Fast Algorithms for Top-k Personalized PageRank Queries

Manish Gupta
IIT Bombay
manishg@cse.iitb.ac.in

Amit Pathak
IIT Bombay
amit@cse.iitb.ac.in

Soumen Chakrabarti
IIT Bombay
soumen@cse.iitb.ac.in

ABSTRACT

In entity-relation (ER) graphs (V, E), nodes V represent typed entities and edges E represent typed relations. For dynamic personalized PageRank queries, nodes are ranked by their steady-state probabilities obtained using the standard random surfer model. In this work, we propose a framework to answer top-k graph conductance queries. Our top-k ranking technique leads to a $4\times$ speedup, and overall, our system executes queries 200–1600× faster than whole-graph PageRank. Some queries might contain hard predicates i.e. predicates that must be satisfied by the answer nodes. E.g., we may seek authoritative papers on public key cryptography, but only those written during 1997. We extend our system to handle hard predicates. Our system achieves these substantial query speedups while consuming only 10–20% of the space taken by a regular text index.

Categories and Subject Descriptors: H.3.1[**Information Systems**]:Information Storage and Retrieval –*Content Analysis and Indexing*; H.3.3[**Information Systems**]:Information Search and Retrieval

General Terms: Algorithms, Experimentation, Measurement, Performance

Keywords: top-k, Pagerank, HubRank, Node-deletion, personalized

1. INTRODUCTION AND RELATED WORK

Graph proximity queries of the form "entities related to a *set of keywords*" can be answered by searching for answer nodes in the vicinity of nodes matching the keywords. Such graph conductance queries assume that a keyword-originated prestige starts at the nodes matching the query words and flows through the graph following edges, and the rank of a node depends on the amount of prestige that reaches it. There are different techniques proposed in literature to compute search rankings; one of which is HubRank [3]. HubRank builds on Personalized Pagerank [4] and Berkhin's [1] Bookmark Coloring Algorithm (BCA).

Consider a graph $G = (V, E)$ where each edge $(u, v) \in E$ is associated with a *conductance* $C(v, u)$: probability of a "random surfer" walking from u to v; $\sum_v C(v, u) = 1$. Personalized PageRank Vector (PPV) for a teleport vector (r) is then defined as:

$$p_r = \alpha C p_r + (1 - \alpha) r = (1 - \alpha)(\mathbb{I} - \alpha C)^{-1} r \qquad (1)$$

where $1 - \alpha$ is teleport probability (typically 0.2–0.25). r is set such that $r(u) > 0$ only if u is a match node and $\sum_u r(u) = 1$. When $r(u) = 1$ for a single node u, we call its PPV as PPV_u. Given a hubset (H) of nodes with precomputed $PPV_h \; \forall h \in H$, we can use Berkhin's [1] asynchronous push to compute p_r for a general r as shown in algorithm 1.

2. BASIC TOPK FRAMEWORK

For ad-hoc search applications, it is adequate to report the answer nodes with the top-k personalized PageRank values. At some time during the execution of algorithm 1, let $u_1, u_2, \cdots$ be the nodes sorted in non-increasing order of their scores $(\hat{p}_r)$ then by proposition 1, we can say that $u_1, u_2, \cdots, u_k$ are the best k answer nodes iff $\hat{p}_r(u_k) \geq \hat{p}_r(u_{k+1}) + \parallel q \parallel_1$. This is the basic idea behind early termination of the algorithm while guaranteeing top-k answers.

PROPOSITION 1. *In algorithm 1, at any time, $\forall$ nodes u,*
$$\hat{p}_r(u) \leq p_r(u) \leq \hat{p}_r(u) + \parallel q \parallel_1$$

In search applications, if the user asked for $\underline{K} = 20$ responses, it is typically acceptable if the system returns a few more (say $\overline{K} = 40$). So, we propose that the number of responses be bracketed in $[\underline{K}, \overline{K}]$. This substantially increases the success rate for termination check. About half the queries terminate through algorithm 2. Also, the actual rank K^* at which the termination check succeeds is typically very close to $\underline{K}$. Further, we refine proposition 1 as:

PROPOSITION 2. *In algorithm 1, at any time, $\forall$ nodes u,*
$$\hat{p}_r(u) \leq p_r(u) \leq \hat{p}_r(u) + (1 - \alpha)^2 q(u) + \alpha \parallel q \parallel_1$$

Algorithm 1 Basic Push Algorithm

```
1: $q \leftarrow r, \hat{p}_r \leftarrow \vec{0}$
2: **while** $\|q\|_1 > \epsilon_{\text{push}}$
3:    pick node $u$ with largest $q(u) > 0$ {delete-max}
4:    Node deletion check {Only for Delete-Push; Alg. 3}
5:    $\hat{q} \leftarrow q(u)$, $q(u) \leftarrow 0$
6:    **if** $(u \in H)$ $\hat{p}_r \leftarrow \hat{p}_r + \hat{q}\,\text{PPV}_u$
7:    **if** $(u \notin H)$
8:       $\hat{p}_r(u) \leftarrow \hat{p}_r(u) + (1 - \alpha)\hat{q}$
9:       **for** each out-neighbor $v$ of $u$
10:         $q(v) \leftarrow q(v) + \alpha C(v, u)\hat{q}$ {increase-key}
11:    top-$k$ quit check {For basic topK+DeletePush;Alg. 2}
12: **return** $\hat{p}_r$
```

Maximum increase in $\hat{p}_r(u)$ due to flow of $q(u)$ that can happen (till push-algorithm termination) is $(1 - \alpha)q(u) + \alpha^2 q(u)$. Similarly, maximum increase in $\hat{p}_r(u)$ due to flow of residual from nodes in V other than u is $\alpha(\parallel q \parallel_1 - q(u))$. Thus, $\hat{p}_r(u)$ can increase atmost by $(1-\alpha)q(u) + \alpha^2 q(u) + \alpha(\parallel q \parallel_1 - q(u)) = (1 - \alpha)^2 q(u) + \alpha \parallel q \parallel_1$. However, this better upper bound is dependent on $q(u)$ and so we will need to maintain lower and upper bounds separately as against in proposition 1. So, we relax the bound as:

PROPOSITION 3. *In algorithm 1, at any time, $\forall$ nodes u,*
$$\hat{p}_r(u) \leq p_r(u) \leq \hat{p}_r(u) + (1 - \alpha)^2 \max_u q(u) + \alpha \parallel q \parallel_1$$

Algorithm 2 Top-k termination check for Basic Push

```
1:initialize an empty min-heap $M$
2: **for** each node $u$ in the score map $\hat{p}_r$
3:    **if** $(|M| < \overline{K})$   insert $u$ into $M$
4:    **else**
5:       let $v \in M$ have the smallest score
6:       **if** $(\hat{p}_r(u) > \hat{p}_r(v))$   replace $v$ with $u$
7: Let $u_1, \cdots, u_{\overline{K}} \in M$ such that $\hat{p}_r(u_1) \geq \cdots \geq \hat{p}_r(u_{\overline{K}})$
8: **for** $b = \underline{K} + 1, \ldots, \overline{K}$
9:    **if** $(\hat{p}_r(u_b) > \hat{p}_r(u_{b+1}) + \|q\|_1)$
10:         **return** can terminate push loop from line 2 to
11:               line 11 of algorithm 1 with $K^* = b$
12: **return** cannot yet terminate push loop
```

Proposition 3 needs less book-keeping than proposition 2. These better bounds provide a 6% reduction in query time without any change in accuracy; more queries quit earlier and at lower K^*.

3. HARD PREDICATES

Consider the query "find top-k papers related to XML published in 2008". This enforces that the answer nodes returned should be papers, published in 2008. In this section, we extend basic top-k framework to answer queries with such hard predicates, efficiently. The target nodes returned as answer nodes, should strictly satisfy the hard predicates. One of the ways is to modify "basic top-k for soft predicate queries", such that a node is considered to be put in heap M (see algorithm 2) only if it belongs to target set. We call this as ***naiveTopk***. We propose a node deletion algorithm which builds on the idea that ranking non-target nodes is not needed in presence of hard predicates. It can delete nodes without affecting authority flow in the remaining graph. We then use it to delete non-target nodes in the graph while executing push.

3.1 Node Deletion Algorithm

Let V contain a special sink node s with self-loop of $C(s,s) = 1$. Aim of node deletion algorithm is to delete a node u from graph G and adapt the graph structure to create $G' = (V', E')$ such that for any teleport $r'^{|V'|\times 1}$ over G', $p'_{r'}(v) = p_r(v)$ for all nodes $v \in V' - s$ where $p'_{r'}(v)$ is computed over G', $r(v) = r'(v)$ for $v \in V'$ and $r(v) = 0$ for $v \notin V'$.

Let u be a node to be deleted, v be one of the in-neighbors of u and w be one of the out-neighbors of u. Let $q(v)$ be the residual at node v at some time instant during execution of push algorithm. Consider the simple case when u does not have a self-loop. v passes on $\alpha q(v)C(u,v)$ to u then u keeps $(1-\alpha)\left[\alpha q(v)C(u,v)\right]$ with itself and passes on $\alpha\left[\alpha q(v)C(u,v)\right]C(w,u)$ to w. This can be achieved by increasing $C(w,v)$ by $\alpha C(u,v)C(w,u)$. To consider the self endorsement by u, $(1-\alpha)\left[\alpha q(v)C(u,v)\right]$ is grounded by sending it to sink node by increasing $C(s,v)$ by $(1-\alpha)C(u,v)$. Algorithm 3 considers the case with self-loop at node u.

Algorithm 3 Node Deletion Algorithm

```
1: Input:Node $u$ to be deleted
2: for each in-neighbor $v$ of $u$
3:    if $(u = v)$ continue
4:    for each out-neighbor $w$ of $u$
5:       if $(u = w)$ continue
6:       $C(w,v) \leftarrow C(w,v) + \frac{\alpha C(u,v) * C(w,u)}{1-\alpha C(u,u)}$
7:       $C(w,u) \leftarrow 0$ {Delete $u \rightarrow w$ edge}
8: $C(s,v) \leftarrow C(s,v) + (C(u,v) * (1-\alpha)(1 + C(u,u)))$
9: $C(u,v) \leftarrow 0$ {Delete $v \rightarrow u$ edge}
```

The algorithm takes $O(inDegree(u) \times outDegree(u))$ time and can potentially increase #edges by $inDegree(u) \times outDegree(u) - (inDegree(u) + outDegree(u))$.

3.2 Ranking only target nodes (DeletePush)

While performing DeletePush, we first pick a node u (step 3 of algorithm 1), delete all the "deletable" non-target entity nodes reachable from u (step 4 of algorithm 1) and then perform push from u. Deleting a non-target node avoids any further pushes from it; thereby saving some work. A single node deletion can bloat #edges, so we need to judiciously pick the victim nodes (non-target entities) to be deleted, keeping in mind these observations about social networks.

1. Block structure phenomena [5], observed in social networks, partitions the graph into clusters, ensuring that edges to be added due to a node deletion already exist in the graph.

2. The indegree and outdegree of the nodes in the graph follow power law [2]. Since large number of nodes have very small indegree or outdegree they can be deleted safely.

As we do not want a large amount of time to be spent in node-deletion, we use a conservative approach where we do a local search (as push may get terminated before authority flows to far-off nodes) in the (out)neighborhood of node u and delete a non-target non-hubset node only if its deletion does not blowup number of new edges.

4. EXPERIMENTS

We used a 1994 snapshot of CiteSeer graph which has 74000 nodes and 289000 edges with a 55 MB text index. $1.9M$ CiteSeer queries have an average of 2.68 words per query. All but the first 100K queries were used to train and tune our index. We fixed $[\underline{K}, \overline{K}]$ as $[20, 40]$. We used samples of typical size 10000 from the first 100000 queries as test data and hubset of size 15000 generated using "naive one-shot" hub inclusion policy [3]. For DeletePush expts, we selected slowest 1000 queries from the above sample.

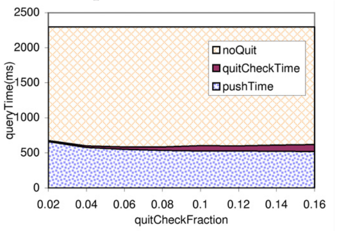

Figure 1: Push times averaged across queries vs. fraction of push time allowed in termination checks.(The top line uses no termination checks.)

As figure 1 shows, algorithm 2 is fast and effective. Quit checks take a very small amount of time and typically give a 4× speed boost. (To control the fraction of time spent in quit checks, here we timed recent quit checks and invoked them only when enough time has been spent in push loops.) Also reassuring is that as little as 4% time invested in quit checks result in robust gains.

Now, let us compare DeletePush with NaiveTop-k for hard predicates. We varied target set size by having different hard predicates on publication years. Note that the time required by DeletePush does not decrease in proportion with the decrease in number of pushes because of deletion overheads. Figure 2 shows that DeletePush works better when the target set sizes are not too large. Thus, by applying a top-k framework over the basic push algorithm, we try to efficiently answer graph conductance queries with hard predicates, achieving better query processing times with low indexing space.

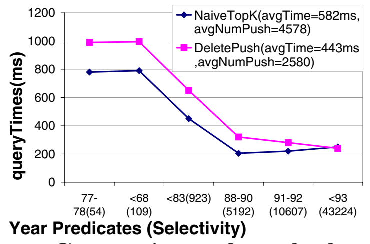

Figure 2: Comparison of *top-k* algorithms

5. REFERENCES

[1] P. Berkhin. Bookmark-coloring approach to personalized pagerank computing. *Internet Mathematics*, 3(1):41–62, Jan. 2007.

[2] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the web. *Computer Networks*, 33(1-6):309–320, 2000.

[3] S. Chakrabarti. Dynamic personalized PageRank in entity-relation graphs. In *www*, Banff, May 2007.

[4] G. Jeh and J. Widom. Scaling personalized web search. In *WWW Conference*, pages 271–279, 2003.

[5] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Exploiting the block structure of the web for computing, Mar. 12 2003.