulting residual vector. In Equation (3.9) of Section 3.3.3 we give an additive error bound in terms of $r_{\max}$ and the target's degree d_t and PageRank $\pi(t)$ which applies on undirected graphs.

2.5 Monte Carlo

The random walk definition of personalized PageRank, Equation (1.2), inspires a natural randomized algorithm for estimating PPR. We simple sample some number w of random walks from the start node (or distribution) s and see what empirical fraction of them end at each target. This technique of simulating a random process

and empirically measuring probabilities or expectations is known as Monte Carlo. More formally, for each $i \in [w]$ we sample length $L_i \sim \text{Geometric}(1/\alpha)$, take a random walk of length L_i starting from s, and and let V_i be the endpoint. Then we define

$$\tilde{\pi}_s(t) = \frac{\#\{i : V_i = t\}}{w}.$$

It is immediate from Equation 1.2 that for any t, $\mathbb{E}[\tilde{\pi}_s(t)] = \pi_s[t]$. If we want to estimate a probability $\pi_s(t)$ of size δ to within relative error ϵ with constant probability, then standard Chernoff bounds like those used in Theorem 1 imply that

$$w = O\left(\frac{1}{\epsilon^2 \delta}\right)$$

is a sufficient number of random walks. These walks can be computed as needed or computed in advance, and various papers [5, 7, 12, 47] consider different variations. Such estimates are easy to update in dynamic settings [7]. However, for estimating PPR values close to the desired threshold δ, these algorithms need $\Omega(1/\delta)$ random-walk samples, which makes them slow for estimating the PPR between a random pair of nodes, since the expected PPR score is $1/n$, so their running time is $\Omega(n)$.

2.6 Algorithms using Precomputation and Hubs

In [10], Berkhin builds upon the previous work by Jeh and Widom [28] and proposes efficient ways to compute the personalized PageRank vector π_s at run time by combining pre-computed PPR vectors in a query-specific way. In particular, they identify "hub" nodes in advance, using heuristics such as global PageRank, and precompute approximate PPR vectors $\hat{\pi}_h$ for each hub node using a local forward-push algorithm called the Bookmark Coloring Algorithm (BCA). Chakrabarti [14] proposes a variant of this approach, where Monte-Carlo is used to pre-compute the hub vectors $\hat{\pi}_h$ rather than BCA.

Both approaches differ from our work in that they construct complete approximations to π_s, then pick out entries relevant to the query. This requires a high-accuracy

estimate for π_s even though only a few entries are important. In contrast, our bidirectional approach allows us compute only the entries $\pi_s(t_i)$ relevant to the query.

Chapter 3

Bidirectional PPR Algorithms

In this chapter we present our bidirectional estimators for Personalized PageRank. We briefly describe the first estimator we developed, FAST-PPR, since its main idea gives useful intuition. Then we fully describe a simpler, more efficient estimator, `BidirectionalPPR`, as well an alternative estimator for undirected graphs `UndirectedBiPPR`. Finally we demonstrate the speed and accuracy of our estimators in experiments on real graphs.

3.1 FAST-PPR

The first bidirectional estimator we proposed was FAST-PPR [40]. Its main idea is that the personalized PageRank from s to t can be expressed as a sum

$$\pi_s[t] = \sum_{w \in F} \mathbb{P}\left[\text{a walk from } s \text{ first hits } F \text{ at } w\right] \cdot \pi_w[t] \tag{3.1}$$

where F is any set which intercepts every path from s to t. Think of F as some set surrounding the set of nodes "close" to the target, as shown in Figure 3.1. In FAST-PPR, we say a node w is "close" to the target if $\pi_w[t]$ is above some threshold, and F is the set of in-neighbors of nodes "close" to the target. FAST-PPR uses the `ReversePush` algorithm (Section 2.3) to estimate the values $\pi_w[t]$ for $w \in F$, then uses Monte Carlo random walks to estimate the value of Equation 3.1. FAST-PPR

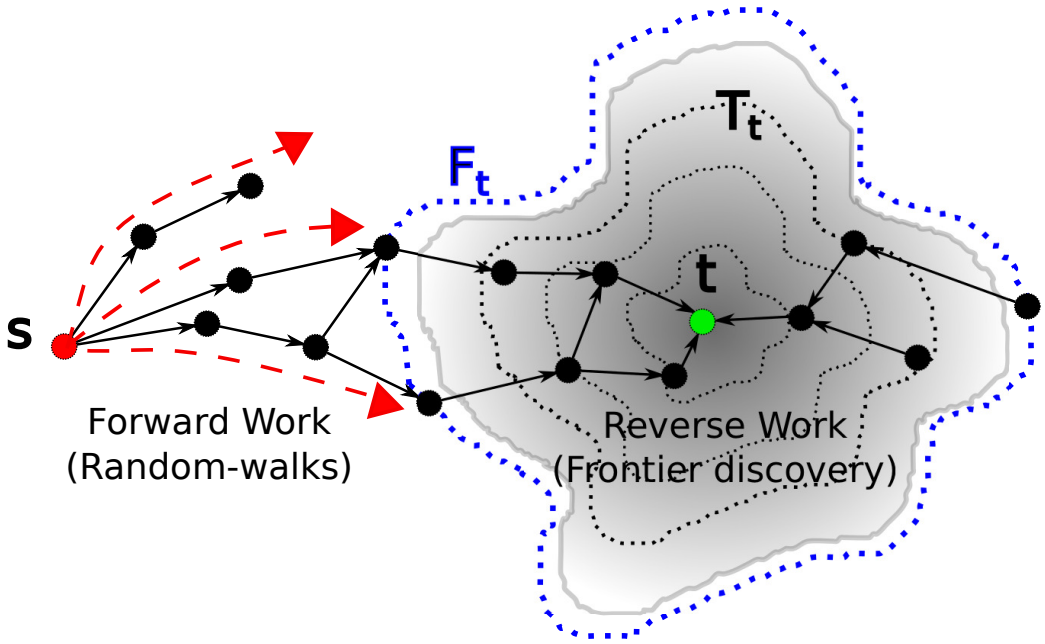


Figure 3.1: The FAST-PPR Algorithm: We first work backward from the target t to find the frontier F_t (with inverse-PPR estimates). Next we work forward from the source s, generating random-walks and testing for hitting the frontier.

has been superseeded [1] by `BidirectionalPPR` so we now move on to describing that estimator.

3.2 Bidirectional-PPR

In this section we present our new bidirectional algorithm for PageRank estimation [38]. At a high level, our algorithm estimates $\pi_s[t]$ by first working backwards from t to find a set of intermediate nodes 'near' t and then generating random walks forwards from s to detect this set. Our method of combining reverse estimates with forward random walks is based on the residual vector returned by the `ReversePush` algorithm (given in Section 2.3 as Algorithm 1) of Andersen et. al. [1]. Our `BidirectionalPPR` algorithm is based on the observation that in order to estimate $\pi_s[t]$ for a *particular* (s, t) pair, we can boost the accuracy of `ReversePush` by sampling random walks from s and incorporating the residual values at their endpoints.

3.2.1 The `BidirectionalPPR` Algorithm

The reverse work from t is done via the `ReversePush`$(t, r_{\max}, \alpha)$, where $r_{\max}$ is a parameter we discuss more in Section 3.2.3. Recall from Section 2.3 that this algorithm produces two non-negative vectors $p^t \in \mathbb{R}^n$ (estimates) and $r^t \in \mathbb{R}^n$ (residual messages) which satisfy the following invariant (Lemma 1 in [1])

$$\pi_s[t] = p^t[s] + \sum_{v \in V} \pi_s[v] r^t[v]. \tag{3.2}$$

The central idea is to reinterpret Equation (3.2) as an expectation:

$$\pi_s[t] = p^t[s] + \mathbb{E}_{v \sim \pi_s}[r^t[v]]. \tag{3.3}$$

[1] The difficulty in FAST-PPR is how to estimate the values $\pi_w[t]$ for $w \in F$ accuratly. The `ReversePush` algorithm gives an additive error guarantee for nodes in F, whereas the analysis needs a relative error guarantee. This issue makes FAST-PPR slower in our experiments than `BidirectionalPPR` to achieve the same accuracy and complicates FAST-PPR's theoretical analysis.

Now, since $\max_v r^t[v] \leq r_{\max}$, the expectation $\mathbb{E}_{v\sim\pi_s[v]}[r^t[v]]$ can be efficiently estimated using Monte Carlo. To do so, we generate $w = c\frac{r_{\max}}{\delta}$ random walks of length $Geometric(\alpha)$ from start node s. This choice of w is based on Theorem 1, which provides a value for c based on the desired accuracy, and here δ is the minimum PPR value we want to accurately estimate. Let V_i be the final node of the ith random walk. From Equation 1.2, $\Pr[V_i = v] = \pi_s[v]$. Our algorithm returns the natural emperical estimate for Equation 3.3:

$$\widehat{\pi}_s[t] = p^t[s] + \frac{1}{w}\sum_{i=1}^{w} r^t[V_i]$$

The complete pseudocode is given in Algorithm 3.

Algorithm 3 `BidirectionalPPR`(s, t, δ)

Inputs: graph G, teleport probability α, start node s, target node t, minimum probability δ, accuracy parameter c (in our experiments we use $c = 7$)

1: $(p_t, r_t) =$ `ReversePush`$(t, r_{\max}, \alpha)$
2: Set number of walks $w = cr_{\max}/\delta$ (cf. Theorem 1)
3: **for** index $i \in [w]$ **do**
4: Sample a random walk starting from s (sampling a start from s if s is a distribution), stopping after each step with probability α, and let V_i be the endpoint
5: **end for**
6: **return** $\widehat{\pi}_s[t] = p_t[s] + (1/w)\sum_{i\in[w]} r_t[V_i]$

As an example, consider the graph in Figure 3.2. Suppose we run `ReversePush` on this graph with target t and teleport probability α to accuracy $r_{\max} = 0.11$. The resulting PPR estimates p and residuals r are shown in Figure 3.3. In the second phase of the algorithm we take random walks forwards. If we set minimum PPR $\delta = 0.01$ and chernoff constant, $c = 7$, the number of walks we need to do is $w = c\frac{r_{\max}}{\delta} = 77$ If we perform 77 walks, we might get the empirical forward probabilities $\tilde{\pi}_s$ shown in Figure 3.4. If we look at the residuals r from Figure 3.3 and the forward probabilities $\tilde{\pi}_s$ in Figure 3.4 we see the only node with nonzero value on both happens to be c, so

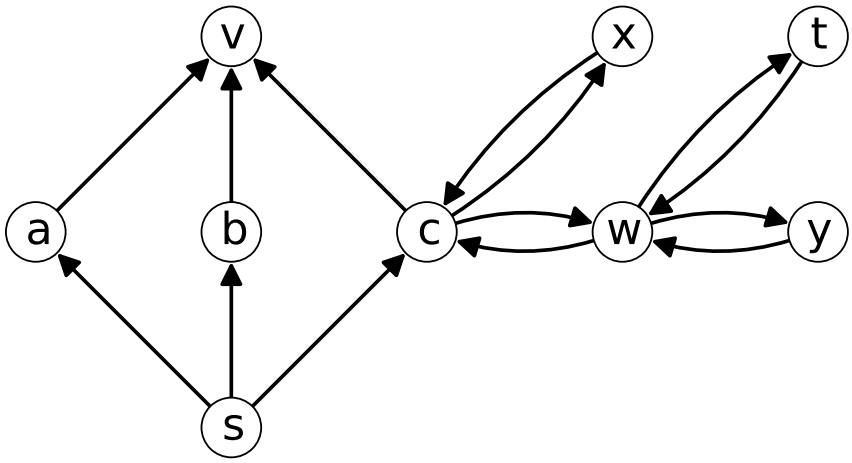


Figure 3.2: An example graph. We will estimate $\pi_s[t]$.

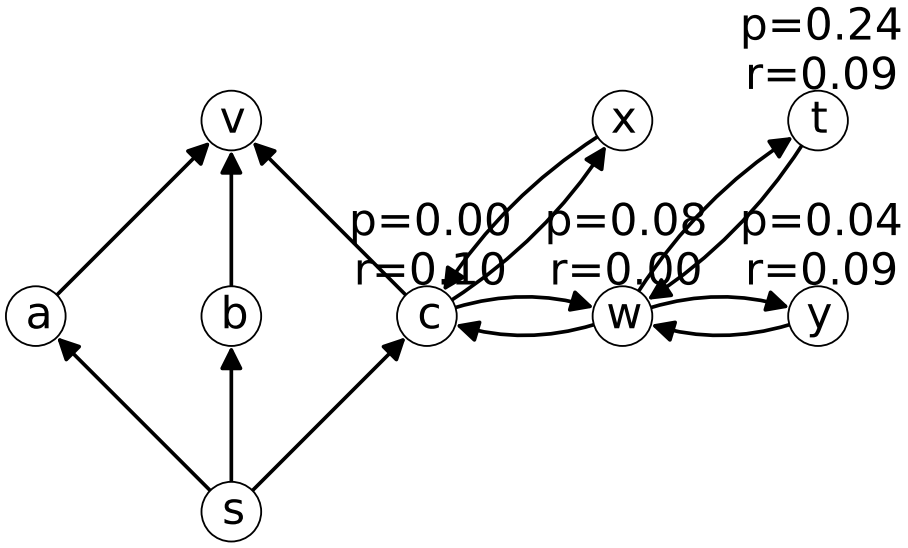


Figure 3.3: An example graph. We will estimate $\pi_s[t]$.

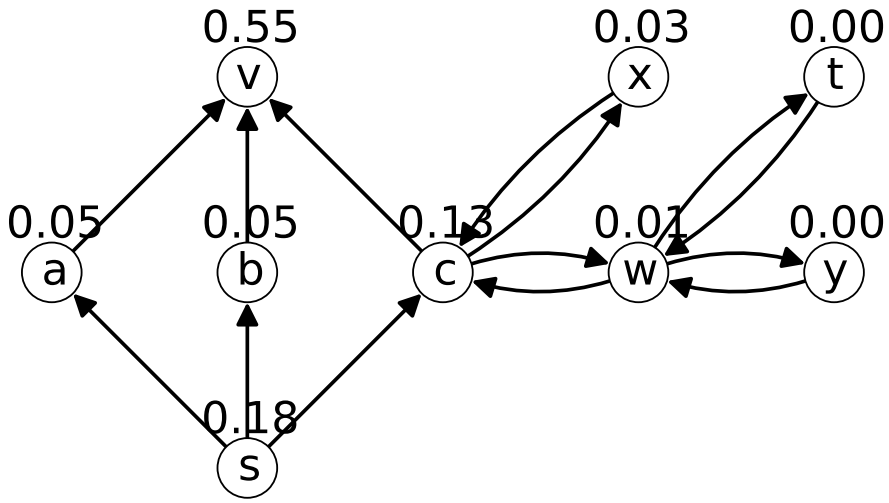


Figure 3.4: The empirical probability of stopping at each node after 77 walks from s for some choice of random seed. Note there is an implicit self loop at v.

our estimate for this run of the algorithm is

$$\hat{\pi}_s[t] = p^t[s] + \sum_v \tilde{\pi}_s[v] r^t[v] = 0 + 0.13 \cdot 0.1 = 0.013.$$

3.2.2 Accuracy Analysis

We first prove that high probability, `BidirectionalPPR` returns an estimate with small relative error if $\pi_s[t]$ is above a given threshold, and returns an estimate with small additive error otherwise.

Theorem 1. *Given start node s (or source distribution σ), target t, minimum PPR δ, maximum residual $r_{max} > \frac{2e\delta}{\alpha\epsilon}$, relative error $\epsilon \leq 1$, and failure probability p_{fail}, `BidirectionalPPR` outputs an estimate $\widehat{\pi}_s[t]$ such that with probability at least $1 - p_{fail}$ the following hold:*

- *If $\pi_s[t] \geq \delta$:* $|\pi_s[t] - \hat{\pi}_s[t]| \leq \epsilon \pi_s[t]$.
- *If $\pi_s[t] \leq \delta$:* $|\pi_s[t] - \hat{\pi}_s[t]| \leq 2e\delta$.

The above result shows that the estimate $\hat{\pi}_s[t]$ can be used to distinguish between

'significant' and 'insignificant' PPR pairs: for pair (s,t), Theorem 1 guarantees that if $\pi_s[t] \geq \frac{(1+2e)\delta}{(1-\epsilon)}$, then the estimate is greater than $(1+2e)\delta$, whereas if $\pi_s[t] < \delta$, then the estimate is less than $(1+2e)\delta$. The assumption $r_{\max} > \frac{2e\delta}{\alpha\epsilon}$ is easily satisfied, as typically $\delta = O\left(\frac{1}{n}\right)$ and $r_{\max} = \Omega\left(\frac{1}{\sqrt{m}}\right)$.

Proof. As shown in Algorithm 3, we will average over

$$w = c\frac{r_{\max}}{\delta},$$

walks, where c is a parameter we choose later. Each walk is of length $Geometric(\alpha)$, and we denote V_i as the last node visited by the i^{th} walk so that $V_i \sim \pi_s$. Now let $X_i = r^t[V_i]$. The estimate returned by `BidirectionalPPR` is:

$$\widehat{\pi}_s[t] = p^t[s] + \frac{1}{w}\sum_{i=1}^{w} X_i.$$

First, from Equation (3.2), we have that $\mathbb{E}[\widehat{\pi}_s[t]] = \pi_s[t]$, that is, our estimate is unbiased. Since `ReversePush` guarantees that for all v, $r^t[v] < r_{\max}$, each X_i is bounded in $[0, r_{\max}]$. Before we apply Chernoff bounds, we rescale X_i by defining $Y_i = \frac{1}{r_{\max}} X_i \in [0,1]$. We also define $Y = \sum_{i=1}^{w} Y_i$

We show concentration of the estimate via the following two Chernoff bounds (see Theorem 1.1 in [18]):

1. $\mathbb{P}[|Y - \mathbb{E}[Y]| > \epsilon\mathbb{E}[Y]] < 2\exp(-\frac{\epsilon^2}{3}\mathbb{E}[Y])$
2. For any $b > 2e\mathbb{E}[Y], \mathbb{P}[Y > b] \leq 2^{-b}$

We perform a case analysis based on whether $\mathbb{E}[X_i] \geq \delta$ or $\mathbb{E}[X_i] < \delta$.

First suppose $\mathbb{E}[X_i] \geq \delta$. Note that this implies that $\pi_s[t] \geq \delta$ so we will prove a

relative error bound of ϵ. Now we have $\mathbb{E}[Y] = \frac{w}{r_{\max}}\mathbb{E}[X_i] = \frac{c}{\delta}\mathbb{E}[X_i] \geq c$, and thus:

$$\begin{aligned}
\mathbb{P}[|\widehat{\pi}_s[t] - \pi_s[t]| > \epsilon\pi_s[t]] &\leq \mathbb{P}[\left|\bar{X} - \mathbb{E}[X_i]\right| > \epsilon\mathbb{E}[X_i]] \\
&= \mathbb{P}[|Y - \mathbb{E}[Y]| > \epsilon\mathbb{E}[Y]] \\
&\leq 2\exp\left(-\frac{\epsilon^2}{3}\mathbb{E}[Y]\right) \\
&\leq 2\exp\left(-\frac{\epsilon^2}{3}c\right) \leq p_{\text{fail}},
\end{aligned}$$

where the last line holds as long as we choose

$$c \geq \frac{3}{\epsilon^2}\ln\left(\frac{2}{p_{\text{fail}}}\right).$$

Suppose alternatively that $\mathbb{E}[X_i] < \delta$. Then

$$\begin{aligned}
\mathbb{P}[|\hat{\pi}_s[t] - \pi_s[t]| > 2e\delta] &= \mathbb{P}[\left|\bar{X} - \mathbb{E}[X_i]\right| > 2e\delta] \\
&= \mathbb{P}\left[|Y - \mathbb{E}[Y]| > \frac{w}{r_{\max}}2e\delta\right] \\
&\leq \mathbb{P}\left[Y > \frac{w}{r_{\max}}2e\delta\right].
\end{aligned}$$

At this point we set $b = \frac{w}{r_{\max}}2e\delta = 2ec$ and apply the second Hoeffding bound. Note that $\mathbb{E}[Y] = \frac{c}{\delta}\mathbb{E}[X_i] < c$, and hence we satisfy $b > 2e\mathbb{E}[Y]$. The second bound implies that

$$\mathbb{P}[|\hat{\pi}_s[t] - \pi_s[t]| > 2e\delta] \leq 2^{-b} \leq p_{\text{fail}} \tag{3.4}$$

as long as we choose c such that:

$$c \geq \frac{1}{2e}\log_2\frac{1}{p_{\text{fail}}}.$$

If $\pi_s[t] \leq \delta$, then equation 3.4 completes our proof.

The only remaining case is when $\pi_s[t] > \delta$ but $\mathbb{E}[X_i] < \delta$. This implies that $p^t[s] > 0$ since $\pi_s[t] = p^t[s] + \mathbb{E}[X_i]$. In the `ReversePush` algorithm when we increase

$p^t[s]$, we always increase it by at least $\alpha r_{\max}$, so we have $\pi_s[t] \geq \alpha r_{\max}$. We have that

$$\frac{|\hat{\pi}_s[t] - \pi_s[t]|}{\pi_s[t]} \leq \frac{|\hat{\pi}_s[t] - \pi_s[t]|}{\alpha r_{\max}}.$$

By assumption, $\frac{2e\delta}{\alpha r_{\max}} < \epsilon$, so by equation 3.4,

$$\mathbb{P}\left[\frac{|\hat{\pi}_s[t] - \pi_s[t]|}{\pi_s[t]} > \epsilon\right] \leq p_{\text{fail}}$$

The proof is completed by combining all cases and choosing $c = \frac{3}{\epsilon^2} \ln\left(\frac{2}{p_{\text{fail}}}\right)$. We note that the constants are not optimized; in experiments we find that $c = 7$ gives mean relative error less than 8% on a variety of graphs. □

3.2.3 Running Time Analysis

The runtime of `BidirectionalPPR` depends on the target t: if t has many in-neighbors and/or large global PageRank $\pi[t]$, then the running time will be slower because the reverse pushes will iterate over a large number of nodes. For example, in the worst case, we might have $d^{in}(t) = \Theta(n)$ and `BidirectionalPPR` takes $\Theta(n)$ time. However, we can give an average case analysis, and in Section 3.4.2 we give a back-of-the-envelope parameterized running time estimate.

Average Case Running Time Analysis

Alternatively, for *a uniformly chosen target node*, we can prove the following:

Theorem 2. *For any start node s (or source distribution σ), minimum PPR δ, maximum residual r_{max}, relative error ϵ, and failure probability p_{fail}, if the target t is chosen uniformly at random, then* `BidirectionalPPR` *has expected running time*

$$O\left(\sqrt{\frac{\bar{d}}{\delta}} \frac{\sqrt{\log(1/p_{fail})}}{\alpha\epsilon}\right).$$

In contrast, the running time for `Monte-Carlo` to achieve the same accuracy guarantee is $O\left(\frac{1}{\delta}\frac{\log(1/p_{\text{fail}})}{\alpha\epsilon^2}\right)$, and the running time for `ReversePush` is $O\left(\frac{\bar{d}}{\delta\alpha}\right)$. The `FAST-PPR` algorithm of [40] has an average running time bound of $O\left(\frac{1}{\alpha\epsilon^2}\sqrt{\frac{\bar{d}}{\delta}}\sqrt{\frac{\log(1/p_{\text{fail}})\log(1/\delta)}{\log(1/(1-\alpha))}}\right)$ for uniformly chosen targets. The running time bound of `BidirectionalPPR` is thus asymptotically better than `FAST-PPR`, and in our experiments (Section 3.4.1) `BidirectionalPPR` is significantly faster than past algorithms.

Proof. We first show that for a uniform random t, `ReversePush` runs in average time $\frac{\bar{d}}{\alpha r_{\max}}$ where $\bar{d}$ is the average degree of a node. This is because each time we push from a node v, we increase $p^t[v]$ by at least $\alpha r_{\max}$. Since $p^t[v] \leq \pi_v[t]$ and $\sum_t \pi_v[t] = 1$, this lets us bound the average running time:

$$\begin{aligned}
&\frac{1}{n}\sum_{t\in V}\sum_{v\in V} d^{in}(v)[\#\text{ times } v \text{ is pushed during } \texttt{ReversePush}(t, r_{\max}, \alpha)] \\
&\leq \frac{1}{n}\sum_{t\in V}\sum_{v\in V} d^{in}(v)\frac{\pi_v[t]}{\alpha r_{\max}} \\
&\leq \frac{1}{n}\sum_{v\in V} d^{in}(v)\frac{1}{\alpha r_{\max}} \\
&\leq \frac{\bar{d}}{\alpha r_{\max}}.
\end{aligned}$$

On the other hand, from Theorem 1, the number of walks generated is $O\left(\frac{r_{\max}}{\delta\epsilon^2}\ln(1/p_{\text{fail}})\right)$ random walks, each of which can be sampled in average time $1/\alpha$. Finally, we choose $r_{\max} = \epsilon\sqrt{\frac{\bar{d}\delta}{\ln(2/p_{\text{fail}})}}$ to minimize our running time bound and get the claimed result. □

A Practical Improvement: Balancing Forward and Reverse Running Time

The dynamic runtime-balancing heuristic proposed in [40] can improve the running time of Bidirectional-PageRank in practice. In this technique, $r_{\max}$ is chosen dynamically in order to balance the amount of time spent by `ReversePush` and the amount of time spent generating random walks. To implement this, we modify `ReversePush` to use a priority queue in order to always push from the node v with the largest value

of $r_t[v]$. Then we change the the while loop so that it terminates when the amount of time spent achieving the current value of $r_{\max}$ first exceeds the predicted amount of time required for sampling random walks, $c_{\text{walk}} \cdot c \cdot \frac{r_{\max}}{\delta}$, where c_{walk} is the average time it takes to sample a random walk. For a plot showing how using a fixed value for $r_{\max}$ results in unbalanced running time for many targets, and how this variant empirically balances the reverse and forward times, see [40]. The complete pseudocode is given as Algorithm 4 and Algorithm 5.

Algorithm 4 `ReversePushBalanced`(G, t, α, δ) [1]

Inputs: graph G with edge weights $w_{u,v}$, teleport probability α, target node t, minimum probability δ, accuracy parameter c (in our experiments we use $c = 7$)

Choose tuning constant $c_{\text{walk time}}$, the average time it takes to generate a walk.
Initialize (sparse) estimate-vector $p^t = \vec{0}$ and (sparse) residual-vector $r^t = e^t$ (i.e. $r^t(v) = 1$ if $v = t$; else 0)
Initial Max-Priority-Queue q with priority r^t
Start stopwatch w
Define function forwardTime$(r_{\max}) = c_{\text{walk time}} c \frac{r_{\max}}{\delta}$
while Q.nonEmpty() and w.elapsedTime() < forwardTime(Q.maxPrioriority()) **do**
    Let $v = Q$.popMax()
    for $u \in \mathcal{N}^{in}(v)$ **do**
        $r^t(u)$ += $(1 - \alpha) w_{u,v} r^t(v)$ (increasing u's priority in Q)
    end for
    $p^t(v)$ += $\alpha r^t(v)$
    $r^t(v) = 0$
end while
$r_{\max} = Q$.maxPriority() if Q is non-empty, otherwise $r_{\max} = 0$
return $(p^t, r^t, r_{\max})$

Choosing the Minimum Probability δ

So far we have assumed that the minimum probability δ is either given or is $O\left(\frac{1}{n}\right)$, but a natural choice in applications when estimating $\pi_s[t]$ is to set $\delta = \pi[t]$. The motivation for this is that if $\pi_s[t] < \pi[t]$, then s is less interested in t than a random source is, so it is not useful to quantify just how (un)interested s is in t.

Algorithm 5 `Bidirectional-PPR-Balanced`(s, t, δ)

Inputs: graph G, teleport probability α, start node s, target node t, minimum probability δ, accuracy parameter c (in our experiments we use $c = 7$)

1: $(p^t, r^t, r_{\max}) =$ `ReversePushBalanced`(G, t, α, δ)
2: Set number of walks $w = cr_{\max}/\delta$ (for c's value, see Theorem 1; we use $c = 7$ in experiments)
3: **for** index $i \in [w]$ **do**
4: Sample a random walk starting from s (sampling a start from s if s is a distribution), stopping after each step with probability α; let V_i be the endpoint
5: Set $X_i = r^t[V_i]$
6: **end for**
7: **return** $\widehat{\pi}_s[t] = p^t[s] + (1/w)\sum_{i\in[w]} X_i$

Generalization

Bidirectional-PageRank extends naturally to generalized PageRank using a source distribution σ rather than a single start node – we simply sample an independent starting node from s for each walk, and replace $p^t[s]$ with $\sum_{v\in V} p^t[v]s[v]$ where $s[v]$ is the weight of v in the start distribution s. It also generalizes naturally to weighted graphs if we simply use the weights when sampling neighbors during the random walks. For a generalization to other walk distributions or to any given walk length, see Chapter 4.

3.3 Undirected-Bidirectional-PPR

Here we present an alternative bidirectional estimator for undirected graphs [36].

3.3.1 Overview

The `BidirectionalPPR` presented above is based on the `ReversePush` algorithm (introduced in Section 2.3) from [1] which satisfies the loop invariant

$$\pi_s[t] = p^t[s] + \sum_{v\in V} \pi_s[v]r^t[v]. \tag{3.5}$$

However, an alternative local algorithm is the `ForwardPush` algorithm (see Section 2.4) of Andersen et al. [2], which returns estimates[2] p_s and residuals r_s which satisfy the invariant

$$\pi_s[t] = p_s[t] + \sum_{v \in V} r_s[v]\pi_v[t], \qquad \forall\, t \in V. \tag{3.6}$$

On undirected graphs this invariant is the basis for an alternative algorithm for personalized PageRank which we call `UndirectedBiPPR`. In a preliminary experiment on one graph, this algorithm has comparable running time to `BidirectionalPPR`, but it is interesting because because of its symmetry to `BidirectionalPPR` and because it enables a parameterized worst-case analysis. We now present our new bidirectional algorithm for PageRank estimation in undirected graphs.

3.3.2 A Symmetry for PPR in Undirected Graphs

The `UndirectedBiPPR` Algorithm critically depends on an underlying *reversibility property* exhibited by PPR vectors in undirected graphs. This property, stated before in several earlier works [4, 24], is a direct consequence of the reversibility of random walks on undirected graphs. To keep our presentation self-contained, we present this property, along with a simple probabilistic proof, in the form of the following lemma:

Lemma 1. *Given any undirected graph G, for any teleport probability $\alpha \in (0,1)$ and for any node-pair $(s,t) \in V^2$, we have:*

$$\pi_s[t] = \frac{d_t}{d_s}\pi_t[s].$$

Proof. For path $P = \{s, v_1, v_2, \ldots, v_k, t\}$ in G, we denote its length as $\ell(P)$ (here $\ell(P) = k+1$), and define its reverse path to be $\overline{P} = \{t, v_k, \ldots, v_2, v_1, s\}$ – note that $\ell(P) = \ell(\overline{P})$. Moreover, we know that a random-walk starting from s traverses path

[2]We use the superscript $p^t \in \mathbb{R}^n$ vs subscript $p_s \in \mathbb{R}^n$ to indicate which type of estimate vector we are referring to (reverse or forward, respectively). We use a similar convention for reverse residual vector $r^t \in \mathbb{R}^n$ to target $t \in V$ and forward residual vector $r^s \in \mathbb{R}^n$ from source $s \in V$.

P with probability $\mathbb{P}[P] = \frac{1}{d_s} \cdot \frac{1}{d_{v_1}} \cdot \ldots \cdot \frac{1}{d_{v_k}}$, and thus, it is easy to see that we have:

$$\mathbb{P}[P] \cdot d_s = \mathbb{P}[\overline{P}] \cdot d_t \tag{3.7}$$

Now let $\mathcal{P}_{st}$ denote the set of paths in G starting at s and terminating at t. Then we can re-write Eqn. (1.2) as:

$$\pi_s[t] = \sum_{P \in \mathcal{P}_{st}} \alpha(1-\alpha)^{\ell(P)} \mathbb{P}[P] = \sum_{\overline{P} \in \mathcal{P}_{ts}} \alpha(1-\alpha)^{\ell(\overline{P})} \mathbb{P}[\overline{P}] = \frac{d_t}{d_s} \pi_t[s] \quad \square$$

$\square$

3.3.3 The `UndirectedBiPPR` Algorithm

At a high level, the `UndirectedBiPPR` algorithm has two components:

- **Forward-work**: Starting from source s, we first use a forward local-update algorithm, the `ForwardPush`$(G, \alpha, s, r_{\max})$ algorithm of Andersen et al. [2] (given in section 2.4 as Algorithm 2). This procedure begins by placing one unit of "residual" probability-mass on s, then repeatedly selecting some node u, converting an α-fraction of the residual mass at u into probability mass, and pushing the remaining residual mass to u's neighbors. For any node u, it returns an estimate $p_s[u]$ of its PPR $\pi_s[u]$ from s as well as a residual $r_s[u]$ which represents un-pushed mass at u.
- **Reverse-work**: We next sample random walks of length $L \sim Geometric(\alpha)$ starting from t, and use the residual at the terminal nodes of these walks to compute our desired PPR estimate. Our use of random walks backwards from t depends critically on the symmetry in undirected graphs presented in Lemma 1.

Note that this is in contrast to `FAST-PPR` and `BidirectionalPPR`, which performs the local-update step in reverse from the target t, and generates random-walks forwards from the source s.

In more detail, our algorithm will choose a maximum residual parameter $r_{\max}$, and apply the local push operation in Algorithm 2 until for all v, $r_s[v]/d_v < r_{\max}$. Andersen et al. [2] prove that their local-push operation preserves the following invariant for

vectors (p_s, r_s):

$$\pi_s[t] = p_s[t] + \sum_{v \in V} r_s[v]\pi_v[t], \qquad \forall\, t \in V. \tag{3.8}$$

Since we ensure that $\forall v, r_s[v]/d_v < r_{\max}$, it is natural at this point to use the symmetry Lemma 1 and re-write this as:

$$\pi_s[t] = p_s[t] + d_t \sum_{v \in V} \frac{r_s[v]}{d_v}\pi_t[v].$$

Now using the fact that $\sum_t \pi_v[t] = n\pi[t]$ get that $\forall\, t \in V$,

$$|\pi_s[t] - p_s[t]| \leq r_{\max} d_t n \pi[t]. \tag{3.9}$$

However, we can get a more accurate estimate by using the residuals. The key idea of our algorithm is to re-interpret this as an expectation:

$$\pi_s[t] = p_s[t] + d_t \mathbb{E}_{V \sim \pi_t}\left[\frac{r_s[v]}{d_V}\right]. \tag{3.10}$$

We estimate the expectation using standard Monte-Carlo. Let $V_i \sim \pi_t$ and $X_i = r_s[V_i]d_t/d_{V_i}$, so we have $\pi_s[t] = p_s[t] + \mathbb{E}[X]$. Moreover, each sample X_i is bounded by $d_t r_{\max}$ (this is the stopping condition for `ForwardPush`), which allows us to efficiently estimate its expectation. To this end, we generate w random walks, where

$$w = \frac{c}{\epsilon^2}\frac{r_{\max}}{\delta/d_t}.$$

The choice of c is specified in Theorem 3. Finally, we return the estimate:

$$\widehat{\pi}_s[t] = p_t[s] + \frac{1}{w}\sum_{i=1}^{w} X_i.$$

The complete pseudocode is given in Algorithm 6.

Algorithm 6 `UndirectedBiPPR`(s, t, δ)

Inputs: graph G, teleport probability α, start node s, target node t, minimum probability δ, accuracy parameter $c = 3\ln(2/p_{\text{fail}})$ (cf. Theorem 3)

1: $(p_s, r_s) =$ `ForwardPush`$(s, r_{\max})$
2: Set number of walks $w = cd_t r_{\max}/(\epsilon^2\delta)$
3: **for** index $i \in [w]$ **do**
4: Sample a random walk starting from t, stopping after each step with probability α; let V_i be the endpoint
5: Set $X_i = r_s[V_i]/d_{V_i}$
6: **end for**
7: **return** $\widehat{\pi}_s[t] = p_s[t] + (1/w)\sum_{i\in[w]} X_i$

3.3.4 Analyzing the Performance of `UndirectedBiPPR`

Accuracy Analysis

We first prove that `UndirectedBiPPR` returns an unbiased estimate with the desired accuracy:

Theorem 3. *In an undirected graph G, for any source node s, minimum threshold δ, maximum residual r_{max}, relative error ϵ, and failure probability p_{fail}, Algorithm 6 outputs an estimate $\widehat{\pi}_s[t]$ such that with probability at least $1 - p_{fail}$ we have: $|\pi_s[t] - \hat{\pi}_s[t]| \leq \max\{\epsilon\pi_s[t], 2e\delta\}$.*

The proof follows a similar outline as the proof of Theorem 1. For completeness, we sketch the proof here:

Proof. As stated in Algorithm 6, we average over $w = cd_t r_{\max}/\epsilon^2\delta$ walks, where c is a parameter we choose later. Each walk is of length $Geometric(\alpha)$, and we denote V_i as the last node visited by the i^{th} walk; note that $V_i \sim \pi_t$. As defined above, let $X_i = r_s[V_i]d_t/d_{V_i}$; the estimate returned by `UndirectedBiPPR` is:

$$\widehat{\pi}_s[t] = p_t[s] + \frac{1}{w}\sum_{i=1}^{w} X_i.$$

First, from Eqn. (3.10), we have that $\mathbb{E}[\widehat{\pi}_s[t]] = \pi_s[t]$. Also, `ForwardPush` guarantees that for all v, $r_s[v] < d_v r_{\max}$, and so each X_i is bounded in $[0, d_t r_{\max}]$; for convenience,

we rescale X_i by defining $Y_i = \frac{1}{d_t r_{\max}} X_i$.

We now show concentration of the estimates via the following Chernoff bounds (see Theorem 1.1 in [18]):

1. $\mathbb{P}[|Y - \mathbb{E}[Y]| > \epsilon \mathbb{E}[Y]] < 2\exp(-\frac{\epsilon^2}{3}\mathbb{E}[Y])$
2. For any $b > 2e\mathbb{E}[Y], \mathbb{P}[Y > b] \leq 2^{-b}$

We perform a case analysis based on whether $\mathbb{E}[X_i] \geq \delta$ or $\mathbb{E}[X_i] < \delta$. First, if $\mathbb{E}[X_i] \geq \delta$, then we have $\mathbb{E}[Y] = \frac{w}{d_t r_{\max}}\mathbb{E}[X_i] = \frac{c}{\epsilon^2 \delta}\mathbb{E}[X_i] \geq \frac{c}{\epsilon^2}$, and thus:

$$\begin{aligned}\mathbb{P}\left[|\widehat{\pi}_s[t] - \pi_s[t]| > \epsilon \pi_s[t]\right] &\leq \mathbb{P}\left[\left|\bar{X} - \mathbb{E}[X_i]\right| > \epsilon \mathbb{E}[X_i]\right] = \mathbb{P}\left[|Y - \mathbb{E}[Y]| > \epsilon \mathbb{E}[Y]\right] \\ &\leq 2\exp\left(-\frac{\epsilon^2}{3}\mathbb{E}[Y]\right) \leq 2\exp\left(-\frac{c}{3}\right) \leq p_{\text{fail}},\end{aligned}$$

where the last line holds as long as we choose $c \geq 3\ln(2/p_{\text{fail}})$.

Suppose alternatively that $\mathbb{E}[X_i] < \delta$. Then:

$$\begin{aligned}\mathbb{P}[|\hat{\pi}_s[t] - \pi_s[t]| > 2e\delta] = \mathbb{P}[\left|\bar{X} - \mathbb{E}[X_i]\right| > 2e\delta] &= \mathbb{P}\left[|Y - \mathbb{E}[Y]| > \frac{w}{d_t r_{\max}} 2e\delta\right] \\ &\leq \mathbb{P}\left[Y > \frac{w}{d_t r_{\max}} 2e\delta\right].\end{aligned}$$

At this point we set $b = 2e\delta w / d_t r_{\max} = 2ec/\epsilon^2$ and apply the second Chernoff bound. Note that $\mathbb{E}[Y] = c\mathbb{E}[X_i]/\epsilon^2\delta < c/\epsilon^2$, and hence we satisfy $b > 2e\mathbb{E}[Y]$. We conclude that:

$$\mathbb{P}[|\hat{\pi}_s[t] - \pi_s[t]| > 2e\delta] \leq 2^{-b} \leq p_{\text{fail}}$$

as long as we choose c such that $c \geq \frac{\epsilon^2}{2e}\log_2 \frac{1}{p_{\text{fail}}}$. The proof is completed by combining both cases and choosing $c = 3\ln(2/p_{\text{fail}})$. □ □

Running Time Analysis

The more interesting analysis is that of the running-time of `UndirectedBiPPR` – we now prove a worst-case running-time bound:

Theorem 4. *In an undirected graph, for any source node (or distribution) s, target t with degree d_t, threshold δ, maximum residual r_{max}, relative error ϵ, and failure*

probability p_{fail}, ***UndirectedBiPPR*** *has a worst-case running-time of:*

$$O\left(\frac{\sqrt{\log \frac{1}{p_{fail}}}}{\alpha\epsilon}\sqrt{\frac{d_t}{\delta}}\right).$$

Before proving this result, we first state a crucial lemma from [2]:

Lemma 2 (Lemma 2 in [2])**.** *Let T be the total number of push operations performed by* ***ForwardPush****, and let d_k be the degree of the vertex involved in the k^{th} push. Then:*

$$\sum_{k=1}^{T} d_k \leq \frac{1}{\alpha r_{max}}$$

To keep this work self-contained, we also include a short proof from [2]

Proof. Let v_k be the vertex pushed in the k^{th} step – then by definition, we have that $r_s(v_k) > r_{\max} d_k$. Now after the local-push operation, the sum residual $||r_s||_1$ decreases by at least $\alpha r_{\max} d_k$. However, we started with $||r_s||_1 = 1$, and thus we have $\sum_{k=1}^{T} \alpha r_{\max} d_k \leq 1$. □ □

Note also that the amount of work done while pushing from a node v is d_v.

of Theorem 4. As proven in Lemma 2, the push forward step takes total time $O\left(1/\alpha r_{\max}\right)$ in the *worst-case*. The random walks take $O(w) = O\left(\frac{1}{\alpha\epsilon^2}\frac{r_{\max}}{\delta/d_t}\right)$ time. Thus our total time is

$$O\left(\frac{1}{\alpha r_{\max}} + \frac{\ln \frac{1}{p_{\text{fail}}}}{\alpha\epsilon^2}\frac{r_{\max}}{\delta/d_t}\right).$$

Balancing this by choosing $r_{\max} = \frac{\epsilon}{\sqrt{\ln \frac{1}{p_{\text{fail}}}}}\sqrt{\delta/d_t}$, we get total running-time:

$$O\left(\frac{\sqrt{\ln \frac{1}{p_{\text{fail}}}}}{\alpha\epsilon}\sqrt{\frac{d_t}{\delta}}\right). \quad \square$$

□

We can get a cleaner worst-case running time bound if we make a natural assumption on $\pi_s[t]$. In an undirected graph, if we let $\alpha = 0$ and take infinitely long walks, the stationary probability of being at any node t is $\frac{d_t}{2m}$. Thus if $\pi_s[t] < \frac{d_t}{2m}$, then s actually has a lower PPR to t than the non-personalized stationary probability of t, so it is natural to say t is not significant for s. If we set a significance threshold of $\delta = \frac{d_t}{2m}$, and apply the previous theorem, we immediately get the following:

Corollary 1. *If $\pi_s[t] \geq \frac{d_t}{2m}$, we can estimate $\pi_s[t]$ within relative error ϵ with constant probability in worst-case time:*

$$O\left(\frac{\sqrt{m}}{\alpha\epsilon}\right).$$

3.4 Experiments

3.4.1 PPR Estimation Running Time

We now compare `BidirectionalPPR` to its predecessor algorithms (namely `FAST-PPR` [39], `Monte Carlo` [5, 21] and `ReversePush` [1]). The experimental setup is identical to that in [39]; for convenience, we describe it here in brief. We perform experiments on 6 diverse, real-world networks: two directed social networks (Pokec (31M edges) and Twitter-2010 (1.5 billion edges)), two undirected social network (Live-Journal (69M edges) and Orkut (117M edges)), a collaboration network (dblp (6.7M edges)), and a web-graph (UK-2007-05 (3.7 billion edges)). Since all algorithms have parameters that enable a trade-off between running time and accuracy, we first choose parameters such that the mean relative error of each algorithm is approximately 10%. For bidirectional-PPR, we find that setting $c = 7$ (i.e., generating $7 \cdot \frac{r_{\max}}{\delta}$ random walks) results in a mean relative error less than 8% on all graphs; for the other algorithms, we use the settings determined in [39]. We then repeatedly sample a uniformly-random start node $s \in V$, and a random target $t \in T$ sampled either uniformly or from PageRank (to emphasize more important targets). For both `BidirectionalPPR` and `FAST-PPR`, we used the dynamic-balancing heuristic described above.

The results are shown in Figure 3.5. We find that `BidirectionalPPR` is at least 70x faster than `Monte Carlo` or `ReversePush` on all six graphs. For example, on

Orkut, past algorithms take 20 seconds per pair while `BidirectionalPPR` takes just 50 milliseconds. On Tiwtter-2010, past algorithms take 5 minutes, while `BidirectionalPPR` takes just 3 seconds

Comparing `BidirectionalPPR` to its predecessor `FAST-PPR`, we find it is 3 to 8 times faster than across all graphs. In particular, `BidirectionalPPR` only needs to sample $7\frac{r_{\max}}{\delta}$ random walks, while FAST-PPR needs $350\frac{r_{\max}}{\delta}$ walks to achieve the same mean relative error. This is because the estimator used by `BidirectionalPPR` is unbiased, while the estimator used in the basic `FAST-PPR` implementation uses a biased estimate provided by `Local Update` for nodes on the Frontier of the target.

3.4.2 PPR Estimation Running Time on Undirected Graphs

We present a preliminary comparison between the running time of `BidirectionalPPR` and `UndirectedBiPPR`. In this experiment, we find that `UndirectedBiPPR` is not significantly faster than `BidirectionalPPR`, but we report this for scientific completeness. To obtain a realistic, large, undirected graph, we compute personalized SALSA [7] on Twitter-2010, since personalized SALSA is equivalent to PPR on a transformed undirected graph (see Section 1.5 for details). We call the resulting graph Twitter-2010-SALSA. It has 1.5 billion undirected edges and 80 million nodes (two nodes for each node in Twitter-2010).

On this graph, we first compare the accuracy of `BidirectionalPPR` and `UndirectedBiPPR` using the following experiment. We run 100 trials. In each trial, we sample $s \in V$ uniformly and $t \in V$ from global PageRank (since users are more likely to search for popular targets). We check if $\pi_s[t] \geq \delta$ (setting $\delta = 1/n$) and if not repeatedly sample s and t in the same way until $\pi_s[t] \geq \delta$. We use the same constant $c = 7$ for both algorithms, and used the variations which balance forward and reverse running time. We find that the mean relative error for both algorithms is a little below 10%. This makes sense, since both algorithms are unbiased estimators and their accuracy comes from the same Chernoff Bound argument.

Next we compare the running time of the two algorithms. We run 1000 trials. As above, in each trial we sample s uniformly and t from global PageRank. We use

$\delta = 4/n$ (for consistency with the runtime experiments in [40]), Chernoff constant $c = 7$ (for determining the number of walks), and the balanced version of both algorithms, tuned such that the forward and reverse time are similar (within 20%). For both algorithms, the mean running time on Twitter-2010-SALSA is around 0.7 seconds per pair. This surprised us at first, since part of our motivation for developing `UndirectedBiPPR` was getting a worst-case running time bound. However, this result makes sense if we estimate the running time of `BidirectionalPPR` under an assumption on the in-degree of the nodes pushed from.

An Informal Parameterized Running Time Estimate

To understand the relationship between the running time of `BidirectionalPPR` and `UndirectedBiPPR` we find it useful to analyze the running time dependence of `BidirectionalPPR` on $\pi[t]$ under the heuristic assumption that the set of nodes we push back from have average in-degree. The analysis in this section is similar to the rigorous running time analysis of `UndirectedBiPPR` in Section 3.3.4.

Theorem 1 of [1] states that `ReversePush`$(t, r_{\max}, \alpha)$ performs at most $\frac{n\pi[t]}{\alpha r_{\max}}$ push-back operations, and the exact running time is proportional to the sum of the in-degrees of all the nodes we push back from. If we heuristically assume the average in-degree of these nodes is the average degree of the graph, $\overline{d} = m/n$, then the running time of `ReversePush`$(t, r_{\max}, \alpha)$ is $O\left(\overline{d}\frac{n\pi[t]}{\alpha r_{\max}}\right)$ and the total running time for `BidirectionalPPR` is

$$O\left(\frac{c_1 r_{\max}}{\alpha\epsilon^2\delta} + \overline{d}\frac{n\pi[t]}{\alpha r_{\max}}\right).$$

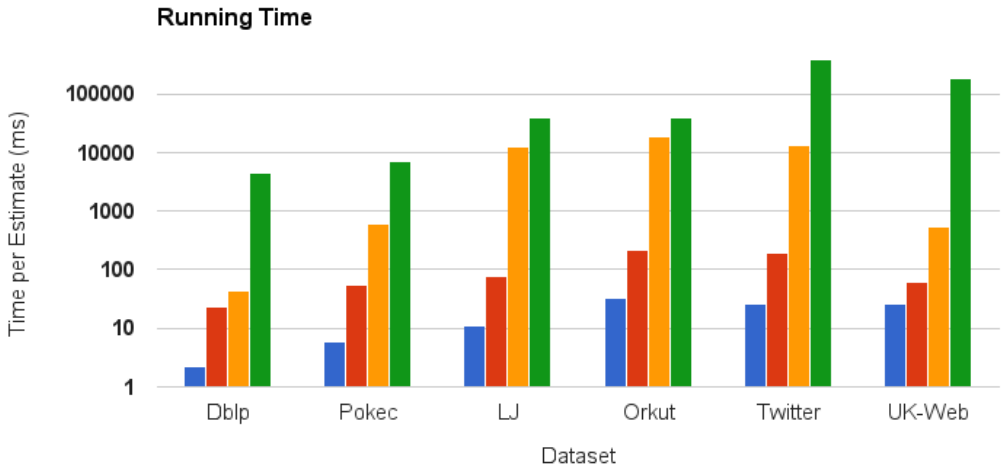
Now if we set $r_{\max} = \sqrt{\frac{n\pi[t]\overline{d}}{c_1\epsilon^2\delta}}$ to minimize this expression, we get running time

$$O\left(\frac{1}{\alpha\epsilon}\sqrt{\frac{cm\pi[t]}{\delta}}\right).$$

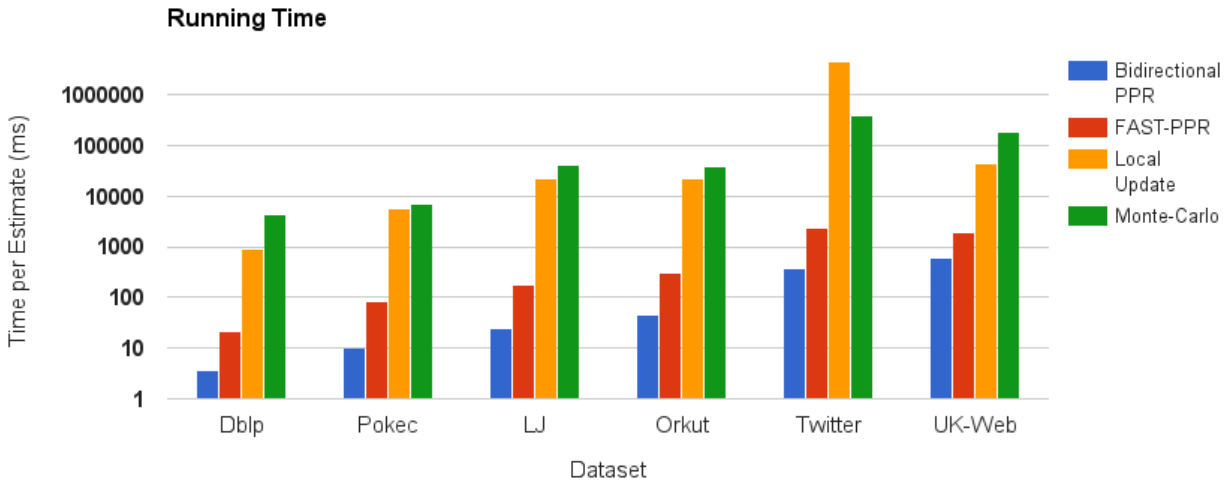
Similarly, in Section 3.3.4 we show the the parameterized running time of `UndirectedBiPPR` is

$$O\left(\frac{1}{\alpha\epsilon}\sqrt{\frac{cd_t}{\delta}}\right).$$

On an undirected graph, the PageRank $\pi[t]$ tends to $\frac{d_t}{2m}$ as $\alpha \to 0$, and we expect $\pi[t]$ to approximate $\frac{d_t}{2m}$ even for positive α. If we replace d_t in the second equation with $\overline{d}n\pi[t]$, then we actually get identical running times. This explains the similar empirical running times we observe.



(a) Sampling targets uniformly



(b) Sampling targets from PageRank distribution

Figure 3.5: Average running-time (on log-scale) for different networks. We measure the time required for estimating PPR values $\pi_s[t]$ with threshold $\delta = \frac{4}{n}$ for 1000 (s, t) pairs. For each pair, the start node is sampled uniformly, while the target node is sampled uniformly in Figure 3.5(a), or from the global PageRank distribution in Figure 3.5(b). In this plot we use teleport probability $\alpha = 0.2$.

Chapter 4

Random Walk Probability Estimation

This chapter generalizes our results to any estimating random walk probabilities where the walks have a given length, and hence to arbitrary Markov Chains [9].

4.1 Overview

We present a new bidirectional algorithm for estimating fixed length transition probabilities: given a weighted, directed graph (Markov chain), we want to estimate the probability of hitting a given target state in ℓ steps after starting from a given source distribution. Given the target state t, we use a (reverse) local power iteration to construct an 'expanded target distribution', which has the same mean as the quantity we want to estimate, but a smaller variance – this can then be sampled efficiently by a Monte Carlo algorithm. Our method extends to any Markov chain on a discrete (finite or countable) state-space, and can be extended to compute functions of multi-step transition probabilities such as PageRank, graph diffusions, hitting/return times, etc. Our main result is that in 'sparse' Markov Chains – wherein the number of transitions between states is comparable to the number of states – the running time of our algorithm for a uniform-random target node is *order-wise smaller* than Monte Carlo and power iteration based algorithms. In particular, while Monte Carlo takes time

$O\left(\frac{\ell}{\epsilon^2 p}\right)$ to estimate a probability p within relative error ϵ with constant probability on sparse Markov Chains, our algorithm's running time is only $O\left(\frac{\ell^{3/2}}{\epsilon}\sqrt{p}\right)$.

4.2 Problem Description

Markov chains are one of the workhorses of stochastic modeling, finding use across a variety of applications – MCMC algorithms for simulation and statistical inference; to compute network centrality metrics for data mining applications; statistical physics; operations management models for reliability, inventory and supply chains, etc. In this paper, we consider a fundamental problem associated with Markov chains, which we refer to as the *multi-step transition probability estimation* (or MSTP-estimation) problem: given a Markov Chain on state space V with transition matrix W, an initial source distribution s over V, a target state $t \in V$ and a fixed length ℓ, we are interested in computing the *ℓ-step transition probability from s to t*. Formally, we want to estimate:

$$\pi_s^\ell[t] := \langle sW^\ell, \underline{\mathbf{e}}_t \rangle = sW^\ell \underline{\mathbf{e}}_t^T, \tag{4.1}$$

where $\underline{\mathbf{e}}_t$ is the indicator vector of state t. A natural parametrization for the complexity of MSTP-estimation is in terms of the minimum transition probabilities we want to detect: given a desired minimum detection threshold δ, we want algorithms that give estimates which guarantee small relative error for any (s, t, ℓ) such that $\pi_s^\ell[t] > \delta$.

Parametrizing in terms of the minimum detection threshold δ can be thought of as benchmarking against a standard Monte Carlo algorithm, which estimates $\pi_s^\ell[t]$ by sampling independent ℓ-step paths starting from states sampled from s. An alternate technique for MSTP-estimation is based on linear algebraic iterations, in particular, the (local) power iteration. We discuss these in more detail in Section 4.2.2. Crucially, however, *both these techniques have a running time of* $\Omega(1/\delta)$ for testing if $\pi_s^\ell[t] > \delta$ (cf. Section 4.2.2).

4.2.1 Our Results

To the best of our knowledge, our work gives *the first bidirectional algorithm for MSTP-estimation* which works for general discrete state-space Markov chains[1]. The algorithm we develop is very simple, both in terms of implementation and analysis. Furthermore, we prove that in many settings, it is order-wise faster than existing techniques.

Our algorithm consists of two distinct *forward* and *reverse* components, which are executed sequentially. In brief, the two components proceed as follows:

- **Reverse-work**: Starting from the target node t, we perform a sequence of reverse local power iterations – in particular, we use the REVERSE-PUSH operation defined in Algorithm 7.
- **Forward-work**: We next sample a number of random walks of length ℓ, starting from s and transitioning according to P, and return the sum of residues on the walk as an estimate of $\pi_s^\ell[t]$.

This full algorithm, which we refer to as the `Bidirectional-MSTP` estimator, is formalized in Algorithm 8. It works for all countable-state Markov chains, giving the following accuracy result:

Theorem 5 (For details, see Section 4.3.3)**.** *Given any Markov chain W, source distribution s, terminal state t, length ℓ, threshold δ and relative error ϵ, `Bidirectional-MSTP` (Algorithm 8) returns an unbiased estimate $\widehat{\pi}_s^\ell[t]$ for $\pi_s^\ell[t]$, which, with high probability, satisfies:*

$$\left|\widehat{\pi}_s^\ell[t] - \pi_s^\ell[t]\right| < \max\left\{\epsilon\pi_s^\ell[t], \delta\right\}.$$

Since we dynamically adjust the number of REVERSE-PUSH operations to ensure that all residues are small, the proof of the above theorem follows from straightforward concentration bounds.

[1]Bidirectional estimators have been developed before for *reversible* Markov chains [23]; our method however is not only more general, but conceptually and operationally simpler than these techniques (cf. Section 4.2.2).

Since `Bidirectional-MSTP` combines local power iteration and Monte Carlo techniques, a natural question is when the algorithm is faster than both. It is easy to to construct scenarios where the runtime of `Bidirectional-MSTP` is comparable to its two constituent algorithms – for example, if t has more than $1/\delta$ in-neighbors. Surprisingly, however, we show that in *sparse Markov chains* and for *random target states*, `Bidirectional-MSTP` is order-wise faster:

Theorem 6 (For details, see Section 4.3.3)**.** *Given any Markov chain W, source distribution s, length ℓ, threshold δ and desired accuracy ϵ; then for a uniform random choice of $t \in V$, the* ***`Bidirectional-MSTP`*** *algorithm has a running time of $\widetilde{O}(\ell^{3/2}\sqrt{\overline{d}/\delta})$, where $\overline{d}$ is the average number of neighbors of nodes in V.*

Thus, for random targets, *our running time dependence on δ is* $1/\sqrt{\delta}$. Note that we do not need for every state that the number of neighboring states is small, but rather, that they are small on average – for example, this is true in 'power-law' networks, where some nodes have very high degree, but the average degree is small. The proof of this result in Section 4.3.3 is similar to the analysis of `BidirectionalPPR` in Section 3.2.3.

Estimating transition probabilities to a target state is one of the fundamental primitives in Markov chain models – hence, we believe that our algorithm can prove useful in a variety of application domains. In Section 4.4, we briefly describe how to adapt our method for some of these applications – estimating hitting/return times and stationary probabilities, extensions to non-homogenous Markov chains (in particular, for estimating graph diffusions and heat kernels), connections to local algorithms and expansion testing. In addition, our MSTP-estimator could be useful in several other applications – estimating ruin probabilities in reliability models, buffer overflows in queueing systems, in statistical physics simulations, etc.

4.2.2 Existing Approaches for MSTP-Estimation

There are two main techniques used for MSTP-estimation, similar to the techniques described in Chapter 2 for PPR estimation. The first is a natural Monte Carlo algorithm (see Section 2.5): we estimate $\pi_s^\ell[t]$ by sampling independent ℓ-step paths,

each starting from a random state sampled from s. A simple concentration argument shows that for a given value of δ, we need $\widetilde{\Theta}(1/\delta)$ samples to get an accurate estimate of $\pi_s^\ell[t]$, irrespective of the choice of t, and the structure of W. Note that this algorithm is agnostic of the terminal state t; it gives an accurate estimate for any t such that $\pi_s^\ell[t] > \delta$.

On the other hand, the problem also admits a natural linear algebraic solution, using the standard power iteration (see Section 2.2) starting with s, or the reverse power iteration starting with $\underline{\mathbf{e}}_t$ which is obtained by re-writing Equation (4.1) as $\pi_s^\ell[t] := s(\underline{\mathbf{e}}_t(W^T)^\ell)^T$. When the state space is large, performing a direct power iteration is infeasible – however, there are localized versions of the power iteration that are still efficient. Such algorithms have been developed, among other applications, for PageRank estimation (see Section 2.4 and 2.3) and for heat kernel estimation [31]. Although slow in the worst case [2], such local update algorithms are often fast in practice, as unlike Monte Carlo methods they exploit the local structure of the chain. However even in sparse Markov chains and for a large fraction of target states, their running time can be $\Omega(1/\delta)$. For example, consider a random walk on a random d-regular graph and let $\delta = o(1/n)$ – then for $\ell \sim \log_d(1/\delta)$, verifying $\pi_{\underline{\mathbf{e}}_s}^\ell[t] > \delta$ is equivalent to uncovering the entire $\log_d(1/\delta)$ neighborhood of s. Since a large random d-regular graph is (whp) an expander, this neighborhood has $\Omega(1/\delta)$ distinct nodes. Finally, note that as with Monte Carlo, power iterations do not only return probabilities between a single pair, but return probabilities from a single source to all targets, or from all sources to a single target.

For *reversible Markov chains*, one can get a bidirectional algorithms for estimating $\pi_{\underline{\mathbf{e}}_s}^\ell[t]$ based on colliding random walks. For example, consider the problem of estimating length-2ℓ random walk transition probabilities in a *regular undirected graph* $G(V, E)$ on n vertices [23, 29]. The main idea is that to test if a random walk goes from s to t in 2ℓ steps with probability $\geq \delta$, we can generate two independent random walks of length ℓ, starting from s and t respectively, and detect if they *terminate at the same intermediate node*. Suppose p_w, q_w are the probabilities that a length-ℓ walk

[2] In particular, local power iterations are slow if a state has a very large out-neighborhood (for the forward iteration) or in-neighborhood (for the reverse update).

from s and t respectively terminate at node w – then from the reversibility of the chain, we have that $\pi_s^{2\ell}[t] = \sum_{w \in V} p_w q_w$; this is also the collision probability. The critical observation is that if we generate $\sqrt{1/\delta}$ walks from s and t, then we get $1/\delta$ potential collisions, which is sufficient to detect if $\pi_s^{2\ell}[t] > \delta$. This argument forms the basis of the *birthday-paradox*, and similar techniques used in a variety of estimation problems (eg., see [43]). Showing concentration for this estimator is tricky as the samples are not independent; moreover, to control the variance of the samples, the algorithms often need to separately deal with 'heavy' intermediate nodes, where p_w or q_w are much larger than $O(1/n)$. Our proposed approach is much simpler both in terms of algorithm and analysis, and more significantly, it extends beyond reversible chains to any general discrete state-space Markov chain.

4.3 The Bidirectional MSTP-estimation Algorithm

Our approach is similar to our approach for bidirectional PPR estimation (Section 3.2). The main difference is that for PPR, the random walk is memoryless, so the `ReversePush` algorithm returns a single residual value $r^t[v]$ for each node. Here, we have a fixed walk length, so we a compute separate residual value $r_t^k[v]$ for each number of steps $k \leq \ell$ back from the target. We then combine residual values k steps back from the target with a random walk probability $\ell - k$ steps forward from the source, and sum over $k \leq \ell$ to estimate $\pi_s^\ell[t]$.

4.3.1 Algorithm

As described in Section 4.2.1, given a target state t, our bidirectional MSTP algorithm keeps track of a pair of vectors – the estimate vector $\mathbf{p}_t^k \in \mathbb{R}^n$ and the residual vector $\mathbf{r}_t^k \in \mathbb{R}^n$ – for each length $k \in \{0, 1, 2, \ldots, \ell\}$. The vectors are initialized all set to $\underline{0}$ (i.e., the all-0 vector), except r_t^0 which is initialized as $\underline{\mathbf{e}}_t$. They are updated using a *reverse push* operation defined as Algorithm 7.

The main observation behind our algorithm is that we can re-write $\pi_s^\ell[t]$ in terms of $\{\mathbf{p}_t^k, \mathbf{r}_t^k\}$ as an *expectation over random sample-paths of the Markov chain* as follows

Algorithm 7 REVERSE-PUSH-MSTP(v, i)

Inputs: Transition matrix W, estimate vector $\mathbf{p}_t^i$, residual vectors $\mathbf{r}_t^i, \mathbf{r}_t^{i+1}$

1: **return** New estimate vectors $\{\widetilde{\mathbf{p}}_t^i\}$ and residual-vectors $\{\widetilde{\mathbf{r}}_t^i\}$ computed as:

$$\begin{aligned}\widetilde{\mathbf{p}}_t^i &\leftarrow \mathbf{p}_t^i + \langle \mathbf{r}_t^i, \underline{\mathbf{e}}_v \rangle \underline{\mathbf{e}}_v \\ \widetilde{\mathbf{r}}_t^i &\leftarrow \mathbf{r}_t^i - \langle \mathbf{r}_t^i, \underline{\mathbf{e}}_v \rangle \underline{\mathbf{e}}_v \\ \widetilde{\mathbf{r}}_t^{i+1} &\leftarrow \mathbf{r}_t^{i+1} + \langle \mathbf{r}_t^i, \underline{\mathbf{e}}_v \rangle \left(\underline{\mathbf{e}}_v W^T \right)\end{aligned}$$

(cf. Equation (4.3)):

$$\pi_s^\ell[t] = \langle s, \mathbf{p}_t^\ell \rangle + \sum_{k=0}^{\ell} \mathbb{E}_{V_k \sim sW^k} \left[\mathbf{r}_t^{\ell-k}(V_k) \right] \tag{4.2}$$

In other words, given vectors $\{\mathbf{p}_t^k, \mathbf{r}_t^k\}$, we can get an unbiased estimator for $\pi_s^\ell[t]$ by sampling a length-ℓ random trajectory $\{V_0, V_1, \ldots, V_\ell\}$ of the Markov chain W starting at a random state V_0 sampled from the source distribution s, and then adding the residuals along the trajectory as in Equation (4.2). We formalize this bidirectional MSTP algorithm in Algorithm 8.

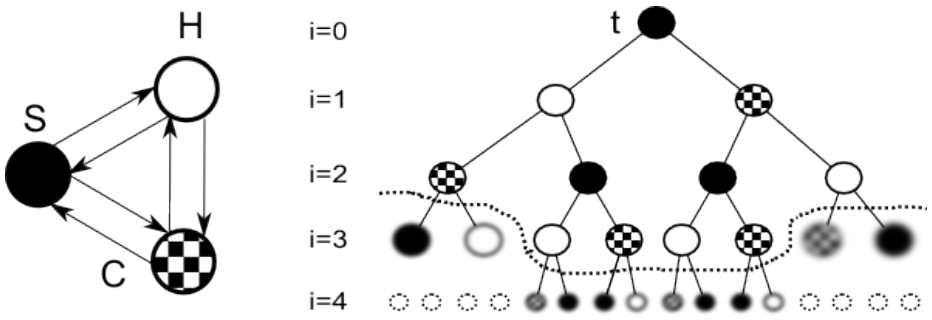


Figure 4.1: Visualizing a sequence of REVERSE-PUSH-MSTP operations: Given the Markov chain on the left with S as the target, we perform REVERSE-PUSH-MSTP operations $(S, 0), (H, 1), (C, 1), (S, 2)$.

Algorithm 8 Bidirectional-MSTP($W, s, t, \ell_{\max}, \delta$)

Inputs: Transition matrix W, source distribution s, target state t, maximum steps $\ell_{\max}$, minimum probability threshold δ, relative error bound ϵ, failure probability p_f

Set accuracy parameter c based on ϵ and p_f and set reverse threshold $r_{\max}$ (cf. Theorems 5 and 6) (in our experiments we use $c = 7$ and $r_{\max} = \sqrt{\delta/c}$)
Initialize: Estimate vectors $\mathbf{p}_t^k = \underline{0}\,,\ \forall\, k \in \{0, 1, 2, \ldots, \ell\}$,
Residual vectors $\mathbf{r}_t^0 = \underline{\mathbf{e}}_t$ and $\mathbf{r}_t^k = \underline{0}\,,\ \forall\, k \in \{1, 2, 3, \ldots, \ell\}$
for $i \in \{0, 1, \ldots, \ell_{\max}\}$ **do**
  while $\exists\, v \in V \quad s.t. \quad \mathbf{r}_t^i[v] > r_{\max}$ **do**
    Execute REVERSE-PUSH-MSTP(v, i)
  end while
end for
Set number of sample paths $n_f = c\ell_{\max} r_{\max}/\delta$ (See Theorem 5 for details)
for index $i \in \{1, 2, \ldots, n_f\}$ **do**
  Sample starting node $V_i^0 \sim s$
  Generate sample path $T_i = \{V_i^0, V_i^1, \ldots, V_i^{\ell_{\max}}\}$ of length $\ell_{\max}$ starting from V_i^0
  For $\ell \in \{1, 2, \ldots, \ell_{\max}\}$: sample $k \sim Uniform[0, \ell]$ and compute $S_{t,i}^{\ell} = \ell \mathbf{r}_t^{\ell-k}[V_i^k]$
  (We reinterpret the sum over k in Equation 4.2 as an expectation and sample k rather sum over $k \leq \ell$ for computational speed.)
end for
return $\{\widehat{\pi}_s^{\ell}[t]\}_{\ell \in [\ell_{\max}]}$, where $\widehat{\pi}_s^{\ell}[t] = \langle s, \mathbf{p}_t^{\ell} \rangle + (1/n_f) \sum_{i=1}^{n_f} S_{t,i}^{\ell}$

4.3.2 Some Intuition Behind our Approach

Before formally analyzing the performance of our MSTP-estimation algorithm, we first build some intuition as to why it works. In particular, we give an interpretation for the residual values $r_t^k[v]$ which is different from the message passing interpretation of the PPR residual values given in Section 2.3. In Figure 4.1, we have considered a simple Markov chain on three states – Solid, Hollow and Checkered (henceforth (S, H, C)). On the right side, we have illustrated an intermediate stage of reverse work using S as the target, after performing the REVERSE-PUSH-MSTP operations $(S, 0), (H, 1), (C, 1)$ and $(S, 2)$ in that order. Each push at level i uncovers a collection of length-$(i+1)$ paths terminating at S – for example, in the figure, we have uncovered

all length 2 and 3 paths, and several length 4 paths. The crucial observation is that *each uncovered path of length i starting from a node v is accounted for in either* $\mathbf{p}_v^i$ *or* $\mathbf{r}_v^i$. In particular, in Figure 4.1, all paths starting at solid nodes are stored in the estimates of the corresponding states, while those starting at blurred nodes are stored in the residue. Now we can use this set of pre-discovered paths to boost the estimate returned by Monte Carlo trajectories generated starting from the source distribution. The dotted line in the figure represents the current reverse-work *frontier* – it separates the fully uncovered neighborhood of $(S, 0)$ from the remaining states (v, i).

4.3.3 Performance Analysis

We first formalize the critical invariant introduced in Equation (4.2):

Lemma 3. *Given a terminal state t, suppose we initialize $\mathbf{p}_t^0 = \underline{0}, \mathbf{r}_t^0 = \underline{\mathbf{e}}_t$ and $\mathbf{p}_t^k, \mathbf{r}_t^k = \underline{0} \,\forall\, k \geq 0$. Then for any source distribution s and length ℓ, after any arbitrary sequence of REVERSE-PUSH-MSTP(v, k) operations, the vectors $\{\mathbf{p}_t^k, \mathbf{r}_t^k\}$ satisfy the invariant:*

$$\pi_s^\ell[t] = \langle s, \mathbf{p}_t^\ell \rangle + \sum_{k=0}^{\ell} \langle sW^k, \mathbf{r}_t^{\ell-k} \rangle \tag{4.3}$$

Proof. The proof follows the outline of a similar result in Andersen et al. [1] for PageRank estimation. First, note that Equation (4.3) can be re-written as $\pi_s^\ell[t] = \left\langle s, \mathbf{p}_t^\ell + \sum_{k=0}^{\ell} \mathbf{r}_t^{\ell-k}(W^k)^T \right\rangle$. Note that under the initial conditions specified in Algorithm 8, for any s, ℓ, the invariant reduces to: $\pi_s^\ell[t] = \left\langle s, \mathbf{p}_t^\ell + \sum_{k=0}^{\ell} \mathbf{r}_t^{\ell-k}(W^k)^T \right\rangle = \langle s, \underline{\mathbf{e}}_t (W^T)^\ell \rangle = \langle sW^\ell, \underline{\mathbf{e}}_t \rangle$, which is true by definition. We now prove the invariant by induction. Suppose at some stage, vectors $\{\mathbf{p}_t^k, \mathbf{r}_t^k\}$ satisfy Equation (4.3); to complete the proof, we need to show that for any pair (v, i), the REVERSE-PUSH-MSTP(v, i) operation preserves the invariant. Let $\{\widetilde{\mathbf{p}}_t^k, \widetilde{\mathbf{r}}_t^k\}$ denote the estimate and residual vectors after the REVERSE-PUSH-MSTP(v, i) operation is applied, and define:

$$\Delta_v^i = \left(\widetilde{\mathbf{p}}_t^\ell + \sum_{k=0}^{\ell} \widetilde{\mathbf{r}}_t^{\ell-k}(W^k)^T \right) - \left(\mathbf{p}_t^\ell + \sum_{k=0}^{\ell} \mathbf{r}_t^{\ell-k}(W^k)^T \right)$$

Now, to prove the invariant, it is sufficient to show $\Delta_v^i = 0$ for any choice of (v, i). Clearly this is true if $\ell < i$; this leaves us with two cases:

- If $\ell = i$ we have: $\Delta_v^i = \left(\widetilde{\mathbf{p}}_t^{\,i} - \mathbf{p}_t^i\right) + \left(\widetilde{\mathbf{r}}_t^i - \mathbf{r}_t^i\right) = \langle \mathbf{r}_t^k, \underline{\mathbf{e}}_v \rangle \underline{\mathbf{e}}_v - \langle \mathbf{r}_t^k, \underline{\mathbf{e}}_v \rangle \underline{\mathbf{e}}_v = 0$
- If $\ell > i$, we have:

$$\begin{aligned}\Delta_v^i &= \left(\widetilde{\mathbf{r}}_t^{i+1} - \mathbf{r}_t^{i+1} + (\widetilde{\mathbf{r}}_t^i - \mathbf{r}_t^i) W^T\right)\left(W^T\right)^{\ell-i-1} \\ &= \left(\langle \mathbf{r}_t^k, \underline{\mathbf{e}}_v \rangle (\underline{\mathbf{e}}_t W^T) - (\langle \mathbf{r}_t^k, \underline{\mathbf{e}}_v \rangle \underline{\mathbf{e}}_v) W^T\right)\left(W^T\right)^{\ell-i-1} = \underline{0}\end{aligned}$$

This completes the proof of the lemma. □

Using this result, we can now characterize the accuracy of the `Bidirectional-MSTP` algorithm:

Theorem 5. *We are given any Markov chain W, source distribution s, terminal state t, maximum length $\ell_{\max}$ and also parameters δ, p_f and ϵ (i.e., the desired threshold, failure probability and relative error). Suppose we choose any reverse threshold $r_{max} > \delta$, and set the number of sample-paths $n_f = cr_{max}/\delta$, where $c = \max\{6e/\epsilon^2, 1/\ln 2\} \ln(2\ell_{\max}/p_f)$. Then for any length $\ell \leq \ell_{\max}$ with probability at least $1 - p_f$, the estimate returned by `Bidirectional-MSTP` satisfies:*

$$\left|\widehat{\pi}_s^\ell[t] - \pi_s^\ell[t]\right| < \max\left\{\epsilon \pi_s^\ell[t], \delta\right\}.$$

Proof. Given any Markov chain W and terminal state t, note first that for a given length $\ell \leq \ell_{\max}$, Equation (4.2) shows that the estimate $\widehat{\pi}_s^\ell[t]$ is an unbiased estimator. Now, for any random-trajectory T_k, we have that the score $S_{t,k}^\ell$ obeys: (i) $\mathbb{E}[S_{t,k}^\ell] \leq \pi_s^\ell[t]$ and (ii) $S_{t,k}^\ell \in [0, \ell r_{\max}]$; the first inequality again follows from Equation (4.2), while the second follows from the fact that we executed REVERSE-PUSH-MSTP operations until all residual values were less than $r_{\max}$.

Now consider the rescaled random variable $X_k = S_{t,k}^\ell/(\ell r_{\max})$ and $X = \sum_{k \in [n_f]} X_k$; then we have that $X_k \in [0, 1]$, $\mathbb{E}[X] \leq (n_f/\ell r_{\max}) \pi_s^\ell[t]$ and also $(X - \mathbb{E}[X]) = (n_f/\ell r_{\max})(\widehat{\pi}_s^\ell[t] - \pi_s^\ell[t])$. Moreover, using standard Chernoff bounds (cf. Theorem

1.1 in [19]), we have that:

$$\mathbb{P}\left[|X - \mathbb{E}[X]| > \epsilon\mathbb{E}[X]\right] < 2\exp\left(-\frac{\epsilon^2\mathbb{E}[X]}{3}\right) \quad \text{and} \quad \mathbb{P}[X > b] \leq 2^{-b} \text{ for any } b > 2e\mathbb{E}[X]$$

Now we consider two cases:

1. $\mathbb{E}[S^\ell_{t,k}] > \delta/2e$ (i.e., $\mathbb{E}[X] > n_f\delta/2e\ell r_{\max} = c/2e$): Here, we can use the first concentration bound to get:

$$\begin{aligned}\mathbb{P}\left[\left|\widehat{\pi}^\ell_s[t] - \pi^\ell_s[t]\right| \geq \epsilon\pi^\ell_s[t]\right] = \mathbb{P}\left[|X - \mathbb{E}[X]| \geq \frac{\epsilon n_f}{\ell r_{\max}}\pi^\ell_s[t]\right] &\leq \mathbb{P}\left[|X - \mathbb{E}[X]| \geq \epsilon\mathbb{E}[X]\right] \\ &\leq 2\exp\left(-\frac{\epsilon^2\mathbb{E}[X]}{3}\right) \leq 2\exp\left(-\frac{\epsilon^2 c}{6e}\right),\end{aligned}$$

   where we use that $n_f = c\ell_{\max}r_{\max}/\delta$ (cf. Algorithm 8). Moreover, by the union bound, we have:

$$\mathbb{P}\left[\bigcup_{\ell\leq\ell_{\max}}\left\{\left|\widehat{\pi}^\ell_s[t] - \pi^\ell_s[t]\right| \geq \epsilon\pi^\ell_s[t]\right\}\right] \leq 2\ell_{\max}\exp\left(-\frac{\epsilon^2 c}{32e}\right),$$

   Now as long as $c \geq (6e/\epsilon^2)\ln(2\ell_{\max}/p_f)$, we get the desired failure probability.
2. $\mathbb{E}[S^\ell_{t,k}] < \delta/2e$ (i.e., $\mathbb{E}[X] < c/2e$): In this case, note first that since $X > 0$, we have that $\pi^\ell_s[t] - \widehat{\pi}^\ell_s[t] \leq (n_f/\ell r_{\max})\mathbb{E}[X] \leq \delta/2e < \delta$. On the other hand, we also have:

$$\mathbb{P}\left[\widehat{\pi}^\ell_s[t] - \pi^\ell_s[t] \geq \delta\right] = \mathbb{P}\left[X - \mathbb{E}[X] \geq \frac{n_f\delta}{\ell r_{\max}}\right] \leq \mathbb{P}\left[X \geq c\right] \leq 2^{-c},$$

   where the last inequality follows from our second concentration bound, which holds since we have $c > 2e\mathbb{E}[X]$. Now as before, we can use the union bound to show that the failure probability is bounded by p_f as long as $c \geq \log_2(\ell_{\max}/p_f)$.

Combining the two cases, we see that as long as $c \geq \max\{6e/\epsilon^2, 1/\ln 2\}\ln(2\ell_{\max}/p_f)$, then we have $\mathbb{P}\left[\bigcup_{\ell\leq\ell_{\max}}\left\{\left|\widehat{\pi}^\ell_s[t] - \pi^\ell_s[t]\right| \geq \max\{\delta, \epsilon\pi^\ell_s[t]\}\right\}\right] \leq p_f$. □

One aspect that is not obvious from the intuition in Section 4.3.2 or the accuracy analysis is if using a bidirectional method actually improves the running time of

MSTP-estimation. This is addressed by the following result, which shows that for typical targets, our algorithm achieves significant speedup:

Theorem 6. *Let any Markov chain W, source distribution s, maximum length $\ell_{\max}$, minimum probability δ, failure probability p_f, and relative error bound ϵ be given. Suppose we set $r_{max} = \sqrt{\frac{\epsilon^2\delta}{\ell_{\max}\log(\ell_{\max}/p_f)}}$. Then for a uniform random choice of $t \in V$, the* `Bidirectional-MSTP` *algorithm has a running time*

$$O\left(\frac{\ell_{\max}^{3/2}\sqrt{\overline{d}\log(\ell_{\max}/p_f)}}{\epsilon}\frac{1}{\sqrt{\delta}}\right).$$

Proof. The runtime of Algorithm 8 consists of two parts:
Forward-work (i.e., for generating trajectories): we generate $n_f = c\ell_{\max}r_{\max}/\delta$ sample trajectories, each of length $\ell_{\max}$ – hence the running time is $O\left(c\delta\ell_{\max}^2/\delta\right)$ for any Markov chain W, source distribution s and target node t. Substituting for c from Theorem 5, we get that the forward-work running time $T_f = O\left(\frac{\ell_{\max}^2 r_{\max}\log(\ell_{\max}/p_f)}{\epsilon^2\delta}\right)$.
Reverse-work (i.e., for REVERSE-PUSH-MSTP operations): Let T_r denote the reverse-work runtime for a *uniform random choice of* $t \in V$. Then we have:

$$\mathbb{E}[T_r] = \frac{1}{|V|}\sum_{t\in V}\sum_{k=0}^{\ell_{\max}}\sum_{v\in V}(d^{in}(v)+1)\mathbb{1}\{\text{REVERSE-PUSH-MSTP}_{(v,k)} \text{ is executed}\}$$

Now for a given $t \in V$ and $k \in \{0, 1, \ldots, \ell_{\max}\}$, note that the only states $v \in V$ on which we execute REVERSE-PUSH-MSTP(v, k) are those with residual $\mathbf{r}_t^k(v) > r_{\max}$ – consequently, for these states, we have that $\mathbf{p}_t^k(v) > r_{\max}$, and hence, by Equation (4.3), we have that $\pi^k_{\underline{\mathbf{e}}_v}[t] \geq r_{\max}$ (by setting $s = \underline{\mathbf{e}}_v$, i.e., starting from state v). Moreover, a REVERSE-PUSH-MSTP(v, k) operation involves updating the residuals for $d^{in}(v) + 1$ states. Note that $\sum_{t\in V}\pi^k_{\underline{\mathbf{e}}_v}[t] = 1$ and hence, via a straightforward counting argument, we have that for any $v \in V$, $\sum_{t\in V}\mathbb{1}_{\{\pi^k_{\underline{\mathbf{e}}_v}[t]\geq r_{\max}\}} \leq 1/r_{\max}$. Thus,

we have:

$$\mathbb{E}[T_r] \leq \frac{1}{|V|}\sum_{t\in V}\sum_{k=0}^{\ell_{\max}}\sum_{v\in V}(d^{in}(v)+1)\mathbb{1}_{\{\pi^k_{\underline{\mathbf{e}}_v}[t]\geq r_{\max}\}} = \frac{1}{|V|}\sum_{v\in V}\sum_{k=0}^{\ell_{\max}}\sum_{t\in V}(d^{in}(v)+1)\mathbb{1}_{\{\pi^k_{\underline{\mathbf{e}}_v}[t]\geq r_{\max}\}}$$
$$\leq \frac{1}{|V|}\sum_{v\in V}(\ell_{\max}+1)\cdot(d^{in}(v)+1)\frac{1}{r_{\max}} = O\left(\frac{\ell_{\max}}{r_{\max}}\cdot\frac{\sum_{v\in V}d^{in}(v)}{|V|}\right) = O\left(\frac{\ell_{\max}\overline{d}}{r_{\max}}\right)$$

Finally, we choose $r_{\max} = \sqrt{\frac{\epsilon^2\delta}{\ell_{\max}\log(\ell_{\max}/p_f)}}$ to balance T_f and T_r and get the result. □

4.4 Applications of MSTP estimation

- **Estimating the Stationary Distribution and Hitting Probabilities:** MSTP-estimation can be used in two ways to estimate stationary probabilities $\pi[t]$. First, if we know the mixing time τ_{mix} of the chain W, we can directly use Algorithm 8 to approximate $\pi[t]$ by setting $\ell_{\max} = \tau_{mix}$ and using any source distribution s. Theorem 6 then guarantees that we can estimate a stationary probability of order δ in time $O(\tau_{mix}^{3/2}\sqrt{\overline{d}/\delta})$. In comparison, Monte Carlo has $O(\tau_{mix}/\delta)$ runtime. We note that in practice, we usually do not know the mixing time – in such a setting, our algorithm can be used to compute an estimate of $\pi_s^\ell[t]$ for all values of $\ell \leq \ell_{\max}$.

  An alternative is to modify Algorithm 8 to estimate the *truncated hitting time* $\widehat{p}_s^{\ell,hit}[t]$(i.e., the probability of hitting t starting from s for the first time in ℓ steps). By setting $s = \underline{\mathbf{e}}_t$, we get an estimate for the expected *truncated return time* $\mathbb{E}[T_t\mathbb{1}_{\{T_t\leq\ell_{\max}\}}] = \sum_{\ell\leq\ell_{\max}}\ell\widehat{p}^{\ell,hit}_{\underline{\mathbf{e}}_t}[t]$ where T_t is the hitting time to target t. Now, using that fact that $\pi[t] = 1/\mathbb{E}[T_t]$, we can get a lower bound for $\pi[t]$ which converges to $\pi[t]$ as $\ell_{\max}\to\infty$. We note also that the truncated hitting time has been shown to be useful in other applications such as identifying similar documents on a document-word-author graph [46].

  To estimate the truncated hitting time, we modify Algorithm 8 as follows: at each stage $i \in \{1,2,\ldots,\ell_{\max}\}$ (note: not $i=0$), instead of REVERSE-PUSH-MSTP(t,i), we update $\widetilde{\mathbf{p}}_t^i[t] = \mathbf{p}_t^i[t] + \mathbf{r}_t^i[t]$, set $\widetilde{\mathbf{r}}_t^i[t] = 0$ *and do not push back* $\mathbf{r}_t^i[t]$ *to the in-neighbors of* t *in the* $(i+1)^{th}$ *stage.* The remaining algorithm remains

the same. It is is plausible from the discussion in Section 4.3.2 that the resulting quantity $\widehat{p}_s^{\ell,hit}[t]$ is an unbiased estimate of $\mathbb{P}[\text{Hitting time of } t = \ell | X_0 \sim s]$ – we leave developing a complete algorithm to future work.

- **Graph Diffusions:** If we assign a weight α_i to random walks of length i on a (weighted) graph, the resulting scoring functions $f(W,s)[t] := \sum_{i=0}^{\infty} \alpha_i \left(s^T W^i\right)[t]$ are known as a *graph diffusions* [15] and are used in a variety of applications. The case where $\alpha_i = \alpha^{i-1}(1-\alpha)$ corresponds to PageRank. If instead the length is drawn according to a Poisson distribution (i.e., $\alpha_i = e^{-\alpha}\alpha^i/i!$), then the resulting function is called the *heat-kernel* $h(G,\alpha)$ – this too has several applications, including finding communities (clusters) in large networks [31]. Note that for any function f as defined above, the truncated sum $f^{\ell_{\max}} = \sum_{i=0}^{\ell_{\max}} \alpha_i \left(\pi_s^T W^i\right)$ obeys $||f - f^{\ell_{\max}}||_\infty \leq \sum_{\ell_{\max}+1}^{\infty} \alpha_i$. Thus a guarantee on an estimate for the truncated sum directly translates to a guarantee on the estimate for the diffusion. We can use MSTP-estimation to efficiently estimate these truncated sums. We perform numerical experiments on heat kernel estimation in the next section.

- **Conductance Testing in Graphs:** MSTP-estimation is an essential primitive for conductance testing in large Markov chains [23]. In particular, in regular undirected graphs, Kale et al [29] develop a sublinear bidirectional estimator based on counting collisions between walks in order to identify 'weak' nodes – those which belong to sets with small conductance. Our algorithm can be used to extend this process to any graph, including weighted and directed graphs.

4.5 Experiments

To demonstrate the efficiency of our algorithm on large Markov chains, we use *heat kernel estimation* (cf. Section 4.4) as an example application. The heat kernel is a graph diffusion, defined as the probability of stopping at the target on a random walk from the source, where the walk length is sampled from a $Poisson(\ell)$ distribution. It is important in practice as it has empirically been shown to detect communities well, in the sense that the heat kernel from s to t tends to be larger for nodes t sharing a

Table 4.1: Datasets used in experiments

Dataset	Type	# Nodes	# Edges
Pokec	directed	1.6M	30.6M
LiveJournal	undirected	4.8M	69M
Orkut	undirected	3.1M	117M
Twitter-2010	directed	42M	1.5B

community with s than for other nodes t. Note that for finding complete communities around s, previous algorithms which work from a single s to many targets are more suitable. Our algorithm is relevant for estimating the heat kernel from s to a small number of targets. For example, in the application of personalized search on a social network, if user s is searching for nodes t satisfying some query, we might find the top-k results using conventional information retrieval methods for some small k, then compute the heat kernel to each of these k targets, to allow us to place results t sharing an inferred community with s above less relevant results. A related example is if a social network user s is viewing a list of nodes t, say the set of users attending some event, our estimator can be used to rank the users t in that list in decreasing order of heat kernel. This can help s quickly find the users t closest to them in the network.

We use our estimator to compute heat kernels on diverse, real-world graphs that range from millions to billions of edges as described in Table 4.1. For each graph, for random (source, target) pairs, we compute the heat kernel. We set average walk-length $\ell = 5$ since for larger values of ℓ, the walk mixes into the stationary distribution, "forgetting" where it started. We set a maximum length of 10 standard deviations above the expectation $\ell + 10\sqrt{\ell} \approx 27$; the probability of a walk being longer than this is 10^{-12}, so we can ignore that event.

We compare Algorithm 8 to two state-of-the-art algorithms: The natural Monte Carlo algorithm, and the push forward algorithm introduced in [31]. All three have parameters which allow them to trade off speed and accuracy. For a fair comparison, we choose parameters such that the mean relative error of all three algorithms is around 10%, and for those parameters we measure the mean running time of all three algorithms. We implemented all three algorithms in Scala (for the push forward

algorithm, our implementation follows the code linked from [31]).

Our results are presented in Figure 4.5. We find that on these four graphs, our algorithm is 100x faster (per (s, t) pair) than these state-of-the-art algorithms. For example, on the Twitter graph, our algorithm can estimate a heat kernel score is 0.1 seconds, while the state-of-the-art algorithms both take more than 4 minutes. We note though that the state-of-the-art algorithms were designed for community detection, and return scores from the source to all targets, rather than just one target. Our algorithm's advantage applies in applications where we want the score from a source to a single target or small set of targets.

The Pokec [53], Live Journal [42], and Orkut [42] datasets were downloaded from the Stanford SNAP project [51]. The Twitter-2010 [11] was downloaded from the Laboratory for Web Algorithmics [33]. For reproducibility, the source code of our experiments are available on our website[3].

[3] http://cs.stanford.edu/~plofgren

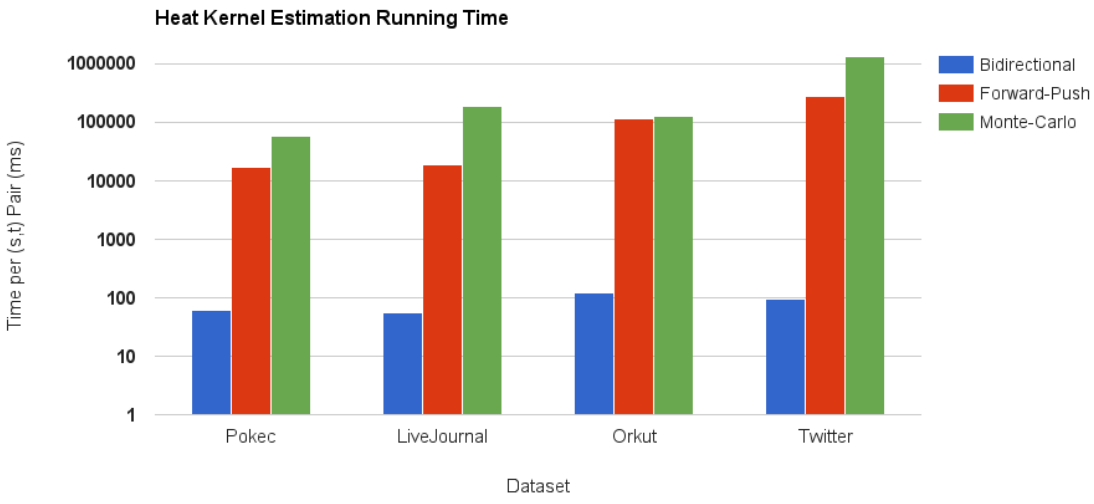


Figure 4.2: Estimating heat kernels: Bidirectional MSTP-estimation vs. Monte Carlo, Forward Push. For a fair comparison of running time, we choose parameters such that the mean relative error of all three algorithms is around 10%. Notice that our algorithm is 100 times faster per (source, target) pair than state-of-the-art algorithms.

Chapter 5

Precomputation for PPR Estimation

Our experiments show that `BidirectionalPPR` is efficient, especially for uniform random targets, but for worst-case targets with many in-neighbors, it can still take several seconds on Twitter-2010, which limits its usefulness for real-time search. In this chapter we describe a simple method of precomputing forward walk and reverse residual vectors for each node in a large graph which enables very fast, real-time PPR estimation.

5.1 Basic Precomputation Algorithm

As we show in Section 6.1, the estimation in `BidirectionalPPR` can be expressed as a simple dot product between a vector x^s depending only on s, and a vector y^t depending only on t, which can both be pre-computed. In particular, if we define $y^t = (p^t, r^t) \in \mathbb{R}^{2n}$ and $x^s = (e_s, \tilde{\pi}_s) \in \mathbb{R}^{2n}$) where e_s is the standard basis vector and $\tilde{\pi}_s$ is a precomputed random walk vector, then $\pi_s(t) \approx x^s \cdot y^t$. The basic idea of this section is to precompute all these vectors x^s and y^t in advance, then compute the dot-product at run time when s and t are given. The challenge is that the sparcity of the vectors y^t vary dramatically: on average they have $O(\sqrt{m}/\epsilon)$ non-zero values for ϵ relative error and constant failure probability (Theorem 1), but "popular" targets t

with high PageRank will have up to $\Theta(n)$ non-zero values. For this reason it can be slow to access y^t and compute the dot-product $x^s \cdot y^t$ at run time on a single machine. Also, the total size of all the x^t and y^t is $O(n\sqrt{m}/\epsilon)$, which makes it infeasible to store them on a single machine.

Our solution is to distribute all the vectors x^s and y^t among a set of servers in advance and compute the dot-product in a distributed way. Suppose we use k servers and sharding function $h : V \to [k]$, so the ith server stores the non-zero coordinates x^s_v and y^t_v for all $s, t \in V$ and for all v such that $h(v) = i$. Then we can write the dot-product as a sum of independent contributions

$$x^s \cdot y^t = \sum_{i=1}^{k} E_i$$

where

$$E_i = \sum_{v:h(v)=i} x^s(v) y^t(v).$$

During pre-computation, we compute each vector x^s and shard it among the k servers, so for each $s \in [n]$, server i has local access to $x^s(v)$ for all v such that $h(v) = i$. We similarly shard every vector y^t among the k servers. Given a query (s, t), we see that each shard can do a local dot-product based on the nodes v assigned to it to compute E_i, send a single number E_i to a central broker, and then the broker can return an estimate to $\pi_s(t)$.

The total communication cost is $O(k)$ and the expected computation per server (which concentrates because the number of nonzero entries of the vectors is much greater than k for realistic values of k and m) is $\frac{1}{k}$ times the computation cost for a single machine, which is the best parallelism speed-up we could hope for. In contrast, if we simply stored each vector x^s and y^r in a distributed key-value store, we would need to transfer at least one of the two vectors over the network, requiring communication $\Theta\left(\sqrt{m}/\epsilon\right)$.

The total storage cost of this method is $O(n\sqrt{m}/\epsilon)$ because we store $2n$ vectors (a forward vector x^v and reverse vector y^v for each $v \in V$) and the proof of Theorem

2 shows they each have $O(\sqrt{m}/\epsilon)$ non-zero coordinates on average. The total pre-computation time is also $O(n\sqrt{m}/\epsilon)$.

5.2 Combining Random Walks to Decrease Storage

We can decrease this storage by combining the random walk vector from multiple nodes. A simple version of this technique (proposed in [20]) just uses the neighbors of the source node s. To illustrate the idea, suppose every node s in the graph has 10 neighbors $v_1, \ldots, v_{10}$, and our accuracy analysis says we need 100,000 random walks from each s to form x^s. Then rather than pre-compute 100,000 random walks from each s, we can pre-compute 10,000 random walks from each node v in the graph to form $\tilde{x}^v$, and average the walks from the neighbors of s to form

$$x^s = (1-\alpha)\frac{1}{10}\sum_{i=1}^{10}\tilde{x}^{v_i}.$$

This is because if we sampled 100,000 walks from s, a fraction $1-\alpha$ of them would continue to a neighbor, and on average $\frac{1}{10}$ of those would continue from each neighbor, so we can combine the 10,000 walks at each neighbor to simulate 100,000 walks from s. Of course in real graphs some nodes s will have small out-degree and others will have large out-degree, so we need a more general technique.

We can combine the walks from a larger set of nodes around s using the `ForwardPush` algorithm from Section 2.4. Recall that this algorithm returns a vector of forward estimates p_s and forward residuals r_s which satisfy the invariant that for all $t \in V$

$$\pi_s(t) = p_s(t) + \sum_{v\in V} r_s(v)\pi_v(t).$$

If we apply the push forward technique until all forward residuals $r_s(v)$ are less than some threshold $r^{\mathrm{f}}_{\max}$, then we can simulate n_f walks from s by taking $r^{\mathrm{f}}_{\max}n_f$ random walks from each node u in the graph, letting $\tilde{x}^v$ be their stopping distribution, and

defining

$$x^s = \sum_{v \in V} r_s(v)\tilde{x}^{u_i}.$$

We don't need to explicitly form x^s on a single machine. Rather we pre-compute p_s and r_s and store them in a key-value store with key s. Then given query (s, t) the broker broadcasts r_s to all servers, each server i computes estimate

$$E_i = \sum_{v:h(v)=i} r_s(u)\tilde{x}^u(v)y^t(v)$$

and the broker returns $p_s(t) + \sum_i E_i \approx \pi_s(t)$.

Notice that averaging walks from the out-neighbors of s is a special case of this technique: after the first forward-push of their algorithm, $r_s(v) = (1-\alpha)\frac{1}{d^{out}(s)}$ for out-neighbors v of s and $r_s(v) = 0$ for other nodes v, so we recover the simpler out-neighbor technique.

The above algorithm can be improved by treating high-degree nodes specially. The issue is that if we push forward from a node with high out-degree (like Barack Obama who follows more than 500,000 people on Twitter), that greatly increases the size of the forward residual vector. Our proposed fix is to choose some maximum degree $d_{\max}$ and only push forward from nodes with degree at most $d_{\max}$. For nodes v of degree larger than $d_{\max}$, we do the full number of forward walks required for an accurate estimate without random walk sharing. Full pseudo-code is given in Algorithm 9 and 10.

5.3 Storage Analysis on Twitter-2010

To estimate the amount of storage needed, both with and without combining walks, we now present a storage analysis for the Twitter-2010 graph, one of the largest publicly available social networks with 1.5 billion edges. Without walk sharing, the only parameter to optimize is $r^r_{\max}$, the maximum residual for the reverse push algorithm. From Theorem 1, the number of walks needed per node is then $c_1\frac{r^r_{\max}}{\delta}$ where δ is the smallest personalized PageRank value we can accurately estimate (for example

Algorithm 9 `PPR-Precomputation`$(G, \alpha, \delta, d_{\max})$

Inputs: graph $G = (V, E)$ with edge weights $w_{u,v}$, teleport probability α, minimum PPR δ (estimates of $\pi_s(t)$ are accurate if $\pi_s(t) \geq \delta$), max degree parameter $d_{\max}$

1: Note: This algorithm runs on a cluster of large machines which can each hold the graph in RAM. The for loop over $v \in V$ is parallelized across these large machines.
2: Choose parameters $r^{\mathrm{r}}_{\max} = \sqrt[3]{\frac{c_2^2\delta}{c_1 c_3}}$, $r^{\mathrm{f}}_{\max} = \sqrt[3]{\frac{c_3^2\delta}{c_1 c_2}}$, $n_{\mathrm{w}} = \frac{c_1 r^{\mathrm{f}}_{\max} r^{\mathrm{r}}_{\max}}{\delta}$ (see Section 5.3 for analysis and details on these constants).
3: Choose some hash function $h : V \to [n_s]$, where n_s is the number of servers.
4: **for** all nodes $v \in V$ **do**
5: Run algorithm `ForwardPush`$(v,\ \alpha,\ r^{\mathrm{f}}_{\max})$ with two modifications: First, Run until $\forall u, r_v(u) < r^{\mathrm{f}}_{\max}$. Second, only push forward from nodes u with $d^{out}(u) \leq d_{\max}$. Store the resulting forward estimate and residual vectors (p_v, r_v) in a distributed key-value store with key v.
6: Run algorithm `ReversePush`$(G, \alpha, v, r^{\mathrm{r}}_{\max})$. With the resulting reverse estimate and residual vectors (p^v, r^v), define $y^v = (p^v, r^v) \in \mathbb{R}^{2n}$ and send each nonzero entry $y^v(u)$ to server $h(u)$.
7: If $d^{out}(v) \leq d_{\max}$, sample n_{w} random walks from v; otherwise sample $\frac{c_1 r^{\mathrm{r}}_{\max}}{\delta}$ walks from v. Stop each walk after each step with probability α. Let $\tilde{\pi}_v$ be the distribution of their endpoints. Define $\tilde{x}_v = (e_s, \tilde{\pi}_s) \in \mathbb{R}^{2n}$ and send each non-zero entry $\tilde{x}_v(u)$ to server $h(u)$.
8: **end for**

Algorithm 10 `PPR-Query-Given-Precomputation`(s, t)

Inputs: source node s, target node t

1: Note: This is executed on an app server or broker server.
2: Retrieve the forward estimate and residual vector (p_s, r_s) from the key-value store.
3: Broadcast (s, t, r_s) to all ppr storage servers.
4: Each server i computes

$$E_i = \sum_{v:h(v)=i} r_s(u)\tilde{x}_u(v)y^t(v)$$

and returns E_i to the broker.
5: **return** $p_s(t) + \sum_i E_i \approx \pi_s(t)$.

$\delta = \frac{4}{n}$ as in our experiments) and c_1 is a constant controlling the relative error of the estimates. Our experiments in Section 3.4.1 found that $c_1 = 7$ achieved mean

relative error less than 10%, which is a natural accuracy goal given that the personalized PageRanks vary on a wide scale. The storage requirement of the random walk endpoint distribution is thus bounded simply by the total number of walks

$$nc_1 \frac{r^{\mathrm{r}}_{\max}}{\delta}.$$

Now we consider the storage cost of the residual vectors. A worst case bound on the total size (i.e. number of non-zeros) of the residual vectors from Theorem 2 is $\frac{m}{r^{\mathrm{r}}_{\max}}$. However, we find that in practice the total size is smaller than this. On Twitter-2010, we sampled 2000 nodes t uniformly at random, ran the algorithm `ReversePush` for various values of $r^{\mathrm{r}}_{\max}$ and measured the number of non-zero entries in the residual vector. Figure 5.1 shows that $\frac{c_2}{r^{\mathrm{r}}_{\max}}$, where $c_2 \approx 0.5$ for Twitter-2010, is a good estimate of the reverse storage required per node. Thus the total storage required is approximately

$$nc_1 \frac{r^{\mathrm{r}}_{\max}}{\delta} + nc_2 \frac{1}{r^{\mathrm{r}}_{\max}}.$$

If we choose $r^{\mathrm{r}}_{\max}$ to minimize this expression, we get storage of $2n\sqrt{\frac{c_1 c_2}{\delta}} = 1.0 \cdot 10^{12}$, or 8TB assuming each non-zero entry takes 8 bytes to store. Here $n = 41$ million and we choose $\delta = \frac{4}{n}$ (meaning we can estimate PPR values 4 times the PPR of a random node).

Now we consider how this improves if we re-use walks. Recall from above that we need to choose a parameter $r^{\mathrm{f}}_{\max}$ which is the maximum forward residual value we allow, and then the number of forward walks needed is $\frac{c_1 r^{\mathrm{f}}_{\max} r^{\mathrm{r}}_{\max}}{\delta}$. To estimate the size of the forward residuals, we sample 2000 uniformly random nodes on Twitter-2010, run the `ForwardPush`algorithm modified as described in Algorithm 9 for each node for various values of $r^{\mathrm{f}}_{\max}$, and measure the mean number of non-zero entries in the forward residual vectors. As described above, we did not push forward from nodes with degree greater than $d_{\max}$, where we chose $d_{\max} = 1000$. Figure 5.2 shows that $\frac{c_3}{r^{\mathrm{f}}_{\max}}$, where $c_3 \approx 10$ for Twitter-2010, is a good estimate of the forward residual

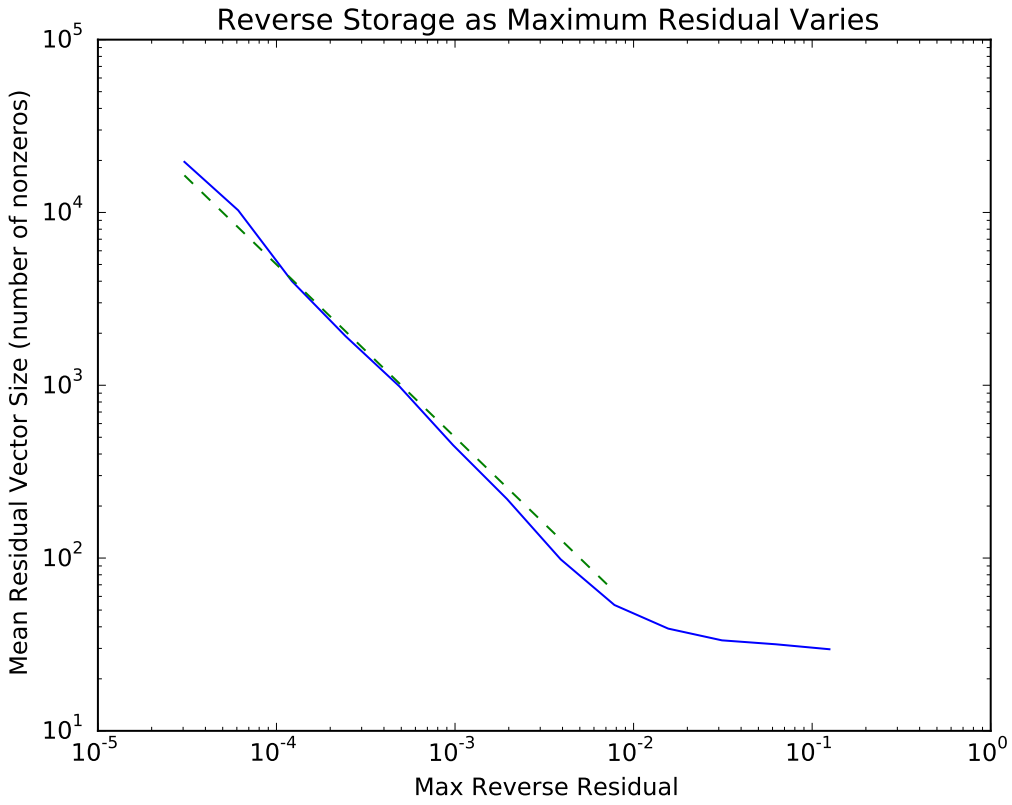


Figure 5.1: Mean reverse residual size (number of non-zeros) as $r^{\mathrm{r}}_{\max}$ varies, on Twitter-2010. The dotted line is $y = \frac{0.5}{r^{\mathrm{r}}_{\max}}$.

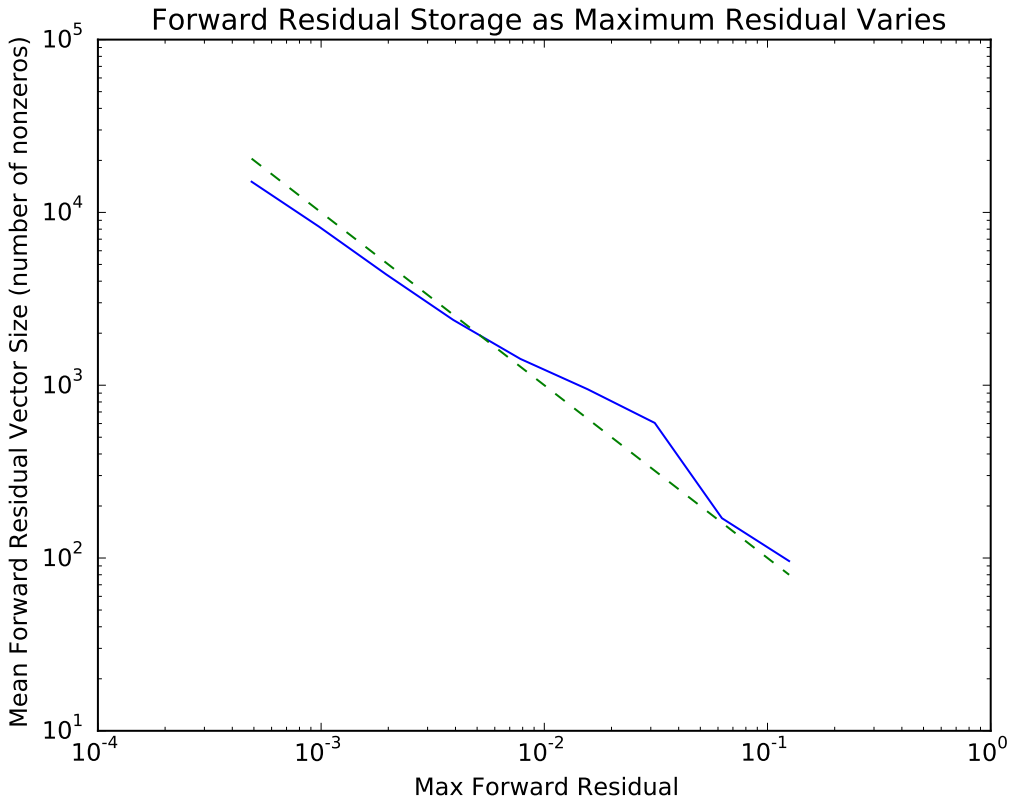


Figure 5.2: Mean forward residual size (number of non-zeros) as $r^{\mathrm{f}}_{\max}$ varies on Twitter-2010. The dotted line is $y = \frac{10}{r^{\mathrm{f}}_{\max}}$.

storage required per node. The total storage required is then approximately

$$n\frac{c_3}{r^{\mathrm{f}}_{\max}} + nc_1\frac{r^{\mathrm{r}}_{\max}r^{\mathrm{f}}_{\max}}{\delta} + n\frac{c_2}{r^{\mathrm{r}}_{\max}}.$$

If we choose $r^{\mathrm{r}}_{\max}$ and $r^{\mathrm{f}}_{\max}$ to minimize this expression, we get storage of $3n\sqrt[3]{\frac{c_1c_2c_3}{\delta}} = 1.4 \cdot 10^{11}$ entries under the same assumptions as above. In addition, for nodes with outdegree greater than $d_{\max}$, we need to store the full number of $\frac{c_1 r^{\mathrm{r}}_{\max}}{\delta}$ random walks. On Twitter-2010, only 220K nodes (0.5% of all nodes) have degree greater than $d_{\max} = 1000$, so their total storage is $220,000 \cdot \frac{c_1 r^{\mathrm{r}}_{\max}}{\delta} \approx 3 \cdot 10^{10}$ entries. The total of these is $1.7 \cdot 10^{11}$ entries, or 1.4 TB.

5.4 Varying Minimum Probability to Reduce Storage

One more optimization that can reduce storage is to vary the minimum probability δ for each target t. In particular, we can set $\delta' = \max(\delta, \pi[t])$, since, as we alluded to in Section 3.2.3, if $\pi_s[t] < \pi[t]$, then s is less interested in t than a uniform random source is, so it is not useful to measure how (un)interested s is in t. Since we pre-compute the walks for each source s independently of t, we cannot vary the number of walks simulated w, but for a given number of simulated walks we can solve the equation $w = c_1\frac{r^{\mathrm{r}}_{\max}}{\delta'} = c_1\frac{r^{\mathrm{r}}_{\max}}{\max(\delta,\pi[t])}$ for $r^{\mathrm{r}}_{\max}$ to get

$$r^{\mathrm{r}}_{\max} = \frac{w\max(\delta, \pi[t])}{c_1}.$$

We leave the analysis of this optimization to future work.

Chapter 6

Personalized PageRank Search

In this chapter, based on [38], we extend our method of estimating PPR between a single (source, target) pair to doing full searches from a source s to a set of candidate targets.

6.1 Personalized PageRank Search

We now turn from Personalized PageRank estimation to the Personalized PageRank search problem:

> *Given a start node s (or distribution σ) and a query q which filters the set of all targets to some list $T = \{t_i\} \subseteq V$, return the top-k targets ranked by $\pi_s[t_i]$.*

We consider as baselines two algorithms which require no pre-computation. They are efficient for certain ranges of $|T|$, but our experiments show they are too slow for real-time search across most values of $|T|$:

- `Monte Carlo` [5, 21]: Sample random walks from s, and filter out any walk whose endpoint is not in T. If we desire n_s samples, this takes time $O(n_s/\pi_s(T))$, where $\pi_s(T) := \sum_{t \in T} \pi_s(t)$ is the probability that a random walk terminates in T. This method works well if T is large, but in our experiments on Twitter-2010 it takes minutes per query for $|T| = 1000$ (and hours per query for $|T| = 10$).

- `BiPPR-Basic`: On the other hand, we can estimate $\pi_s(t)$ to each $t \in T$ separately using `BidirectionalPPR`. This has an average-case running time $O\left(|T|\sqrt{\bar{d}/\delta_k}\right)$ where δ_k is the PPR of the k^{th} best target. This method works well if T is small, but is too slow for large T; in our experiments, it takes on the order of seconds for $|T| \leq 100$, but more than a minute for $|T| = 1000$.

If we are allowed pre-computation, then we can improve upon the sequential `BidirectionalPPR` procedure (`BiPPR-Basic`) by precomputing and storing a reverse vector from all target nodes. To this end, we first observe that the estimate $\hat{\pi}_s(t)$ can be written as a dot-product. Let $\tilde{\pi}_s$ be the empirical distribution over terminal nodes due to w random walks from s (with w chosen as in Theorem 1); we define the *forward vector* $x_s \in \mathbb{R}^{2n}$ to be the concatenation of the basis vector e_s and the random-walk terminal node distribution. On the other hand, we define the *reverse vector* $y^t \in \mathbb{R}^{2n}$, to be the concatenation of the estimates p^t and the residuals r^t. Formally, we have:

$$x_s = (e_s, \tilde{\pi}_s) \in \mathbb{R}^{2n}, \qquad y^t = (p^t, r^t) \in \mathbb{R}^{2n} \tag{6.1}$$

Finally, from Algorithm 3, we have:

$$\hat{\pi}_s(t) = \langle x_s, y^t \rangle \tag{6.2}$$

The above observation motivates the following algorithm:

- `BiPPR-Precomp`: In this approach, we first use `BidirectionalPPR` to pre-compute and store a reverse vector y^t for each $t \in V$. At query time, we generate random walks to form the forward vector x_s; now, given any set of targets T, we compute $|T|$ dot-products $\langle x_s, y^t \rangle$, and use these to rank the targets. This method now has an *worst-case* running time $O\left(|T|\sqrt{\bar{d}/\delta_k}\right)$. In practice, it works well if T is small, but is too slow for large T. In our experiments (doing 100,000 random walks at runtime) this approach takes around a second for $|T| \leq 30$, but this climbs to a minute for $|T| = 10,000$.

Although the `BiPPR-Precomp` approach is faster than `BiPPR-Basic` (albeit requiring additional precomputation and storage), and also faster than `Monte Carlo` for small sets T, it is still not efficient enough for real-time personalized search. This

motivates us to find a more efficient algorithm that scales better than `BiPPR-Basic` for large T, yet is fast for small $|T|$. In the following sections, we propose two different approaches for this – the first based on pre-grouping the precomputed reverse-vectors, and the second based on sampling target nodes from T according to PPR. For convenience, we first summarize the two approaches:

- `BiPPR-Precomp-Grouped`: Here, as in `BiPPR-Basic`, we compute an estimate to each $t \in T$ using `BidirectionalPPR`. However, we leverage the sparsity of the reverse vectors $y^t = (p^t, r^t)$ by first grouping them by coordinates; this makes the dot-product more efficient. This method has a *worst-case* running time of $O\left(|T|\sqrt{\bar{d}/\delta_k}\right)$; more significantly, in experiments, we find it is much faster than `BiPPR-Precomp`. For our parameter choices its running time is less than 250ms across the range of $|T|$ we tried.
- `BiPPR-Precomp-Sampling`: We again first pre-compute the reverse vectors y^t. Next, for a given target t, we define the *expanded target-set* $T_t = \{v \in [2n] | y^t(v) \neq 0\}$, i.e., the set of nodes with non-zero reverse vectors from t. At run-time, we now sample random walks forward from s to nodes in the expanded target sets. Using these, we create a sampler in average time $O\left(r_{\max}/\delta_k\right)$ (where as before δ_k is the k^{th} largest PPR value $\pi_s[t_i]$), which samples nodes $t \in T$ with probability proportional to the PPR $\pi_s(t)$. We describe this in detail in Section 6.1.2. Once the sampler has been created, it can be sampled in $O(1)$ time per sample. The algorithm works well for any size of T, and has the unique property that in can identify the top-k target nodes without computing a score for all $|T|$ of them. For our parameter choice its running time is less than 250ms across the range of $|T|$ we tried.

We note here that the case $k = 1$ (i.e., for finding the top PPR node) corresponds to solving a Maximum Inner Product Problem. In a recent line of work, Shrivastava and Li [48, 49] propose a sublinear time algorithm for this problem based on Locality Sensitive Hashing; however, their method assumes that there is some bound U on $\|y^t\|_2$ and that $\max_t \langle x_s, y^t \rangle$ is a large fraction of U. In personalized search, we usually encounter small values of $\max_t \langle x_s, y^t \rangle$ relative to $\max \|y^t\|_2$ – finding an LSH for Maximum Inner Product Search in this regime is an interesting open problem

for future research. Our two approaches bypass this by exploiting particular structural features of the problem – `BiPPR-Precomp-Grouped` exploits the sparsity of the reverse vectors to speed up the dot-product, and `BiPPR-Precomp-Sampling` exploits the skewed distribution of PPR scores to find the top targets without even computing the dot-products.

6.1.1 Bidirectional-PPR with Grouping

In this method we improve the running-time of `BiPPR-Precomp` by pre-grouping the reverse vectors corresponding to each target set T. Recall that in `BiPPR-Precomp`, we first pre-compute reverse vectors $y^t = (p^t, r^t) \in \mathbb{R}^{2n}$ using `ReversePush` for each t; at run-time, given s, we compute forward vector $x_s = (e_s, \tilde{\pi}_s)$ by generating sufficient random-walks, and then compute the scores $\langle x_s, y^t \rangle$ for $t \in T$. Our main observation is that *we can decrease the running time of the dot-products by pre-grouping the vectors y^t by coordinate*. The intuition behind this is that in each dot product $\sum_v x_s[v]y^t[v]$, the nodes v where $x_s[v] \neq 0$ often don't have $y^t[v] \neq 0$, and most of the product terms are 0. Hence, we can improve the running time by grouping the vectors y^t in advance by coordinate v. Now, at run-time, for each v such that $x_s[v] \neq 0$, we can efficiently iterate over the set of targets t such that $y^t[v] \neq 0$.

An alternative way to think about this is as a sparse matrix-vector multiplication $Y^T x_s$ after we form a matrix Y^T whose rows are y^t for $t \in T$. This optimization can then be seen as a sparse column representation of that matrix.

Algorithm 11 BiPPRGroupedPrecomputation($T, r_{\max}$)

Inputs: Graph G, teleport probability α, target nodes T, maximum residual $r_{\max}$

1: $z \leftarrow$ empty hash map of vectors such that for any v, $z[v]$ defaults to an empty (sparse) vector in $\mathbb{R}^{2|V|}$
2: **for** $t \in T$ **do**
3: Compute $y^t = (p_t, r_t) \in \mathbb{R}^{2|V|}$ via `ReversePush`$(G, \alpha, t, r_{\max})$
4: **for** $v \in [2|V|]$ such that $y_t[v] > 0$ **do**
5: $z[v][t] = y^t[v]$
6: **end for**
7: **end for**
8: **return** z

Algorithm 12 BiPPRGroupedRankTargets($s, r_{\max}, z$)

Inputs: Graph G, teleport probability α, start node s, maximum residual $r_{\max}$, z: hash map of reverse vectors grouped by coordinate

Set number of walks $w = c\frac{r_{\max}}{\delta}$ (In experiments we found $c = 20$ achieved precision@3 above 90%.)
Sample w random-walks of length $Geometric(\alpha)$ from s; compute $\tilde{\pi}_s[v]$ = fraction of walks ending at node v
Compute $x_s = (e_s, \tilde{\pi}_s) \in \mathbb{R}^{2|V|}$
Initialize empty map score from V to $\mathbb{R}$
for v such that $x_s[v] > 0$ **do**
  for t such that $z[v][t] > 0$ **do**
    score(t) += $x_s[v]z_v[t]$
  end for
end for
Return T sorted in decreasing order of score

We refer to this method as `BiPPR-Precomp-Grouped`; the complete pseudo-code is given in Algorithm 12. The correctness of this method follows again via Theorem 1. In experiments, this method is efficient for T across all the sizes of T we tried, taking less than 250 ms even for $|T| = 10,000$. The improved running time of `BiPPR-Precomp-Grouped` comes at the cost of more storage compared to `BiPPR-Precomp`. In the case of name search, where each target typically only has a first and last name, each vector y^t only appears in two of these pre-grouped structures, so the storage is only twice the storage of `BiPPR-Precomp`. On the other hand if a target t contains many keywords, it will be included in many of these pre-grouped data structures, and its significantly improved running time is achieved through the cost of significantly greater storage.

6.1.2 Sampling from Targets Matching a Query

The key idea behind this alternate method for PPR-search is that by sampling a target t in proportion to its PageRank we can quickly find the top targets without iterating over all of them. After drawing several samples, the targets can be ranked according to the number of times they are sampled. Alternatively a full `BidirectionalPPR` query can be issued for some subset of the targets before ranking. This approach

exploits the skewed distribution of PPR scores in order to find the top targets. In particular, prior empirical work based on the Twitter graph has shown that for each fixed s, the values $\pi_s[T]$ follow a power law [7].

We define the PPR-Search Sampling Problem as follows:

Given a source distribution s and a query q which filters the set of all targets to some list $T = \{t_i\} \subseteq V$, sample a target t_i with probability $p[t_i] = \frac{\pi_s[t_i]}{\sum_{t \in T} \pi_s[T]}$.

We develop two solutions to this sampling problem. The first, in $O(w) = O(r_{\max}/\delta_k)$ time, generates a data structure which can generate an arbitrary number of independent samples from a distribution which approximates the correct distribution. The second can generate samples from the exact distribution $\pi_s[t_i]$, and generates complete paths from s to some $t \in T$, but requires time $O(r_{\max}/\pi_s[T])$ per sample. Because the approximate sampler is more efficient, we present that here and defer the exact sampler to Section 6.2.

The high level idea behind our method is *hierarchical sampling*. Recall that the start node s has an associated forward vector $x_s = (e_s, \pi_s)$, and from each target t we have a reverse vector y^t; the PPR-estimate is given by $\pi_s[T] \approx \langle x_s, y^t \rangle$. Thus we want to sample $t \in T$ with probability:

$$p[T] = \frac{\langle x_s, y^t \rangle}{\sum_{j \in T} \langle x_s, y^j \rangle}.$$

We will sample t in two stages: first we sample an intermediate node $v \in V$ with probability:

$$p'_s[v] = \frac{x_s[v] \sum_{j \in T} y^j[v]}{\sum_{j \in T} \langle x_s, y^j \rangle}.$$

Following this, we sample $t \in T$ with probability:

$$p''_v[T] = \frac{y^t[v]}{\sum_{j \in T} y^j[v]}.$$

It is easy to verify that $p[T] = \sum_{v \in V} p'_s[v] p''_v[T]$. Figure 6.1 shows how the sampling algorithm works on an example graph.

The `BiPPR-Precomp-Sampling` Algorithm

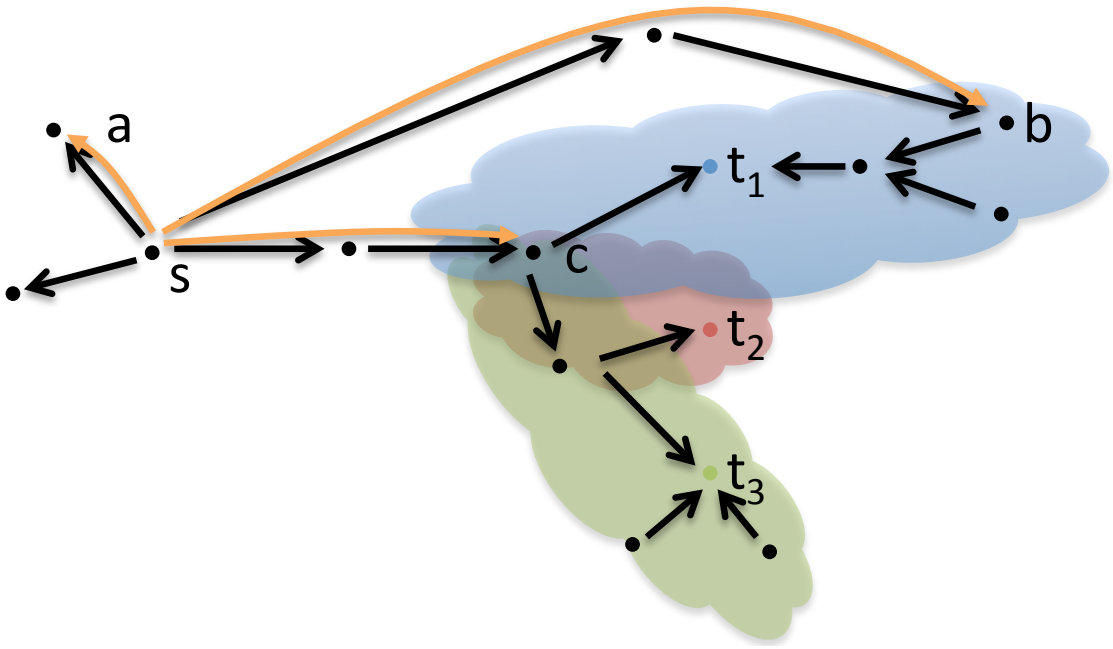


Figure 6.1: Search Example: Given target set $T = \{t_1, t_2, t_3\}$, for each target t_i we have drawn the expanded target-set, i.e., nodes v with positive residual $y^{t_i}[v]$. From source s, we sample three random walks, ending at nodes a, b, and c. Now suppose $y^{t_1}(b) = 0.64, y^{t_1}(c) = 0.4, y^{t_2}(c) = 0.16$, and $y^{t_3}(c) = 0.16$ – note that the remaining residuals are 0. Then we have $y^T(a) = 0, y^T(b) = 0.64$ and $y^T(c) = 0.72$, and consequently, the sampling weights of (a, b, c) are $(0, 0.213, 0.24)$. Now, to sample a target, we first sample from a, b, c in proportion to its weight, so we first sample from (a, b, c) – if we sample b, we always return t_1; if we sample c, then we sample (t_1, t_2, t_3) with probability $(5/9, 2/9, 2/9)$.

For a query Q, let $T = \{t \in V : t \text{ is relevant to } Q\}$ be the associated set of target nodes. We assume that some set of supported queries is known in advance, and we first pre-compute and store a separate data-structure for each query (i.e., for each target-set T).

Precomputation: Given a target-set T, for each $t \in T$, we compute residuals r^t and estimates p^t using `ReversePush`$(G, \alpha, t, r_{\max})$ (Algorithm 1); the choice of $r_{\max}$ is discussed in the analysis section. For each T and each v, we also pre-compute

$\sum_{j \in T} y_v^j$ and a constant-time sampler for the distribution:

$$p''_v[T] = \frac{y^t[v]}{\sum_{j \in T} y^j[v]}.$$

The pseudo-code is given in Algorithm 13.

Algorithm 13 SamplerPrecomputationForSet($T, r_{\max}$)

Inputs: Graph G, teleport probability α, target-set T, maximum residual $r_{\max}$

1: **for** $t \in T$ **do**
2: Compute $y^t = (p_t, r^t) \in \mathbb{R}^{2|V|}$ via `ReversePush`$(G, \alpha, t, r_{\max})$
3: **end for**
4: Compute $y^T = \sum_{t \in T} y^t$
5: **for** $v \in V$ such that $y^T[v] > 0$ **do**
6: Create sampler$_v$ which samples t with probability $p''_v[T]$ (For example, using the alias sampling method [56], [32, section 3.4.1]).
7: **end for**
8: **return** $(y^T, \{\text{sampler}_v\})$

In addition, we can optionally pre-compute w random walks from each start-node s, and store the forward vector x_s.

Sampling Algorithm: Given start node s and target-set T, we load the relevant target data structures for T and either load x_s or compute it at run-time. Next, in $O(w)$ time, we create a sampler for the distribution $p'_s[v]$ over v:

$$p'_s[v] = \frac{x_s[v] \sum_{j \in T} y^j[v]}{\sum_{j \in T} \langle x_s, y^j \rangle} = \frac{x_s[v] y^T[v]}{\langle x_s, y^T \rangle}.$$

Now we can generate independent samples in constant time per sample. We sample v from this distribution, and then sample t from $p''_v[T]$ using the pre-computed sampler. The complete pseudo-code is given in Algorithm 14.

We refer to this method as the `BiPPR-Precomp-Sampling` procedure.

Running Time: For a small relative error for targets with $\pi_s[T] > \delta$, we use $w = cr_{\max}/\delta$ walks, where c is chosen as in Theorem 1. The support of our forward sampler is at most w so its construction time is $O(w)$ using the alias method of sampling from a discrete distribution [56], [32, section 3.4.1]. Once constructed, we can get

Algorithm 14 SampleAndRankTargets($s, r_{\max}, y^T, \{\text{sampler}_v\}$)

Inputs: Graph G, teleport probability α, start node s, maximum residual $r_{\max}$, reverse vectors y^T, intermediate-node-to-target samplers $\{\text{sampler}_v\}$.

1: Set number of walks $w = c\frac{r_{\max}}{\delta}$. In experiments we found $c = 20$ achieved precision@5 above 90% on Twitter-2010.
2: Set number of samples n_s (We use $n_s = w$)
3: Sample w random walks from s and let $\tilde{\pi}_s$ be the empirical distribution of their endpoints; compute forward vector $x_s = (e_s, \tilde{\pi}_s) \in \mathbb{R}^{2|V|}$
4: Create sampler_s to sample $v \in [2|V|]$ with probability $p'_s[v]$, i.e., proportional to $x_s[v]y^T[v]$
5: Initialize empty map score
6: **for** $j \in [0, n_{\text{sample}} - 1]$ **do**
7: Sample v from sampler_s
8: Sample t from sampler_v
9: Increment score(t)
10: **end for**
11: Return T sorted in decreasing order of score

independent samples in $O(1)$ time. Thus the query time to generate n_s samples is $O(cr_{\max}/\delta + n_s)$.

Accuracy: `BiPPR-Precomp-Sampling` does not sample exactly in proportion to π_s; instead, the sample probabilities are proportional to a distribution $\hat{\pi}_s$ satisfying the guarantee of Theorem 1. In particular, for all targets t with $\pi_s[T] \geq \delta$, this will have a small relative error ϵ, while targets with $\pi_s[T] < \delta$ will likely be sampled rarely enough that they won't appear in the set of top-k most sampled nodes.

Storage Required: The storage requirements for the `BiPPR-Precomp-Sampling` algorithm (and also for the previous algorithms) depends on the distribution of keywords and how $r_{\max}$ is chosen for each target set. For simplicity, here we assume a single maximum residual $r_{\max}$ across all target sets, and assume each target is relevant to at most γ keywords. For example, in the case of name search, each user typically has a first and last name, so $\gamma = 2$.

Theorem 7. *Let graph G, minimum-PPR value δ and time-space trade-off parameter r_{max} be given, and suppose every node contains at most γ keywords. Then the total storage needed for* ***`BiPPR-Precomp-Sampling`*** *to construct a sampler for any source*

node (or distribution) s and any set of targets T corresponding to a single keyword is $O(\gamma m/r_{max})$.

We can choose $r_{\max}$ to trade-off this storage requirement with the running time requirement of $O(cr_{\max}/\delta)$ – for example, we can set both the query running-time and per-node storage to $\sqrt{c\gamma\bar{d}/\delta}$ where $\bar{d} = m/n$ is the average degree. Now for name search, if we choose $\delta = \frac{1}{n}$ and since $\gamma = 2$, we see that the query-time and per-node storage is $O(\sqrt{m})$.

Proof. For each set T corresponding to a keyword, and each $t \in T$, we push from nodes v until for each v, $r^t[v] < r_{\max}$. Each time we push from a node v, we add an entry to the residual vector of each node $u \in \mathcal{N}^{in}(v)$, so the space cost is $d^{in}(v)$. Each time we push from a node v, we increase the estimate $p_t[v]$ by $\alpha r^t[v] \geq \alpha r_{\max}$, and $\sum_{t\in T} p_t[v] \leq \pi_v[T]$ so v can be pushed from at most $\frac{\pi_v[T]}{\alpha r_{\max}}$ times. Thus (ignoring the constant α) the total storage required is

$$\sum_{v\in V} d^{in}(v)(\#\text{ of times } v \text{ pushed}) \leq \sum_{v\in V} d^{in}(v)\frac{\pi_v[T]}{r_{\max}} \tag{6.3}$$

Let $\mathcal{T}$ be the set of all target sets (one target set per keyword). Then the total storage over all keywords is

$$\begin{aligned}\sum_{T\in\mathcal{T}}\sum_{t\in T}\sum_{v\in V} d^{in}(v)\frac{\pi_v[t]}{r_{\max}} &\leq \gamma\sum_{v\in V}\sum_{t\in V} d^{in}(v)\frac{\pi_v[t]}{r_{\max}} \\ &\leq \gamma\sum_{v\in V} d^{in}(v)\frac{1}{r_{\max}} \leq \gamma\frac{m}{r_{\max}}.\end{aligned}$$

□

Adaptive Maximum Residual: One way to improve the storage requirement is by using larger values of $r_{\max}$ for target sets T with larger global PageRank. Intuitively, if T is large, then it's easier for random walks to get close to T, so we don't need to push back from T as much as we would for a small T. We now formalize this scheme, and outline the savings in storage via a (heuristic) argument, based on a model of personalized PageRank values introduced by Bahmani et al. [7].

For a fixed s, we assume the values $\pi_s[t_i]$ for all $t_i \in V$ approximately follow a power law with exponent β. Empirically, this is known to be an accurate model for the Twitter graph – Bahmani et al. [7] find that the mean exponent for a user is $\beta = 0.77$ with standard deviation 0.08. To analyze our algorithm, we further assume that π_s restricted to T also follows a power law, i.e.:

$$\pi_s[t_i] = \frac{1-\beta}{|T|^{1-\beta}} i^{-\beta} \pi_s[T]. \tag{6.4}$$

Suppose we want an accurate estimate of $\pi_s[t_i]$ for the top-k results within T, so we set $\delta_k = \pi_s[t_k]$. From Theorem 1, the number of walks required is:

$$w = c\frac{r_{\max}}{\delta_k} = c_2 \frac{r_{\max} |T|^{1-\beta}}{\pi_s[T]}$$

where $c_2 = k^\beta c/(1-\beta)$. If we fix the number of walks as w, then we must set $r_{\max} = w\pi_s[T]/(c_2 |T|^{1-\beta})$. Also, for a uniformly random start node s, we have $\pi_s[T] = \pi[T]$. This suggests we choose $r_{\max}[T]$ for set T as:

$$r_{\max}[T] = \frac{w\pi[T]}{c_2 |T|^{1-\beta}} \tag{6.5}$$

Going back to equation (6.3), suppose for simplicity that the average $d^{in}(v)$ encountered is $\bar{d}$. Then the storage required for this keyword is bounded by:

$$\sum_{v \in V} d^{in}(v) \frac{\pi_v[T]}{r_{\max}} = \bar{d}\frac{n\pi[T]}{r_{\max}} = \frac{mc_2 |T|^{1-\beta}}{w}.$$

Note that this is independent of $\pi[T]$ - however, there is still a dependence on $|T|$, which is natural since for larger T there are more nodes which make it harder to find the top-k. For $\beta = 0.77$, the rate of growth, $|T|^{0.23}$ is fairly small, and in particular is sublinear in $|T|$.

Dynamic Graphs: So far we have assumed that the graph and keywords are static, but in practice they change over time. When a keyword is added to some node T, the node's reverse vector y^t needs to be added to the sampling data structure for that

keyword. When an edge is added, the residual values need to be updated, and in in Section 7.2.3 we sketch an algorithm that could be used for this. We leave extension the dynamic graphs to future work.

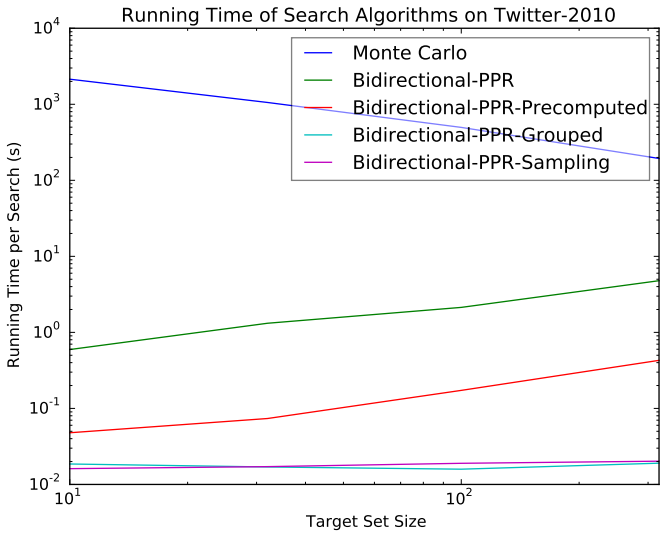
6.1.3 Experiments

We conduct experiments to test the efficiency of these personalized search algorithms as the size of the target set varies. We use one of the largest publicly available social networks, Twitter-2010 [33] with 40 million nodes and 1.5 billion edges. For various values of $|T|$, we select a target set T uniformly among all sets with that size, and compare the running times of the four algorithms we propose in this work, as well as the `Monte Carlo` algorithm. We repeat this using 10 random target sets and 10 random sources s per target set, and report the median running time for all algorithms. We use the same target sets and sources for all algorithms.

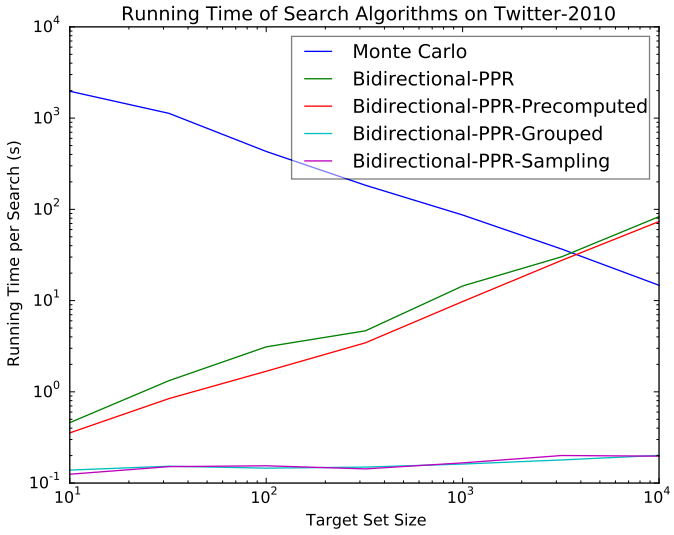
Parameter Choices: Because all five algorithms have parameters that trade-off running time and accuracy, we choose parameters such that the accuracy is comparable so we can compare running time on a level playing field. To choose a concrete benchmark, we chose parameters such that the precision@3 of the four algorithms we propose are consistently greater than 90% for the range of $|T|$ we used in experiment. We chose parameters for `Monte Carlo` so that our algorithms are consistently more accurate than it, and its precision@3 is greater than 85%. In the appendix we plot the precision@3 of the algorithms for the parameters we use when comparing running time.

We used $\delta = \pi_s(t_k)$ where $\pi_s(t_k)$ is estimated using Eqn. 6.4, using $k = 3$, power law exponent $\beta = 0.77$ (the mean value found empirically on Twitter), and assuming $\pi_s(T) = \frac{|T|}{n}$ (the expected value of $\pi_s(T)$ since T is chosen uniformly at random). Then we use Equation 6.5 to set $r_{\max}$, using $c = 20$ and two values of w, 10,000 and 100,000. We used the same value of $r_{\max}$ for `BiPPR-Precomp`, `BiPPR-Precomp-Grouped`, and `BiPPR-Precomp-Sampling`. For `Monte Carlo`, we sampled $\frac{40}{\delta}$ walks[1].

[1]Note that `Monte Carlo` was too slow to finish in a reasonable amount of time, so we measured the average time required to take 10 million walks, then multiplied by the number of walks needed. When measuring precision, we simulated the target weights `Monte Carlo` would generate, by sampling t_i



(a) Running Time, More Precomputation



(b) Running Time, Less Precomputation

Figure 6.2: Running time on Twitter-2010 (1.5 billion edges), with parameters chosen such that the Precision@3 of our algorithms exceeds 90% and exceeds the precision@3 of `Monte Carlo`. The two plots demonstrate the storage-runtime tradeoff: Figure 6.2(a) (which performs $10K$ walks at runtime) uses more pre-computation and storage compared to Figure 6.2(b) (with $100K$ walks).

Results: Figure 6.2 shows the running time of the five algorithms as $|T|$ varies for two different parameter settings in the trade-off between running time and precomputed storage requirement. Notice that `Monte Carlo` is very slow on this large graph for small target set sizes, but gets faster as the size of the target set increases. For example when $|T| = 10$ Monte Carlo takes half an hour, and even for $|T| = 1000$ it takes more than a minute. `BidirectionalPPR` is fast for small T, but slow for larger T, taking more than a second when $|T| \geq 100$. In contrast, `BiPPR-Precomp-Grouped` and `BiPPR-Precomp-Sampling` are both fast for all sizes of T, taking less than 250 ms when $w = 10,000$ and less than 25 ms when $w = 100,000$.

The improved running time of `BiPPR-Precomp-Grouped` and `BiPPR-Precomp-Sampling`, however, comes at the cost of pre-computation and storage. With these parameter choices, for $w = 10,000$ the pre-computation size per target set in our experiments ranged from 8 MB (for $|T| = 10$) to 200MB (for $|T| = 1000$) per keyword. For $w = 100,000$, the storage per keyword ranges from 3 MB (for $|T| = 10$) to 30MB (for $|T| = 10,000$).

To get a larger range of $|T|$ relative to $|V|$, we also perform experiments on the Pokec graph [51] which has 1.6 million nodes and 30 million edges. Figure 6.3 shows the results on Pokec for $w = 100,000$. Here we clearly see the cross-over point where `Monte Carlo` becomes more efficient than `BidirectionalPPR`, while `BiPPR-Precomp-Grouped` and `BiPPR-Precomp-Sampling` consistently take less than 250 milliseconds. On Pokec, the storage used ranges from 800KB for $|T| = 10$ to 3MB for $|T| = 10,000$.

We implement our algorithms in Scala and report running times for Scala, but in preliminary experiments `BiPPR-Precomp-Grouped` is 3x faster when re-implemented in C++, we expect the running time would improve comparably for all five algorithms. Also, we ran each experiment on a single thread, but the algorithms parallelize naturally, so the latency could be improved by a multi-threaded implementation. We ran our experiments on a machine with a 3.33 GHz 12-core Intel Xeon X5680 processor, 12MB cache, and 192 GB of 1066 MHz Registered ECC DDR3 RAM. We measured the running time of the tread running each experiment to exclude garbage collector

with probability $\pi_s(t_i)$; this produces exactly the same distribution of weights as `Monte Carlo` would.

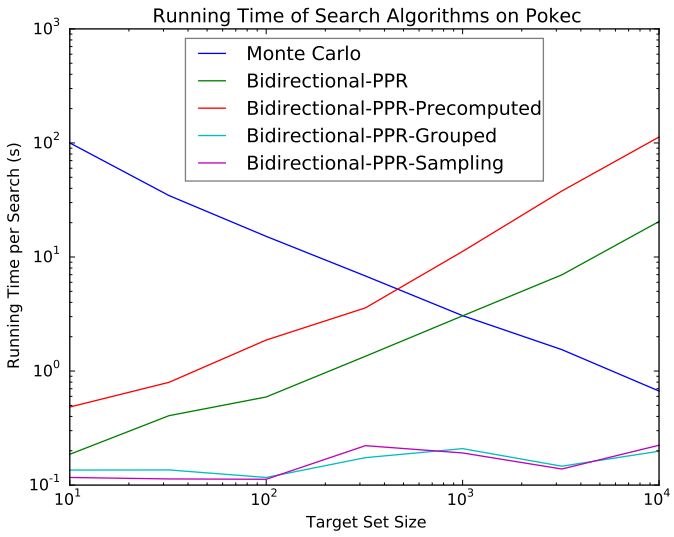


Figure 6.3: Running time on Pokec (30 million edges) performing 100K walks at runtime. Notice that `Monte Carlo` is slow for small $|T|$, `BidirectionalPPR` is slow for large $|T|$, and `BiPPR-Precomp-Grouped` and `BiPPR-Precomp-Sampling` are fast across the entire range of $|T|$.

time. We loaded the graph used into memory and completed any pre-computation in RAM before measuring the running time of the algorithms.

6.2 Sampling Paths Conditioned on Endpoint

6.2.1 Introduction

In this chapter we apply our bidirectional ideas to create an algorithm for sampling a random path from a given source $s \in V$ conditioned on the path ending in a given set of targets $T \subseteq V$ (which of course might be a single target $T = \{t\}$). Our results generalize to any source distribution and Markov Chain based the methods of Chapter 4, and in fact the algorithm is likely simpler for a fixed path length, but in this chapter we present the algorithm for the case of a single source node and for Personalized PageRank (i.e. a geometrically distributed length).

This is a general problem which could be used as a subroutine in other graph

algorithms and could have diverse applications. Our motivation was in personalized search. The method in this chapter can be viewed as an exact version of the approximate target sampler presented in section 6.1.2. In contrast to that sampler, given source s this method samples a target t from a given set T with probability exactly proportional to $\pi_s(t)$. A related application to personalized search is showing a user why a search result came up. For example, if a user searches on Twitter for "Adam" and finds Adam Messinger, this algorithm could generate a few representative paths through the Twitter network from the user to Adam Messinger.

We are not aware of prior work on this problem. Our baseline of comparison is simply Monte Carlo with rejection sampling: we repeatedly sample paths from s until we find one that happens to end in T. The running time of that algorithm is geometrically distributed with expected time $\frac{1}{\pi_s[T]}$ where $\pi_s[T]$ is the probability that a path from s happens to end in T.

Our approach, similar to that of `BidirectionalPPR` (Section 3.2), is to use the `ReversePush` algorithm from the target set to find a larger set of nodes with a significant probability of probability of reaching T on a random walk. However, we modify the `ReversePush` algorithm such that each node v also keeps track of where it receives its residual probabilities from, so from v we can sample a neighbor in proportion to the amount of residual v received from that neighbor.

The running time of our sampler is similar to the running time of `BidirectionalPPR`. Like `BidirectionalPPR` our algorithm has a parameter $r_{\max}$ which lets it trade off reverse work and forward work. The reverse running time is the same as for `BidirectionalPPR` , $O\left(|T|\frac{\bar{d}}{r_{\max}}\right)$ for an average target set T (see Section 3.2.3 for more detail). After executing `ReversePush`, the time per independent path sampled is $O\left(\frac{r_{\max}}{\pi_s[T]}\right)$ or $O(\frac{nr_{\max}}{|T|})$ for a random target set. If we chose $r_{\max}$ to minimize the running time for average target sets, we get $O(\sqrt{m})$ running time to sample a random path, or $O(\sqrt{km})$ time to sample k independent paths.

6.2.2 Path Sampling Algorithm

Algorithm Overview

If we run the `ReversePush` algorithm starting with one unit of residual on each element of T, as in equation (3.2) we get

$$\pi_s(T) = p^T[s] + \sum_{v \in V} \pi_s[v] r^T[v].$$

Our basic approach will to be sample $v \sim \pi_s$ using a random walk, then accept that walk with probability $\frac{r^T[v]}{r_{\max}}$. (If we reject, we keep sampling until we accept.) Once we accept, we sample the remaining path from v to a random target $t \in T$ using information about where each node received residual from to trace the residual back to its source in T.

To enable that, we modify the `ReversePush` algorithm so that each node stores where it received its residual from, as shown in Algorithm 15. Then we can sample a path forward from an intermediate node u towards T in proportion to where residual came from. For example, if a node u received 0.1 units of residual from its outneighbor v_1 and 0.2 units of residual from its outneighbor v_2, then from u we would choose v_1 with probability $\frac{0.1}{0.1+0.2}$. We continue sampling like this until we reach one of the initial residuals that started at each target in T.

One complication in this approach is that a node u might receive residual mass from some node v, push that mass backwards, then later receive new residual mass from another node v'. If we reach u its then unclear with what probability we should go to v vs v'. Our solution to this problem is to store multiple residual vector samplers at u, one for each time u pushes back, plus an additional sampler for any residual mass remaining at u. Then if we reach u when sampling a path, we can transition to the next node based on where u had recieved residual at the relevant time when u pushed back.

An Edge Case

Typically, $p^T[s] = 0$, since $p^T[s]$ is non-zero only for s "close" to t. If $p^T[s] > 0$, then $\pi_s(T) > \alpha r_{\max}$, so simply sampling random walks without using any of these techniques will find a path to T in time $O\left(\frac{1}{r_{\max}}\right)$.

However, if $r_{\max}$ is very small the running time of that method could be significant. To fully handle the case $p^T[s] > 0$ we make two changes to the algorithm. First, when selecting intermediate node v, instead of sampling $v \sim \pi_s$ and accepting with probability $\frac{r^T[v]}{r_{\max}}$, we accept $v = s$ (without doing a walk) with probability

$$\frac{p^T[s]}{p^T[s] + r_{\max}}.$$

We also take a walk to $v \sim \pi_s$ and accept it with probability

$$\frac{r^T[v]}{p^T[s] + r_{\max}}.$$

(As shown in the pseudo-code, we do this in such a way that both events cannot happen.) With the remaining probability we try again for both options. Once we accept, the probability of accepting $v = s$ without a walk is

$$\frac{p^T[s]}{p^T[s] + \sum_v \pi_s[v] r^T[v]} = \frac{p^T[s]}{\pi_s(T)}.$$

Similarly the probability of accepting node v after a walk is $\frac{\pi_s[v] r^T[v]}{\pi_s(T)}$. The second change we make is to allow us to sample residuals of v in the case that we choose $v = s$ without a walk. Since v may push multiple times, and the nodes which gave v residual may be different for different pushes, we store a sampler $P^T[v]$ at each node v which pushes, and when v pushes we add its current residual vector $R^T[v]$ to $P^T[v]$ with weight $r^T[v]$. The complete algorithm is shown as Algorithm 16.

Running Time

Our probability of accepting v on each iteration of the while loop in SampleTarget is

$$\frac{p^T[s] + \sum_v \pi_s[v] r^T[v]}{p[s] + r_{\max}} = \frac{\pi_s(T)}{p^T[s] + r_{\max}}$$

so on average we expect to do

$$\frac{p^T[s] + r_{\max}}{\pi_s(T)} \leq 1 + \frac{r_{\max}}{\pi_s(T)}$$

walks before accepting, and each walk takes constant expected time $O\left(\frac{1}{\alpha}\right)$.

Correctness

We now show that this method is correct by induction on the reverse push operation in `PrecomputePathSamplers`.

Theorem 8. *For any weighted graph G, source node (or distribution) s, target set T, and maximum residual r_{max}, `SamplePathToTarget` (executed on the output of `PrecomputePathSamplers`(T, r_{max})) returns a random walk from s conditioned on ending in T.*

Proof. (Sketch) We prove this by induction on the number of iterations of the while loop of `PrecomputePathSamplers`. As our base case, before the while loop executes, we have $r^T[v] = \mathbb{1}[v \in T]$, so `SamplePathToTarget` will simply do rejection sampling until it finds a walk ending at some node $v \in T$, which is clearly sampled from the right distribution.

Now suppose by induction that we have some (r^T, p^T, R^T, P^T) such that `SamplePathToTarget`$(s, T, r_{\max}, r^T, p^T, R^T, P^T)$ returns a node v from the correct distribution. Let (r'^T, p'^T, P'^T, R'^T) be the residuals and estimates after one more iteration of the while loop in `PrecomputePathSamplers` applied to some node v (i.e. one more reverse push operation on v). Then we have

$$p'^T[v] = p^T[v] + \alpha r^T[v],$$

$$P'^T[v] = P^T[v] \text{ with increased value } \alpha r^T[v] \text{ on } R^T[v],$$

$$r'^T[v] = 0,$$

and for $u \in \mathcal{N}^{in}(v)$

$$\Delta_u = (1-\alpha)\frac{r^T[v]}{d^{out}[u]}$$
$$r'^T[u] = r^T[u] + \Delta_u$$
$$R'^T[u][v] = R^T[u][v] + \Delta_u$$

All other entries of r^T, p, R^T, and P^T are unchanged. Consider how the behavior of `SamplePathToTarget` has changed with this iteration. Before this iteration, there was a probability $\frac{(\pi_s[v]r^t[v])/(p^T[s]+r_{\max})}{(\sum_v \pi_s[v]r^t[v])/(p^T[s]+r_{\max})+p^T[s]/(p^T[s]+r_{\max})} = \pi_s[v]\frac{r^T[v]}{\pi_s(T)}$ that on line 9 we would accept v, and that probability is now 0. However, after this iteration, there is an increased probability of picking some in-neighbor $u \in \mathcal{N}^{in}(v)$ on line 9 and transitioning to v on the first iteration of line 14. The change $\Delta(R[v])$ in the probability of choosing $R[v]$ either on line 9 or line 14 is exactly

$$\begin{aligned}
\Delta(R[v]) &= \frac{\mathbb{1}[s=v]p'[v] + \pi_s[v]r'^T[v]}{\pi_s(T)} + \sum_{u\in\mathcal{N}^{in}(v)} \left(\pi_s[u]\frac{r'^T[u]}{\pi_s(T)}\right)\left(\frac{R'^T[u][v]}{r'^T[u]}\right) \\
&\quad - \frac{\mathbb{1}[s=v]p[v] + \pi_s[v]r^T[v]}{\pi_s(T)} - \sum_{u\in\mathcal{N}^{in}(v)} \left(\pi_s[u]\frac{r^T[u]}{\pi_s(T)}\right)\left(\frac{R^T[u][v]}{r^T[u]}\right) \\
&= \frac{\mathbb{1}[s=v]\alpha r^T[v] - \pi_s[v]r^T[v] + \sum_{u\in\mathcal{N}^{in}(v)} \pi_s[u]\Delta_u}{\pi_s(T)} \\
&= \frac{r^T[v]}{\pi_s(T)}\left(\mathbb{1}[s=v]\alpha - \pi_s[v] + \sum_{u\in\mathcal{N}^{in}(v)} \pi_s[u]\frac{1-\alpha}{d^{out}[u]}\right) \\
&= 0
\end{aligned}$$

The last line follows from a recursive equation for PPR (Equation (11) in [2]):

$$\pi_s[v] = \sum_{u\in\mathcal{N}^{in}(v)} \pi_s[u]\frac{1-\alpha}{d^{out}[u]} + \alpha\mathbb{1}[s=v].$$

Thus the probability of reaching the sampler $R[v]$ on line 14 is unchanged by this iteration. Furthermore, the distribution of paths occurring before reaching the sampler $R[v]$ is unchanged. If $v = s$, then before the push at v with probability $\alpha r^T[v]/\pi_s(T)$ we reach $R[v]$ directly, and after the push the increase in probability of reaching $R[v]$ directly is $p'^T[v] - p^T[v] = \alpha r^T[v]/\pi_s(T)$.

Finally note that the probability of reaching any other sampler $R(v')$ is also unchanged, since before increasing $r_T[u]$ this probability is

$$\frac{r^T[u]}{\pi_s(t)} \frac{R(u, v')}{r^T(u)}$$

which is equal to the probability of reaching sampler $R(v')$ after increasing $r_t[u]$:

$$\frac{r'^T[u]}{\pi_s(t)} \frac{R(u, v')}{r'^T(u)}$$

Thus the distribution of paths returned using (r'^T, p'^T, R'^T, P'^T) is identical to the distribution of paths returned using (r^T, p^T, R^T, P^T). Our induction hypothesis was that distribution of paths returned using (r^T, p^T, R^T, P^T) is the correct distribution, so our proof is complete.

□

Algorithm 15 `PrecomputePathSamplers`$(T, r_{\max})$ [1]

Inputs: Graph G, teleport probability α, target nodes T, maximum residual $r_{\max}$

Initialize (sparse) estimate-vector $p^T = \vec{0}$ and (sparse) residual-vector

$$r^T[v] = \mathbb{1}[v \in T] = \begin{cases} 1 & \text{if } v \in T \\ 0 & \text{else} \end{cases}$$

We will also store a map of samplers R^T to keep track of where each node received its residual from. For each u, $R_t[u]$ is map from samplers to their current weight, where the weight of the sampler associated with node v equals the *current* residual amount u has received from v. Equivalently, $R_t[u]$ is a discrete probability distribution that samples a sampler associated with v with probability proportional to the current residual u has received from v. Note that over the course of the algorithm, if a node u is pushed multiple times, it will create a new sampler each time, and any previous references to its sampler will refer to a previous sampler. Initially each $t \in T$ receives one unit of residual from a constant sampler which always returns t. The current sampler represents the current residual, so we maintain the invariant $r^T[u] = \sum_v R^T[u][v]$.
We also store a map of samplers P^T based on estimates received. For each node v which has pushed at least once, $P^T[v]$ will be a sampler over previous samplers associated with v, with weight equal to the amount of estimate v accumulated during that push. Initially P^T is an empty map. We maintain the invariant $p[u] = \sum_z P^T[u](z)$.

while $\exists v \in V \; s.t. \; r^T[v] > r_{\max}$ **do**
  $\text{sampler}_v = \text{makeImmutableSampler}(R^T[v])$
  $p^T[v] = p^T[v] + \alpha r^T[v]$
  $P^T[v].\text{increaseValue}(\text{sampler}_v, \alpha r^T[v])$
  for $u \in \mathcal{N}^{in}[v]$ **do**
    $\Delta_u = (1-\alpha)\frac{r^T[v]}{d^{out}[u]}$
    $r^T[u] = r^T[u] + \Delta_u$
    $R^T[u].\text{increaseValue}(\text{sampler}_v, \Delta_u)$
  end for
  Update $r^T[v] = 0$
  Update $R^T[v] = \vec{0}$ (An empty map/sampler)
end while
return (r^T, p^T, R^T, P^T)

Algorithm 16 `SamplePathToTarget`$(s, T, r_{\max}, p^T, r^T, R^T)$

Inputs: Graph G, teleport probability α, start node s, target set T, maximum residual $r_{\max}$, residuals, estimates, and residual and estimate maps (r^T, p^T, R^T, P^T) from ContributionsToSet$(T, r_{\max})$

while currentSampler has not been set **do**
  Sample $x \sim$ Uniform$[0, 1]$
  if $x < \frac{p^T[s]}{p^T[s]+r_{\max}}$ **then**
    Set currentSampler = P[s].sample().
    Set walk $W = [s]$ (i.e. a length-1 walk)
  else
    Sample a walk W from s, and let u be its last node.
    if $x - \frac{p^T[s]}{p^T[s]+r_{\max}} < \frac{r^T[u]}{p^T[s]+r_{\max}}$ **then**
      Set $currentSampler = R[u]$
    end if
  end if
end while
while currentSampler is not a ConstantSampler **do**
  currentSampler = currentSampler.sample()
  Append node of currentSampler to W
end while
return W

Chapter 7

Conclusion and Future Directions

In this chapter, we briefly recap the contributions in this thesis, and then describe a number of interesting research problems that could lead to nice publications.

7.1 Recap of Thesis

In this thesis we presented new algorithms for estimating random walk scores, including personalized PageRank. While past algorithms either use linear algebra or sample random walks, we achieve dramatic speed improvements by combining local liner algebra operations from the target node backwards with random walks from the source node forwards. Our algorithm gives provably accurate estimates, and on six diverse graphs it is 70x faster than any past algorithm at estimating individual PPR scores. For estimating a single value $\pi_s[t]$, `BidirectionalPPR` takes average time $O(\sqrt{m}/\epsilon)$ for constant failure probability, and [40] includes a lower bound of $\Omega(\sqrt{n})$, so there is little asymptotic improvement to be made on sparse graphs for random targets. However, on the Twitter-2010 graph, `BidirectionalPPR` still takes several seconds to compute scores for the most popular targets, and this motivates pre-computing the reverse linear algebra (residuals) and forward walks. That pre-computation motivates several problems described in the next section.

We then observed that on undirected graphs, rather than do linear algebra backwards from the target and random walks forwards, it is possible to do random walks

backwards from the target and linear algebra forwards. This alternative algorithm allows for cleaner worst-case bounds and may be useful in future applications. There is also an interesting symmetry between `BidirectionalPPR` and this alternative algorithm, and there may be applications which can exploit that symmetry.

We also observed that our bidirectional algorithm does not require the memoryless property of Personalized Pagerank walks, but generalizes naturally to estimating random walk probabilities of any given length on any weighted graph (or Markov Chain). This algorithm can be used for example, to estimate the heat kernel score, and we hope future applications of Markov Chains find use for this algorithm. Natural extensions of this algorithm to other random walk quantities like hitting time would be interesting.

We then applied these algorithms to describe new personalized search ranking algorithms. In particular, we described an algorithm which pre-groups targets by name and allows the scores of all targets matching a given name query to be efficiently computed, even when the set of nodes matching the name query is large. We also described a sampling algorithm which can find the most relevant nodes without needing to score them all. Note that these two algorithms are not restricted to estimating PPR, but could be applied to personalized search with any scoring function which can be written as a dot-product between some vector associated with the searching user and some vector associated with each document. For example, the vectors could arise from some machine learning or latent semantic analysis algorithm.

This is an exciting new capability, but we believe there is still significant improvements to be made. In particular, it may be possible to decrease the amount of storage needed, especially in the case of text search when each target is relevant to multiple keywords. Our methods also require the potential queries to be known in advance.

7.2 Open Problems in PPR Estimation

7.2.1 The Running-Time Dependence on $\bar{d}$

The average-case running time of `BidirectionalPPR` to detect a PPR value of size $\delta = \frac{1}{n}$ to constant relative error with constant failure probability is $O(\sqrt{m})$. However, the average-case lower bound presented in [40] is $\Omega(\sqrt{n})$. This leaves a gap of $\sqrt{\bar{d}} = \sqrt{m/n}$ between the achieved running time and the lower bound. Can the average-case running time be improved to $O(\sqrt{n})$, or alternatively does there exist a graph where for random (s, t) pairs, $\Omega(\sqrt{m})$ edges must be looked at on average to estimate a PPR value of size $\frac{1}{n}$?

7.2.2 Decreasing Pre-computation Storage or Running Time

When we precompute all the residual vectors r^t from every target in order to make real-time queries faster, the storage is significant, $O(n\sqrt{m}/\epsilon)$. Is there some way to decrease the amount of storage required without increasing the running time at query time?

In Section 5.2 we presented a method to to combine random walks from the out-neighbors of s to decrease the number of stored walks. One approach to this problem is to find an analogous way to combine the residual vectors from different nodes to create the residual vector for the given target t at runtime.

7.2.3 Maintaining Pre-computed Vectors on Dynamic graphs

Suppose we precompute the residual vectors r^t for every target t to enable real-time PPR estimation or for personalized search. How can we update these residual vectors as edges are added to the graph?

For the forward walks, there is a natural algorithm for maintaining the walks as the graph evolves [7]. Something similar might work for maintaining the residuals. In particular, when an edge (u, v) is added, we can compute how much excess residual u has received from other out-neighbors, as well as how much residual u should have received from v, and from this restore the loop invariant of `ReversePush` (Section

2.3), at which point push operations can restore the requirement that for all $t, u \in V$, $|r^t[u]| \leq r_{\max}$.

7.2.4 Parameterized Worst-Case Analysis for Global PageRank

For a fixed choice of δ, we give an average-case running time analysis for `BidirectionalPPR`, but no worst-case running time analysis. Is there a variation of `BidirectionalPPR` or some assumption on the graph for which there is a worst-case running time bound, perhaps parameterized by some property of the target like its global PageRank?

One approach is to use a sequence of δ values, say $\delta_i = 2^{-i}$ and try progressively smaller values of δ until the estimate returned is greater than $(1 + 2e)\delta$, as needed by Theorem 1 to guarantee a small relative error. The heuristic parameterized estimate in Section 3.4.2 may also be relevant.

7.2.5 Computing Effective Resistance or Hitting Times

As briefly sketched in Section 4.4, our random walk methods can be used to estimate other random walk scores. As defined for example in these lecture notes[1], the hitting time between two nodes s and t is the expected number of steps before a random walk from s first reaches t. On an undirected graph, the effective conductance between a and b is d_a times the probability that a random walk from a reaches b before returning to a, and the effective resistance between s and t is the inverse of this. These scores have applications in the analysis of networks. It would be interesting to apply bidirectional techniques to estimating them and compare the running time of the resulting algorithm with the best previous algorithms.

[1] http://web.stanford.edu/class/cme305/Notes/6.pdf

7.3 Open Problems in Personalized Search

7.3.1 Decreasing Storage for Text Search

For text search, for each target t, the algorithm in Section 6.1.2 requires the associated residual r^t to be included in a data-structure for every word t is related to. For example, there is a data structure incorporating $\{r^t : t \text{ is relevant to "computer"}\}$ and one incorporating $\{r^t : t \text{ is relevant to "science"}\}$, so a target relevant to both "computer" and "science" will have its entire residual vector included in both. Is there some way to not include r^t in the data structure associated with every keyword for which t is relevant.

7.3.2 Handling Multi-Word Queries

In Chapter 6, we assume that a single keyword is given, for example "Adam" or "computer" but not both. If two keywords are given, our algorithm is useful in some cases. In particular, if one keyword is likely conditioned on the other, for example if 10% of all people named "Adam" are relevant to "computer," we can use rejection sampling. In this example, we could find the top 30 people named Adam, reject those not relevant to "computer," and we would have the top 3 people matching "Adam" and "computer." However, there is some efficiency loss in this algorithm, and it requires the keywords to be correlated, so an interesting open problem is finding an algorithm for PPR search which supports multiple keywords or more general queries.

7.3.3 User Study of Personalized Search Algorithms for Social Networks

While there has been past empirical work measuring the improvement in relevance for topic-based Personalized PageRank [27], history based personalized search [17], and social network search based on shortest path [55], we are not aware of any published evaluation of personalized PageRank for ranking name or text search queries on a social network. This experiment for example could be done on Twitter, Facebook, or

Google+. Participants could choose common names to search for (or be given common names to search for), shown results using different algorithms (either interleaved or side-by-side), and note which results are most relevant. Or a live A/B test could be done on a social network site, and implicit feedback based on clicks could be used (see the chapter on "Evaluation in information retrieval" in [41] for more background). Natural ranking strategies to compare could include global PageRank (or in-degree), shortest path distance, and personalized PageRank.

Bibliography

[1] R. Andersen, C. Borgs, J. Chayes, J. Hopcraft, V. S. Mirrokni, and S.-H. Teng. Local computation of pagerank contributions. In *Algorithms and Models for the Web-Graph*. Springer, 2007.

[2] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, 2006.

[3] K. B. Athreya and Ö. Stenflo. Perfect sampling for doeblin chains. *Sankhyā: The Indian Journal of Statistics*, pages 763–777, 2003.

[4] K. Avrachenkov, P. Gonçalves, and M. Sokol. On the choice of kernel and labelled data in semi-supervised learning methods. In *Algorithms and Models for the Web Graph*, pages 56–67. Springer, 2013.

[5] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Journal on Numerical Analysis*, 2007.

[6] L. Backstrom and J. Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*. ACM, 2011.

[7] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment*, 2010.

[8] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar, D. Ravichandran, and M. Aly. Video suggestion and discovery for youtube: taking random walks through the view graph. In *Proceedings of the 17th international conference on World Wide Web*, pages 895–904. ACM, 2008.

[9] S. Banerjee and P. Lofgren. Fast bidirectional probability estimation in markov models. In *NIPS*, 2015.

[10] P. Berkhin. Bookmark-coloring algorithm for personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, 2006.

[11] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World Wide Web*. ACM Press, 2011.

[12] C. Borgs, M. Brautbar, J. Chayes, and S.-H. Teng. Multi-scale matrix sampling and sublinear-time pagerank computation. *Internet Mathematics*, 2013.

[13] D. Carmel, N. Zwerdling, I. Guy, S. Ofek-Koifman, N. Har'El, I. Ronen, E. Uziel, S. Yogev, and S. Chernov. Personalized social search based on the user's social network. In *Proceedings of the 18th ACM conference on Information and knowledge management*, pages 1227–1236. ACM, 2009.

[14] S. Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *Proceedings of the 16th international conference on World Wide Web*, pages 571–580. ACM, 2007.

[15] F. Chung. The heat kernel as the pagerank of a graph. *Proceedings of the National Academy of Sciences*, 104(50):19735–19740, 2007.

[16] W. Doeblin. Elements d'une theorie generale des chaines simples constantes de markoff. In *Annales Scientifiques de l'Ecole Normale Supérieure*, volume 57, pages 61–111. Société mathématique de France, 1940.

[17] Z. Dou, R. Song, and J.-R. Wen. A large-scale evaluation and analysis of personalized search strategies. In *Proceedings of the 16th international conference on World Wide Web*, pages 581–590. ACM, 2007.

[18] D. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, 2009.

[19] D. P. Dubhashi and A. Panconesi. *Concentration of measure for the analysis of randomized algorithms*. Cambridge University Press, 2009.

[20] D. Fogaras and B. Rácz. Towards scaling fully personalized pagerank. In *Algorithms and Models for the Web-Graph*, pages 105–117. Springer, 2004.

[21] D. Fogaras, B. Rácz, K. Csalogány, and T. Sarlós. Towards scaling fully personalized pagerank: Algorithms, lower bounds, and experiments. *Internet Mathematics*, 2005.

[22] D. F. Gleich. PageRank beyond the web. *arXiv*, cs.SI:1407.5107, 2014. Accepted for publication in SIAM Review.

[23] O. Goldreich and D. Ron. On testing expansion in bounded-degree graphs. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*. Springer, 2011.

[24] V. Grolmusz. A note on the pagerank of undirected graphs. *Information Processing Letters*, 2015.

[25] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22nd international conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2013.

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

[27] T. H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, pages 517–526. ACM, 2002.

[28] G. Jeh and J. Widom. Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*. ACM, 2003.

[29] S. Kale, Y. Peres, and C. Seshadhri. Noise tolerance of expanders and sublinear expander reconstruction. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*. IEEE, 2008.

[30] H.-r. Kim and P. K. Chan. Personalized search results with user interest hierarchies learnt from bookmarks. In *Advances in Web Mining and Web Usage Analysis*, pages 158–176. Springer, 2006.

[31] K. Kloster and D. F. Gleich. Heat kernel based community detection. In *ACM SIGKDD'14*, 2014.

[32] D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd Edition*. Reading, Mass.: Addison-Wesley, 1998.

[33] Laboratory for web algorithmics. http://law.di.unimi.it/datasets.php. Accessed: 2014-02-11.

[34] C. E. Lee, A. Ozdaglar, and D. Shah. Computing the stationary distribution locally. In *Advances in Neural Information Processing Systems*, pages 1376–1384, 2013.

[35] R. Lempel and S. Moran. The stochastic approach for link-structure analysis (salsa) and the tkc effect. *Computer Networks*, 33(1):387–401, 2000.

[36] P. Lofgren, S. Banerjee, and A. Goel. Bidirectional pagerank estimation: From average-case to worst-case. In *Workshop on Algorithms and Models for the Web Graph (WAW)*, 2015.

[37] P. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. FAST-PPR: Scaling personalized PageRank estimation for large graphs. In *ACM SIGKDD'14*, 2014.

[38] P. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. Personalized pagerank estimation and search: A bidirectional approach. Technical report, Stanford University, 2015. available at http://cs.stanford.edu/~plofgren/.

[39] P. Lofgren and A. Goel. Personalized pagerank to a target node. *arXiv preprint arXiv:1304.4658*, 2013.

[40] P. A. Lofgren, S. Banerjee, A. Goel, and C. Seshadhri. Fast-ppr: Scaling personalized pagerank estimation for large graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, pages 1436–1445, New York, NY, USA, 2014. ACM.

[41] C. D. Manning, P. Raghavan, H. Schütze, et al. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[42] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and Analysis of Online Social Networks. In *Proceedings of the 5th ACM/Usenix Internet Measurement Conference (IMC'07)*, San Diego, CA, October 2007.

[43] R. Motwani, R. Panigrahy, and Y. Xu. Estimating sum by weighted sampling. In *Automata, Languages and Programming*, pages 53–64. Springer, 2007.

[44] S. Negahban, S. Oh, and D. Shah. Iterative ranking from pair-wise comparisons. In *Advances in Neural Information Processing Systems*, pages 2474–2482, 2012.

[45] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[46] P. Sarkar, A. W. Moore, and A. Prakash. Fast incremental proximity search in large graphs. In *Proceedings of the 25th international conference on Machine learning*, pages 896–903. ACM, 2008.

[47] A. D. Sarma, A. R. Molla, G. Pandurangan, and E. Upfal. Fast distributed pagerank computation. *Distributed Computing and Networking*, 2013.

[48] A. Shrivastava and P. Li. Asymmetric lsh (alsh) for sublinear time maximum inner product search (mips). In *Advances in Neural Information Processing Systems*, pages 2321–2329, 2014.

[49] A. Shrivastava and P. Li. Improved asymmetric locality sensitive hashing (alsh) for maximum inner product search (mips). *stat*, 1050:13, 2014.

[50] M. Speretta and S. Gauch. Personalized search based on user search histories. In *Web Intelligence, 2005. Proceedings. The 2005 IEEE/WIC/ACM International Conference on*, pages 622–628. IEEE, 2005.

[51] Stanford network analysis platform (snap). http://http://snap.stanford.edu/. Accessed: 2014-02-11.

[52] J. Steinhardt and P. Liang. Learning fast-mixing models for structured prediction. In *ICML'15*, 2015.

[53] L. Takac and M. Zabovsky. Data analysis in public social networks. In *International. Scientific Conf. & Workshop Present Day Trends of Innovations*, 2012.

[54] H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. 2006.

[55] M. V. Vieira, B. M. Fonseca, R. Damazio, P. B. Golgher, D. d. C. Reis, and B. Ribeiro-Neto. Efficient search ranking in social networks. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. ACM, 2007.

[56] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw.*, 3(3):253–256, Sept. 1977.

[57] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, page 3. ACM, 2012.