TopK conductance queries for Entity Relationship graphs

M. Tech Dissertation

Submitted in partial fulfillment of the requirements
for the degree of

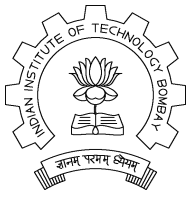
Master of Technology

by

Manish Gupta
Roll No: 05305006

under the guidance of

Dr. Soumen Chakrabarti



Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai

Acknowledgements

I would like to thank my guide, **Dr. Soumen Chakrabarti**, for spending his valuable time with me on the project. Apart from introducing me to the field of IR, he suggested various ideas, helped me when I was stuck at various points and even coded. He was not just a guide for me, but a good friend too. I would like to thank the IITB and the TCS for providing me with the infrastructure and the financial support for my stay here. I would like to thank my colleague, Amit Pathak for fruitful discussions and for accompanying me throughout the project. I would like to thank my friends at IITB for their continuous support throughout my stay at IITB; with special thanks to Krishna Prasad and my lab-mates. Neha, Ritesh and Pari really helped me a lot in keeping me refreshed always. I would like to thank both the CSE and the KReSIT sysads for bearing with me, when my experiments used to hog the memory and the CPU on their servers. I would like to thank my family members, especially my mother and my sister Shalini, who kept my enthusiasm high towards the work, throughout my MTech course duration. Last and obviously not the least, I would like to thank the Almighty for creating conditions in which I could successfully complete my MTech course.

Manish S. Gupta
Date: January 14, 2008
Place: IITB, Mumbai

Abstract

Popularity of the object oriented paradigm has ensured the prevalence of entity-relationship graphs in various fields. Providing proximity search on ER graphs is therefore, of importance. We provide a methodology for graph conductance based proximity search on ER graphs. Proximity is evaluated using entity scores based on steady-state distributions of random walks with restarts, considering the ER graph as a Markovian network. This helps us answer queries like "Entities near keywords biochemistry, Information, Retrieval". We first introduce a conductance-based algorithm - the Bookmark coloring Algorithm which performs pushes using a special hubset of nodes for which steady-state ditributions have been pre-computed and stored and then propose a topK-based termination of the algorithm. Our search system is both time-wise and space-wise efficient and almost as accurate as the ObjectRank system, which we use as a baseline. We show this, via our experiments on Citeseer graphs of sizes 74K and 320K nodes.

We explain the need of approximating some of the Personalized PageRank vectors with destination distributions obtained by running multiple random walks originating from those nodes, known as fingerprints. These ditributions approach the PPVs if the number of random walks are large. So, we present some of the schemes for allocating number of random walks for fingerprint computation viz. Saturate-Fill, Linear, Squared, Cohen-based, Uniform. We then justify the usage of a hybrid hubset which consists of word PPVs and entity FPs.

The modified conductance-based algorithm can be used to compute directed proximity from a node 'a' to a node 'b' in the ER graph. This is defined as the probability that a random walk started from node 'a' reaches node 'b' before reaching node 'a'. We also present ways to find topK most proximal nodes to a particular node.

In many applications, the ER graph keeps changing(nodes get added/deleted, edge weights change). We propose an algorithm to update the above forms of proximity, given perturbations to the graph conductance matrix. We also propose a way to use this update algorithm to compute Personalized PageRank vectors faster.

Contents

List of Figures

Chapter 1

Ranking for ER graphs

1.1 Introduction

Entity relationship graphs are widely applicable in a variety of fields. In ER graphs, nodes represent entities while the edges represent relationships among these entities. e.g. in an email graph, a node can be a person, an email or a company while the edges could denote relationships like "wrote" (between a person and an email), "works–for" between a person and a company etc. Another example could be that of a citation graph where each node could be a paper or an author while the edges could represent relationships like "cites", "is_cited_by", "wrote", "is_written_by". These graphs could be bidirectional or uni-directional. Considering the popularity of these graphs, it is essential to provide proximity search on them so that we can answer queries like "Entities near keywords biochemistry, Information, Retrieval". A particular node in the graph may not have all these keywords exactly, but might still be important with respect to these keywords because it is linked from many nodes that contain them. Ideally, for the above query, a user would expect as the result – a 'paper node' representing a paper related to the application of information retrieval techniques in biochemistry or an 'author node' representing an author who has written papers both in biochem and in IR.

We expect that the answer nodes returned should be such that

1. they are close to the nodes that contain exact matches
2. they lie on large number of paths emanating from nodes with exact matches.

This intuition led to various ranking algorithms.

1.2 Related work

1.2.1 PageRank

PageRank [PBMW98] finds all documents containing the query words, ranks them based on the graph structure (irrespective of the query words) and returns the ranked results. In PageRank , with a probability equal to "teleport prob", the random surfer jumps to any of the web pages in the entire web graph uniformly at random. Thus PageRank vector depends only on the teleport probability and the graph structure. It is a democratic score that has no preference for any particular pages. PageRank is computed as

$p_r = \alpha C p_r + (1 - \alpha) r.$

where p_r is the PageRank vector, α is the teleport probability, C is the conductance matrix and r is the teleport vector.

1.2.2 ObjectRank

ObjectRank [BHP04] uses either AND or OR semantics to rank documents using the random surfer model. The random surfer walks as in PageRank but random jumps are now restricted to a subset of the nodes of the graph structure. This subset contains those nodes which contain either all the query words(AND semantics) or atleast one of the query words(OR semantics). Thus, as against PageRank , ObjectRank can return pages which may not contain the query words per se but are still high ranked because they are linked from nodes containing those words. ObjectRank is not personalized i.e. it does not consider the notion of a preference set. In ObjectRank , the random surfer jumps randomly to any of the web pages containing either or all of the query words uniformly at random.

1.2.3 Personalized PageRank

In Personalized PageRank [JW03], the random surfer jumps randomly to any of the web pages included in the user's personalized set S of web pages uniformly at random. The score vector that we get is called PPV(Personalized PageRank Vector) with respect to the set S. While PageRank provides importance of a webpage with respect to graph structure, Personalized PageRank provides importance of a webpage with respect to structure of the graph; the ranking being biased towards a set of preferred pages. In Personalized PageRank computation, linearity theorem helps us to express the PPV for a preference set vector in terms of a linear combination of the PPVs for basis vectors. To compute PPV for a basis vector corresponding to a particular node faster, we express it in terms of the PPVs of its neighbors and the basis vector itself using Decomposition theorem. Thus Personalized PageRank provides a ranking vector for a particular user with respect to his preferred pages. But it does not incorporate query at all while computing this PPV.

Linearity theorem:

PageRank is given by $p_r = (1-\alpha)(\mathbb{I} - \alpha C)^{-1} r$.

The inverse $(\mathbb{I} - \alpha C)^{-1}$ exists because C is column stochastic and so columns of αC add up to less than 1.

Further, $p_r = (1-\alpha)(\sum_{k>=0} \alpha^k C^k) r$.

From the above, we see that p_r is linear in r.

Therefore, $p_{\gamma r} = \gamma p_r$ and also $p_{r_1+r_2} = p_{r_1} + p_{r_2}$ for any scalar γ.

If we were to use linearity theorem, to compute PPVs with respect to all the words, then this would potentially take $O(\mid V \mid^2)$ space, where V is the size of the vocabulary.

On the 320K node graph which has a vocabulory size of 562,000 words, this would take 22,000 CPU-hours to precompute all the PPVs. If PPVs are not pre-computed, online ObjectRank returns an error message and computes them offline. Also, storing so many PPVs consumes lot of space. Considering, 8 bytes per PPV entry, it is $O(|V|^2)$ space. For our graph, it can take upto 2157 GB of space. Truncation of PPVs has been proved to work well as shown in the figures below, but pre-computation time is high.

The figure 1.1 shows the variation of L_1 norm, Precision, RAG and Kendall's τ of the rankings at a particular percent thresholding of the posting lists with respect to the rankings obtained without any truncation of the posting lists of the query keywords.

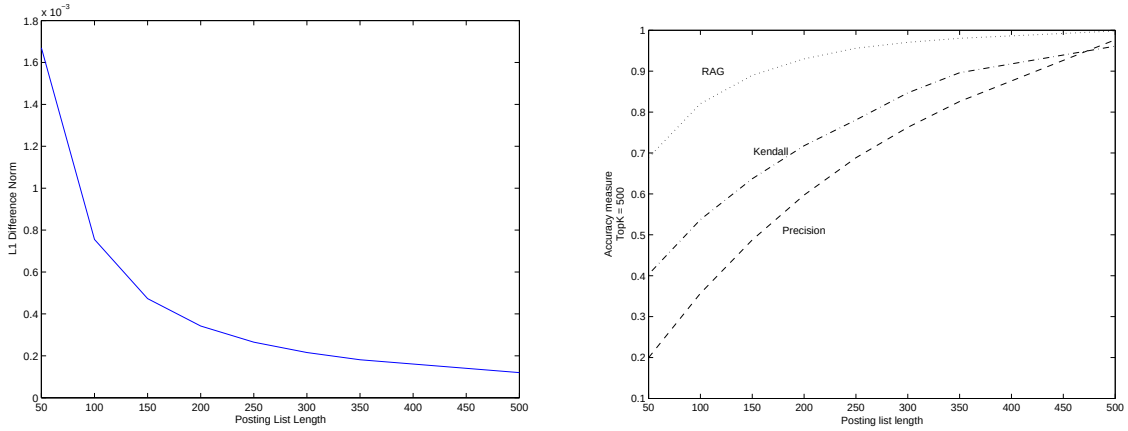


Figure 1.1: L_1 norm, Precision, RAG and Kendall's τ vs Posting list length

1.2.4 Bookmark Coloring Algorithm(BCA) –push

Bookmark Coloring Algorithm proposed by Berkhin in [Ber07] regards the authority flow in PageRank model as the diffusion of a coloring substance across the graph. Initially, a unit amount of paint is injected into a selected node called as a bookmark. A fraction $1-\alpha$ of the paint sticks to the node, while the remainder (α) fraction flows along out-links. This propagation continues down the graph recursively. Intensity of the accumulated color plays the role of the node authority. The final distribution of the paint gives the Bookmark Coloring Vector with respect to the start node. This Bookmark Coloring Vector(BCV) is similar to the PPV.

Algorithm 1 Push Algorithm

Input: G, origin node o, tolerance $\epsilon > 0$

Output: Approximation to p_{δ_o}

1: $s \leftarrow 0$, q$\leftarrow \delta_o$
2: **while** $||q(u)|| > \epsilon$ **do**
3: select node u such that $q(u) > 0$
4: $s(u) \leftarrow s(u) + (1-\alpha)q(u)$
5: **for all** $v : (u,v) \in E$ **do**
6: $q(v) \leftarrow q(v) + \alpha C(v,u)q(u)$
7: **end for**
8: $q(u) \leftarrow 0$
9: **end while**
10: **return** s

1.2.5 Bookmark Coloring Algorithm(BCA) for hubs

Berkhin in [Ber07] also proposes usage of hubnodes when doing the query processing. When pushing, if a particular node is reached, it would certainly help if the PPV of that node is known beforehand. In that case, pushes from that node can be stopped and the already stored PPV can be used. He calls such nodes as hub. If while pushing the activation from a node to

its neighbors, a hubnode h is encountered, then activation spreading can be stopped and the stored- precalculated p_{δ_h} can be used rather than re-doing the activation-spreading job. BCA with hubset works faster than ObjectRank.

The modified algorithm considering the hubset H is as given below:

Algorithm 2 Push Algorithm with a Hubset

Input: G, origin node o, hubset H, tolerance $\epsilon > 0$

Output: Approximation to p_{δ_o}

1: $s \leftarrow 0$, q$\leftarrow \delta_o$
2: **while** $||q(u)|| > \epsilon$ **do**
3: select node u such that $q(u) > 0$
4: **if** $u \in H$ **then**
5: retrieve p_{δ_u} from cache
6: $s \leftarrow s$+q(u)p_{δ_u}
7: **else**
8: $s(u) \leftarrow s(u) + (1-\alpha)q(u)$
9: **for all** $v : (u, v) \in E$ **do**
10: $q(v) \leftarrow q(v) + \alpha C(v, u)q(u)$
11: **end for**
12: **end if**
13: $q(u) \leftarrow 0$
14: **end while**
15: **return** s

Note that though line 5 in algorithm above says that PPVs are retrieved from the database, these loads can be all done together after the push has finished. This ensures that loading of the same PPV is not done multiple times and also batched loads always take less time than the individual ones.

1.2.6 Fingerprinting

Untruncated PPVs are quite large. Though truncated PPVs are small, still the computation time for a PPV is quite high. So, Chakrabarti in [Cha07] proposes to replace the PPVs with their approximation – the fingerprints. A fingerprint, as defined in [FR04], of a vertex u is a random walk starting from u; the length of the walk follows geometric distribution with parameter $1-\alpha$, i.e. after every step the walk ends with probability $1-\alpha$, and takes a further step with probability α. If N_u is the total number of the fingerprints started from node u , the empirical distribution of the end vertex of these random walks starting from u gives an approximation to the PPV for the node u. To increase the precision, N_u should be large enough. The paper discusses probabilistic error bounds in the computed value of PPV(u,v) i.e. the v^{th} coordinate of PPV(u) when N_u fingerprints are used. They propose that to guarantee that the PPV of a node(computed using FPs) does not get hyped by more than $(1+\delta)$ of its actual PPV value, total number of fingerprints should be proportional to $\frac{1}{\delta^2 avgPPV_u}$ which is impractical.

1.2.7 Other notions of proximity

There are different ways of expressing proximity on ER graphs. In [SZ89], Stephenson *et al.*use information centrality as the proximity measure. This uses set of all paths between two nodes weighted by an information-based weighting that is derived from the inverse of its length for computing the information centrality. In [WS03], Scott *et al.*present weighted importance that decays along a path exponentially with path length as the proximity measure. In BANKS [BNH$^+$02], the authors use the inverse of the sum of the edge scores over paths as the notion of proximity.

To consider the importance of a hubnode with respect to the other nodes in the graph, we consider graph conductance as the proximity measure.

In [TKF07], Tong et. al. define proximity from i to j in terms of escape probability from i to j. Escape probability from node i to node j is the probability that a random particle that starts from node i will visit node j before it returns to node i. They present an iterative solution to the problem of computation of single node proximity such that it avoids matrix inversion. They also define the problem of group-group proximity and present fast solutions for both single-node and group-group proximity problems.

They explain that this proximity measure can be computed by solving a linear system. We propose that the system can be solved to evaluate the proximity using the BCA push algo. We then also propose a topK solution for this algorithm.

1.3 Organization of the report

In Chapter 2, we describe the architecture of our system and our test bed and also define some of the relevant terms. In Chapter 3, we discuss the Bookmark coloring algorithm in detail and suggest how to modify it to include hubset nodes and also how to include topK as a quitting strategy for BCA. We also discuss about the order in which we consider the nodes for pushes in BCA, alongwith the data structures for implementing BCA. In Chapter 4, we present different schemes for allocating number of random walks for fingerprint computation. In Chapter 5, we discuss the application of the basic BCA algorithm to compute directional proximity between two nodes based on escape probabilities. In Chapter 6, we present some work on how to update the personalized pagerank vectors, given perturbations to the graph conductance matrix. We also provide a way to use this update algorithm to compute PPVs faster.

Chapter 2

System Architecture

2.1 Typed Word Graph architecture

As is shown in the figure 2.1 the entity word graph structure consists of two parts – the entity graph (that contains one node each for an entity) and a word graph (that contains a word node for each word in the query). There is also a dummy node which is used to allow teleports through the word nodes.

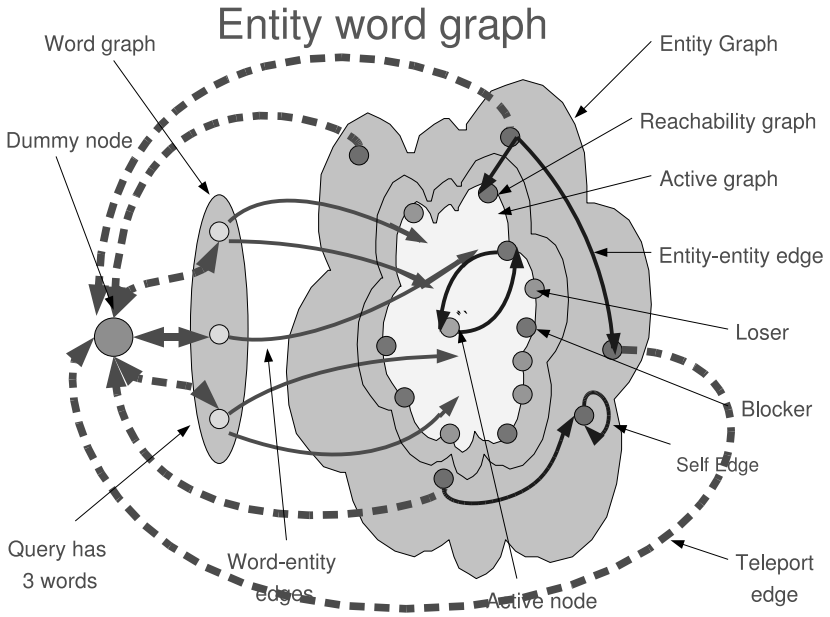


Figure 2.1: The structure of the graph

Edges present in the word-entity graph are of these types:

1. Conductance edges between the entities

2. From words to the entities that contain those words

3. Self edges on dead-end entity nodes, to ensure that the weights of the outLinks of every node sum up to one

4. The teleport edges from every word or entity node to the dummy node and from dummy to every word node.

Weights of edges have been chosen based on intuition.

PPVs are precomputed and stored for a set of nodes in the graph. This set of nodes is called the hubset. The BCA algorithm works on an activation-spreading model where the activation starts from the dummy node and keeps spreading throughout the entire graph. The activation

spreading along a path is stopped when it meets a hubnode(as its stored PPV already encodes the way its activation would be spread). Thus the faster the path reaches a hubnode, the faster will the activation spreading end. That is why we also call the hubnodes as blockers. Note that the PPVs for the hubnodes are pre-computed and stored in their truncated forms or in terms of their approximate forms called FPs. So, the later you find the blocker on the path, the more accurate will be the PPV value.

Activation spreading can also stop if the activation that reaches to a node goes below the activation threshold. We call such a node a loser. We call the part of the entity graph reachable from the dummy and bounded by losers as the reachability graph. The active graph is the part of the entity graph bounded by blockers and losers.

2.2 Data preparation

First we build entity part of the graph and then augment it with word nodes for every query.

Our entity graph has been formed by extracting the author names and paper abstracts and titles from the Citeseer metadata which was available to us in the XML form.

We cleaned the Citeseer metadata, extracted a temporal subset of all the publications before 1994. We call this the pre1994 dataset. Temporal subsetting was done using the "pubyear" tag in the Citeseer metadata XML file. The citations were tracked using the "references" and the "identifier" tags. By having a subset of data which contains information about papers before a particular year, we ensured that all citations (which are inherently backward in time) remain preserved. Then we parsed the XML to get authors, papers and built up the entity graph. Entity graph thus contains two types of nodes and four types of edges.

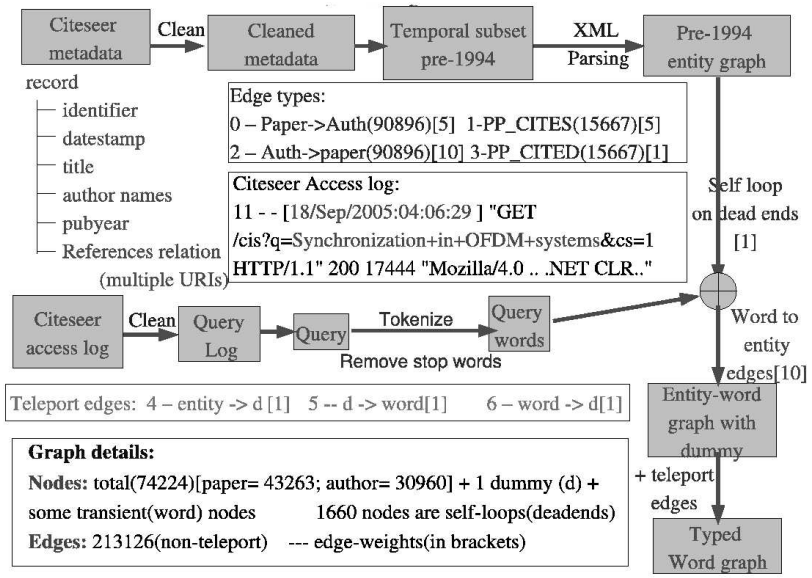
The full Citeseer data has 709173 words over 1127741 entity nodes and 3674827 edges. We perform most of our experiments using two subsets – pre1994 and pre1998. However we also present some scalability studies with the pre1996 and the pre2000 subsets as we hope that scaling would bring out the benefits of BCA better.

Parameter	pre1994 graph	pre1998 graph
TextIndexSize	55 MB	259 MB
#paper nodes	43263	204365
#author nodes	30960	115242
#total nodes	74224	319607
#Paper → Author edges	90896	462810
#Paper → Paper(Cites) edges	15667	89833
#Author → Paper edges	90896	462810
#Paper → Paper(Cited) edges	15667	89833
#Entity → dummy(d) edges	74224	319607
#Self loop edges	1660	5737
#Total edges	287350	1430630

Details of the pre1994 dataset are shown in figure 2.2.

For escape-probability-based-proximity experiments described in section 5.2, we use just the

Figure 2.2: Method for construction of TypedWordGraph



entity graph as the proximity values are independent of the content.

2.3 Query Log

We used the Citeseer query log which has 1.9 million queries collected over 36 days from 21^{st} Aug to 25^{th} Sep, 2005 sorted on time with word frequencies that follow a Zipfian distribution. We used the first 10000 queries as the test queries for most of the experiments. For training purposes we used all but the first 100000 queries to avoid any kind of correlation between the test and the train logs.

2.4 System Overview

The system is based on the codebase of [Cha07]. We parse the pre-1994 XML temporal subset to create the pre1994 entity graph. We index the text to create a Lucene-based NodeTextIndex. We use the train-part of the QueryLog and the NodeTextIndex to create a PersistentQueryLog which contains the query sequence number and the query word set(set of querywords also present in the NodeTextIndex). We use the PersistentQueryLog and the entity graph to choose the nodes that form the hubset.

The hubset selection problem is as follows: Select a set of nodes k out of $\mid V \mid$ such that, when evaluating all the queries using this hubset, the total pushtime is the least compared to that when using any other k-node hubset. Ideally we should consider each of the $(\mid V \mid)_{C_k}$ subsets and select the one which results in the least pushtime aggregated over all the queries.

We use a progressive hubset selection scheme which is based on a benefit-cost model. The benefit represents the benefit of including a node in the hubset(Work saved). This is estimated by the time saved i.e. the push-time when initiating a push from that node with the initial authority at that node being equal to the LidStone-smoothed query-log weighted graph conductance. This can be estimated from the reachability(PushActive) in terms of active nodes that can be

reached when pushing from this node, which can in turn be estimated in terms of shortest-path reachability (PathActive) to the active nodes. Computing PushActive is efficient when we use a randomized-approximation reachability estimation algorithm by Cohen and compute CohenActive actually. This helps us in choosing our hubset. We observed that a hubset with a mix of entities and words is better than a hubset of only words or a hubset of only entities.

The main drawbacks of this scheme are as follows:

1. It is a batchwise greedy scheme. We select a batch of nodes based on their current benefit-cost scores and do not remove them from hubset in any of the later iterations.

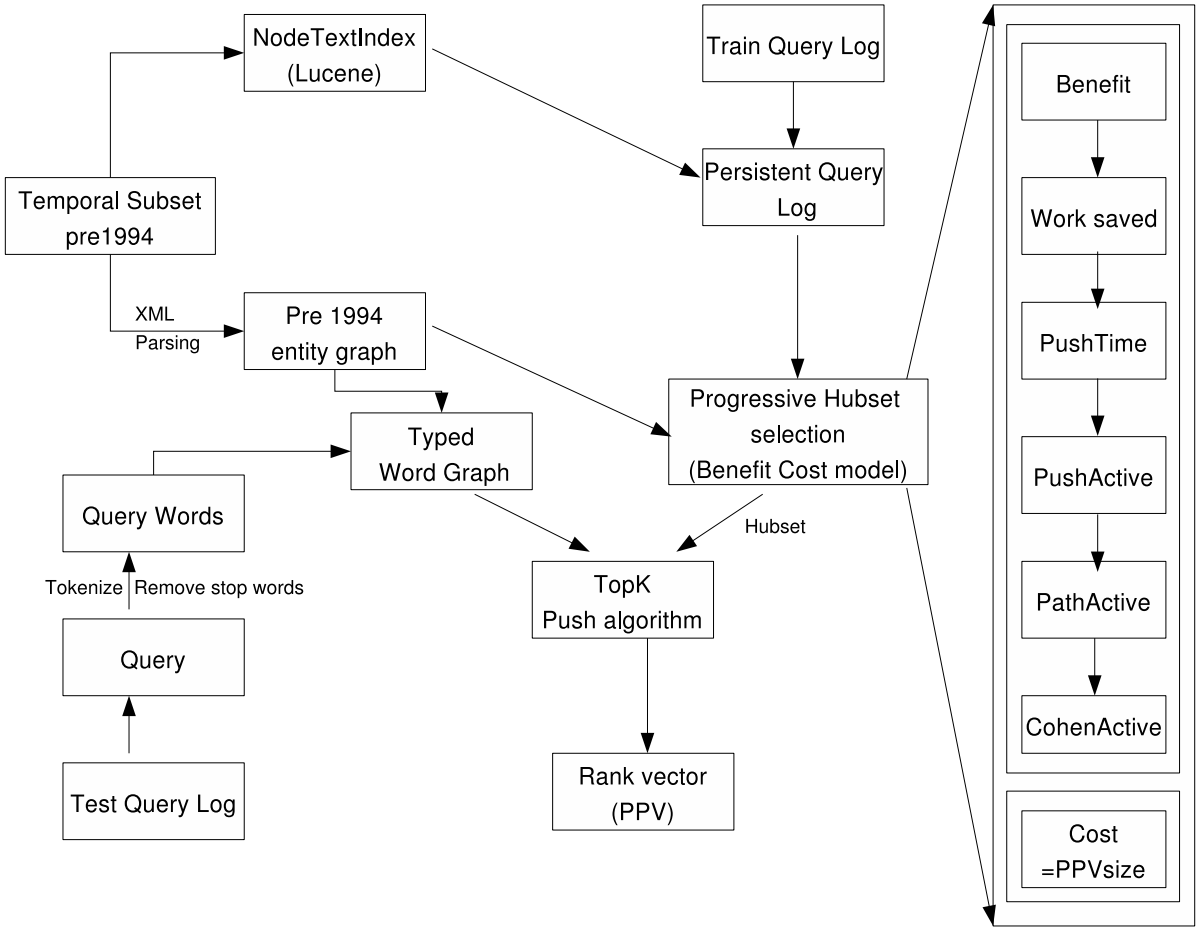
2. It is not constructed considering the Fagin Quit Model(as explained in section 3.2)

When a query is given as the input to our system, we first tokenize the query, remove stop words and words not present in our NodeTextIndex, and then combine them with the entity graph to get the TypedWordGraph. We then use this Typed-Word-Entity-Graph along with the selected hubset as parameters to our topK "push" algorithm and finally get a ranking vector for the query.

Hubset selection is dealt with in detail in [Pat07]. In the next section, we look at the "push" algorithm that is used to generate the ranking vector for the user query.

The figure 2.3 provides an overview of our system.

Figure 2.3: System Overview



Chapter 3

TopK push with hubs

3.1 Correctness and Termination of the Push algorithm

3.1.1 Distribution theorem

Distribution theorem states that PPV with respect to a teleport vector t is the weighted sum of the teleport vector itself and the PPV with respect to the teleport vector αCt.

For a vector t, $p_t = (1-\alpha)t + p_{\alpha Ct}$

Proof:

$$\begin{aligned} RHS &= (1-\alpha)t + p_{\alpha Ct} \\ &= (1-\alpha)t + (1-\alpha)\left(\sum_{k>=0} \alpha^k C^k\right)(\alpha Ct) \\ &= (1-\alpha)\left[\mathbb{I} + \left(\sum_{k>=1} \alpha^k C^k\right)\right] t \\ &= (1-\alpha)\left(\sum_{k>=0} \alpha^k C^k\right) t \\ &= p_t \end{aligned}$$

3.1.2 Correctness of the algorithm

The following shows why the above algorithm finally returns the PPV.

Say, if s, q and s', q' are the values before and after one iteration, then we know that

$q' =$q$-q(u)\delta_u+$q(u)$\alpha C\delta_u$

$s' = s + (1-\alpha)q(u)\delta_u$

Consider

$$\begin{aligned} s + p_q &= s + p_{q'+q(u)\delta_u - q(u)\alpha C\delta_u} \\ &= s + p'_q + p_{q(u)\delta_u} - p_{q(u)\alpha C\delta_u} \\ &= s + p'_q + (1-\alpha)q(u)\delta_u + p_{\alpha Cq(u)\delta_u} - p_{q(u)\alpha C\delta_u} \\ &= (s + (1-\alpha)q(u)\delta_u) + p'_q \\ &= s' + p'_q \end{aligned}$$

Initially $s = 0$ and $q = \delta_o$ and also $p_0 =$0

And finally $q \to 0$ so $s \to p_{\delta_o}$

3.1.3 Termination

q' =q−q(u)δ_u+q(u)$\alpha C\delta_u$

Initially $||q||$ is 1.

At every iteration, we remove $||q(u)\delta_u$−q(u)$\alpha C\delta_u||$ i.e. $||q(u)\delta_u(\mathbb{I} - \alpha C)||$ from 1.

Now $||\delta_u||$ is 1.

$||\alpha C||$ is α itself.

Min value of a selected $q(u)$ at any iteration is $\epsilon/ \mid V \mid$ as specified by the termination condition.

Thus, $||q(u)\delta_u$−q(u)$\alpha C\delta_u|| > (1-\alpha)\epsilon/ \mid V \mid$ in every iteration.

In other words, if n is the number of iterations then, $n\epsilon(1-\alpha) <\mid V \mid$

i.e. $n < \frac{|V|}{\epsilon(1-\alpha)}$

Thus no of iterations is $O(\mid V \mid /(\epsilon(1-\alpha)))$

3.2 Push Algorithm for Hubs with Fagin Quits

Iterative PageRank algorithm considers "the point where max difference in any of the element values of the PageRank vectors across two iterations is less than ϵ_{iter}" as the termination condition.

Note that the value of ϵ is very crucial to determining the average time required for the push algorithm to stop. The greater the ϵ, the less will be the time for "push" termination but less will be the accuracy. At any point of time, when push terminates, the residual vector might be containing all the nodes in the worst case. So, if we want to ensure a ϵ_{iter} as the sum of the residual vector when we terminate (considering all the current elements in the heap as losers), we need to have $\epsilon = \frac{\epsilon_{iter}}{|V|}$ However with ϵ_{iter} as low as 10^{-6} and graph size as large as 74223 or 320000, this ϵ is so low that it takes a long time for push to terminate.

So, we propose a topK based termination condition for the "push" algorithm such that even for low ϵ the push terminates quite faster ensuring that the topK rankings are accurate.

In the push algorithm mentioned in the section 1.2.5, the loading of the PPVs are done right at line 5 then, we also have the intermediate PPV score for every node. Final PPV score of any node can be equal to the current PPV score of the node in the worst case and equal to the current score plus sum of the residual vector in the best case.

Thus we have bounds on the PPV score of every node as we iterate through the push algorithm. We can capitalise on these bounds for early push termination using Fagin Threshold algorithm(TA). The only problem here is that we need to load the PPVs within the push iterations itself, but this gets efficient as Berkeley DB maintains its own in-memory cache. Other than this, we need to maintain an extra heap of the worstscores(i.e. the currentscores of all the nodes so far). Note that we don't need any extra book-keeping for the bestscores as bestscores can simply be obtained by adding sumResidual value to the worstscore values. So if we want that the top K rankings be accurate we should set the topK of the Fagin's TA to K. However, many times topK seems to have failed just because the Kth entry has very high currentscore in the initial steps of the algorithm leading to a very good bestscore which prevents TA to exit. So, we implement a TA algorithm with a sloppy topK. This just means that if user wishes to ask for best 10 responses for a query, we just return him say 20 best responses,

Algorithm 3 Push Algorithm for hubs with TopK

1: initialize an empty min-heap M
2: **for** each node u in the score map s **do**
3: **if** $|M| < \overline{K}$ **then**
4: insert u into M
5: **else**
6: let $v \in M$ have the smallest score
7: **if** $s(u) > s(v)$ **then**
8: replace v with u
9: **end if**
10: **end if**
11: **end for**
12: consider M in score order $s(u_1) \geq \cdots \geq s(u_{\overline{K}})$
13: **for** $b = \underline{K} + 1, \ldots, \overline{K}$ **do**
14: **if** $\hat{p}_r(u_b) > \hat{p}_r(u_{b+1}) + \|q\|_1$ **then**
15: **return** can terminate push loop on line 2 of the algorithm in the above section
16: **end if**
17: **end for**
18: **return** cannot yet terminate push loop

which means our K varies from 10 to 20. We therefore propose that the number of responses be *bracketed* in $[\underline{K}, \overline{K}]$.

The figure Figure 3.2 shows that sloppyTopK indeed succeeds for a huge number of queries. Infact most of the times the Fagin Quit happens quite close to the lower value of K. Here x-axis denotes the rank at which quit succeeded (-1 = did not succeed) while y-axis denotes the count of queries which succeeded at that rank

Checking for Fagin Quits does involve overheads. So, it is important that we do Fagin checks not so frequently. The figure Figure 3.1 shows that Fagin Quit Checks take very less time(as little as around 4%) and still lead to about 4 times increase in the query execution speed.

If myRes is Residual of a node u currently, then max part of the Residual that goes out of node u in this push and can come back to u ever is $(1-\alpha)myRes + \alpha^2(myRes)$ and max part of the Residual that goes out of other nodes (except u) in this push and can come back to u ever is $\alpha(sumRes - myRes)$

So, increase in score of node u can be at max $(1-\alpha)myRes + \alpha^2(myRes) + \alpha(sumRes - myRes)$

$= (1-\alpha)^2 myRes + \alpha(sumRes)$

Now, this is a stricter bound for the bestscore but implementing this requires that we maintain an additional heap for bestscores as now the gap between the bestscores and the worstscores is not flat. So, we approximate this by using a slightly looser bound:

$(1-\alpha)^2 max_u(Residualvector) + \alpha(sumRes)$

Taking a $max_u(Residual\ vector)$ removes the overhead of maintaining two heaps. The gap is still flat but less compared to sumRes. Using this bound for topK reduces the query execution time by about 6 % without any change in accuracy. Number of queries that exit due to Fagin Quits increases slightly. Also, queries that quit at lower values of K are more now as compared

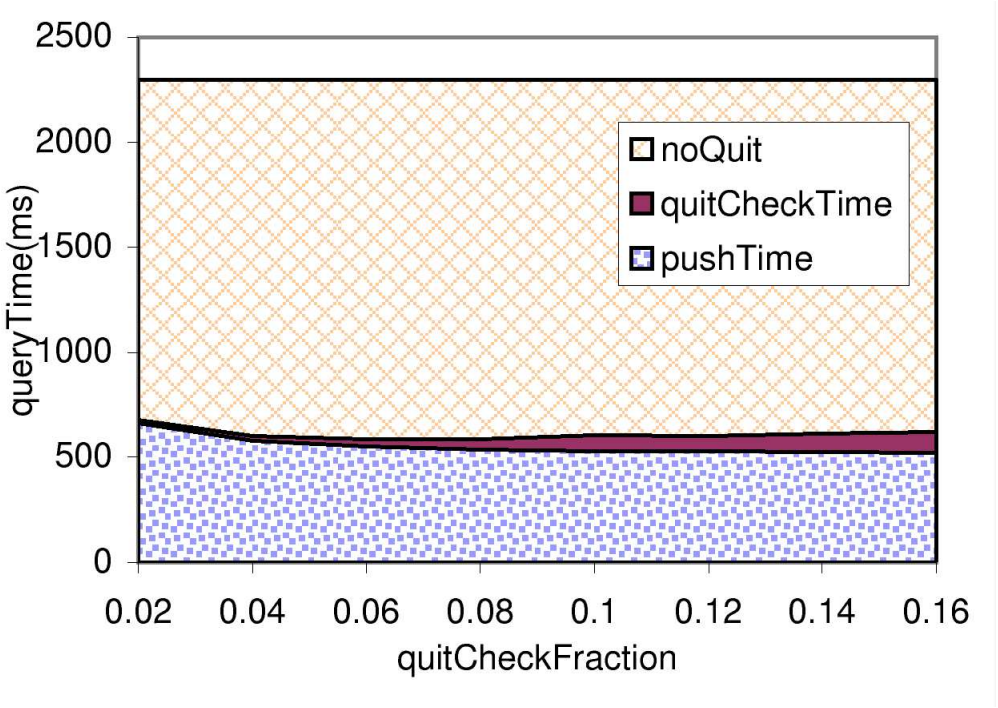


Figure 3.1: Push times averaged across queries vs. fraction of push time allowed in termination checks. (The top line uses no termination checks.)

to that with the old gap=sumResidual. This comparison is shown in the figure 3.2 Implementing the above complex bounds just requires that we maintain a $max_u(Residual\ vector)$ other than the book-keeping required when implementing fagin quit with gap=sumResidual. Supporting this, we observe that fagin quits checks are equivalently expensive and therefore overall query times improve marginally.

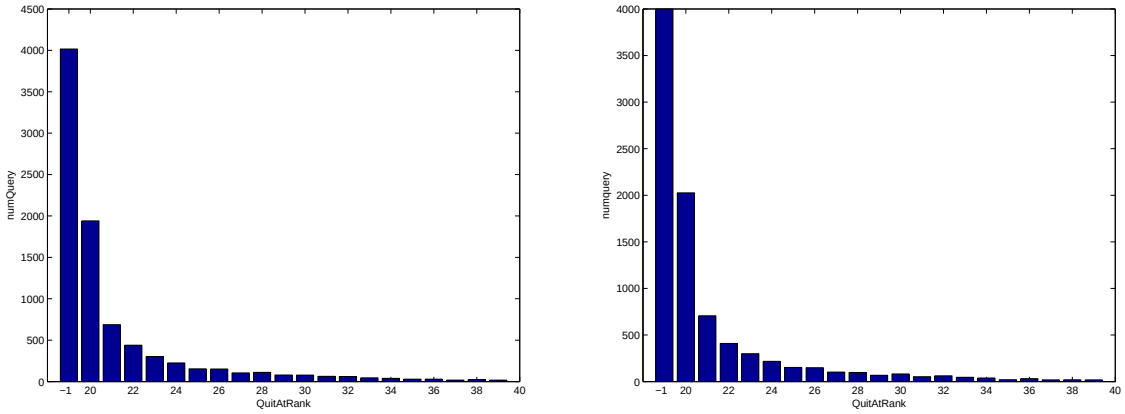


Figure 3.2: (a)Frequency of Fagin Quits with gap = sumResidual (b)Frequency of Fagin Quits with better bounds on gap

3.3 Ordering of the nodes for BCA

When implementing the push algorithm, the main aim was to reduce the sum of the elements of the vector 'q'(residual vector) as fast as possible.

So as to achieve that, one of the hueristics is as follows. At every iteration, we should remove the max $q(u)$ from the q array. For this the residual array q needs to be kept sorted. However note that line no 6 of the Push Algorithm causes change in the values in the q array. So to maintain the q array always sorted we first thought of using a Heap structure.

In the heap implementation, everytime we want a new $q(u)$ to be processed, we take the max $q(u)$ in the q vector. The heap keeps the q array always sorted. This helps the algorithm to converge faster i.e. in less of iterations. But the time required for a lookup in the heap takes far too long.

Another hueristic could be considering a random order of the nodes to be pushed. So we also studied an implementation of the algorithm using HashMap. In this strategy finding node with max residual will take quite long. So, we scan the HashMap from the beginning(this scanning is random due to the inherent property of a HashMap). We take $q(u)$ as the next candidate if its value $> \epsilon$ else we remove that $q(u)$ from the HashMap. This removal step, causes some error in principle but practically we observe that this error is far too less and can therefore be ignored.

So we observe that

1. heap takes less iterations but more time per iteration.

2. A single hashmap takes more iterations but less time per iteration

The Fibonacci Heap, that we used for implementing the heap datastructure so as to consider a residual-sorted order of nodes for the "push" takes O(1) cost for insert and increase-key operations but $O(log \mid V \mid)$ time for delete and deletemax operations. This is too expensive.

A lazy-sort datastructure which maintains the list of all the residuals as an array but sorts them regularly at fixed time intervals ensures that most of the times, the node to be considered for "push" would be among the max residual nodes. However it does not have the expense of maintaining the list always sorted. We call this as the SillyHeap or the LazySort datastructure.

Logarithmic bins which can be implemented as an array of hashmaps of nodes can also be used to store residual values. Each hashmap corresponds to a bin. The number of bins is nbins=ln(number of nodes in graph). The last bin i.e bin no 'nbins-1' is the 'belowTol' bin (it stores all the nodes with score < prTol). Pagerank scores are not linearly distributed so we need to decide the ranges properly. Geometric progression can be used to decide range distribution from 10^{-6} to 1. The penultimate bin should have upper bound=10^{-6}. Thus $r^n = 10^{-6}$. This gives r. Allot the node with score y to the bin no ceil($\frac{log\ y}{log\ r}$)

But ideally we need that the bounds of each of the bins should be dynamically decided so that we can share the load among all the bins equivalently. However, that would again be expensive.

Another datastructure that can store the residual values is the approximate heap called the Chazelle's SoftHeap[Cha00]. By carefully "corrupting" (increasing) the keys of at most a certain fixed percentage of values in the heap, it is able to achieve amortized constant-time bounds for all five of its operations:

create(S): Create a new soft heap

insert(S, x): Insert an element into a soft heap

meld(S, S'): Combine the contents of two soft heaps into one, destroying both

delete(S, x): Delete an element from a soft heap

findmin(S): Get the element with minimum key in the soft heap

The term "corruption" here indicates that a corrupted key's value has been changed by the data structure from the correct value which was initially inserted to a larger, incorrect value. It is unpredictable which keys will be corrupted in this manner; it is only known that at most a fixed percentage will be changed. This is what makes soft heaps "soft"; you can't be sure whether or not any particular value you put into it will be modified.

Key corruption simply means that the order of the nodes chosen for push is not strictly the decreasing sumResidual order. However, this the correctness claims of the push algorithm still hold true.

We observe that the decrease in the residual is almost at the same rate as with the Fibonacci Heap as is shown in the figure 3.3(a)

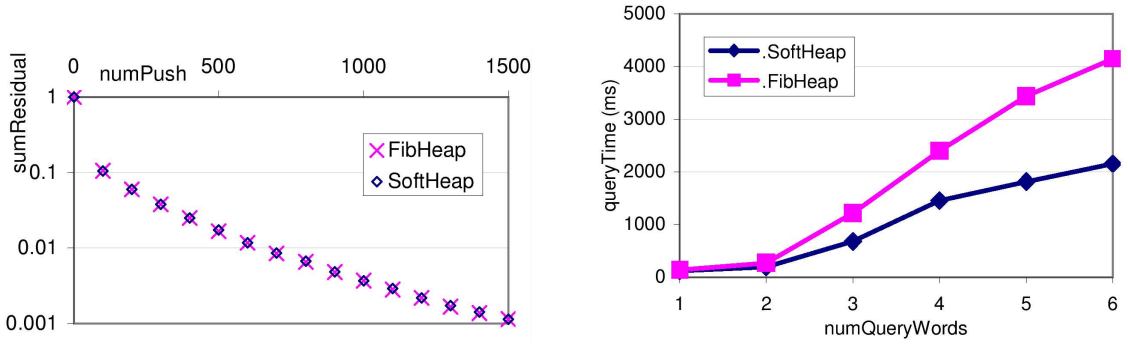


Figure 3.3: (a)Frequency of Fagin Quits with gap = sumResidual (b)Frequency of Fagin Quits with better bounds on gap

We observe that using SoftHeap, the query execution becomes twice as fast as that when using FibonacciHeap. As shown in figure 3.3(b), the benefits become more and more evident as the number of words in the query increase.

The following set of figures shows a comparison of the different data structures. We observe that the SoftHeap provides almost the same accuracy with respect to all the 4 measures – norm, precision, Kendall's Tau and RAG. However as we can see from the figures below that the pushTime for a query execution is quite less when using SoftHeap as compared to that when we use other data structures. These graphs show data over randomly sampled 45 queries from our test query log. All these are without considering any Fagin Quits.

The figure 3.5(a) shows that the number of pushes required when using hashmap as the data structure are far more than any other data structure.

On the other hand, the number of pushes required for SoftHeap is far less as shown in 3.5(b)

We also studied the effect of varying the ϵ on various parameters when using different data structures. We observe that all the accuracy parameters worsen the same way for all the data structures when we increase the ϵ. The figure 3.6 shows a comparison of number of pushes when using different data structures and different ϵ.

The figures 3.7(a) and 3.7(b) show a comparison of the pushtimes.

SoftHeap (push) without Fagin was found to be around 24 times better than ObjectRank even with effectiveMachineEps=$\epsilon/\mid V \mid$. Since the SoftHeap seems to be the best data structure,

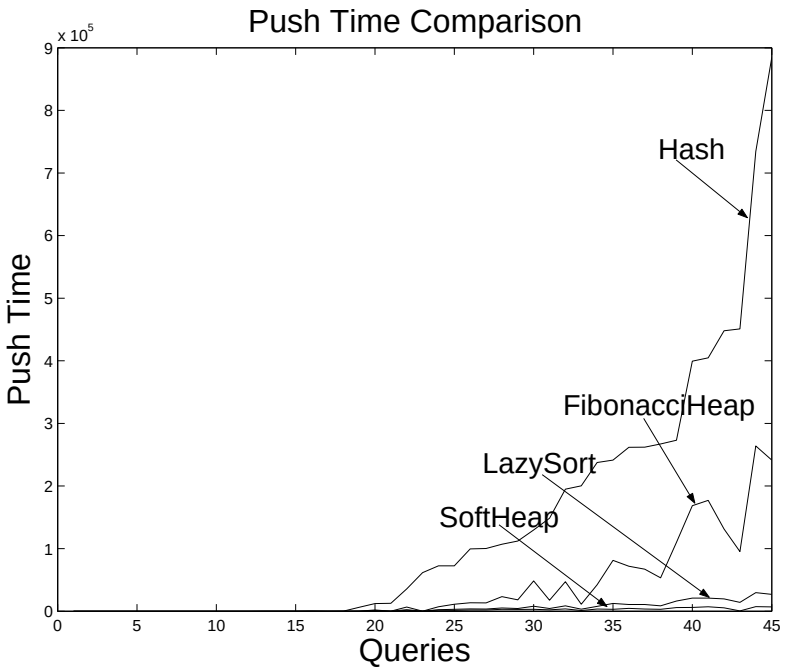


Figure 3.4: SoftHeap is faster than any other datastructure.

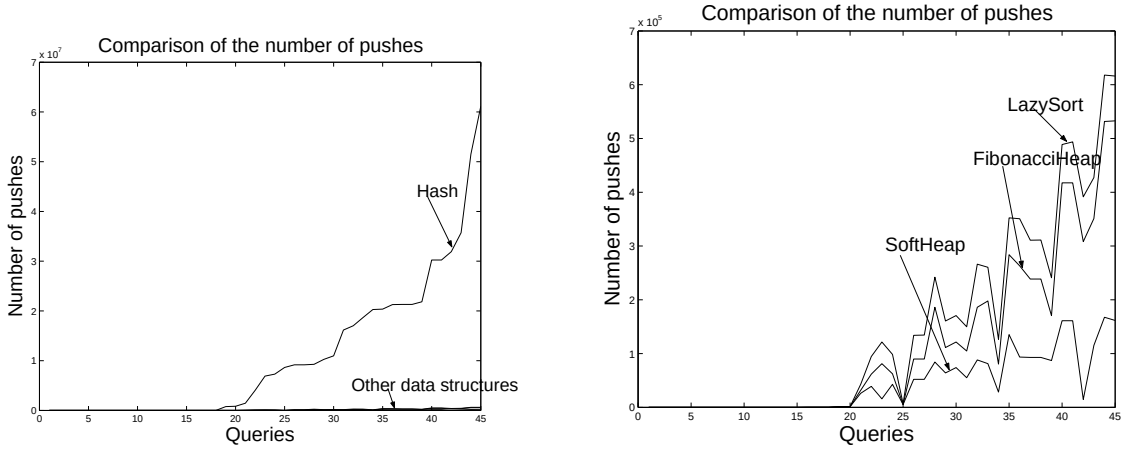


Figure 3.5: (a)Large number of pushes when using HashMap (b)Comparison wrt number of pushes

we also experimented effect of various the K of topK for SoftHeap. We observe that accuracy parameters don't get affected much when we use the same K internally for the evaluation of these parameters. The following figures show the variation in pushTimes as we vary the K in the topK Fagin Quits.

However note that this ordering for the selection of the nodes to be pushed is not the optimal order.

During the push, the same node keeps entering the heap multiple times. This does cause many queries to do more pushes and slow down. Proof for this motivation is the following: I divided the 8615 queries into 2 parts(good and bad) depending on whether the queries take $>$ 1sec. This resulted into 707 BadQueries and 7908 GoodQueries. numPush/pushNumActive

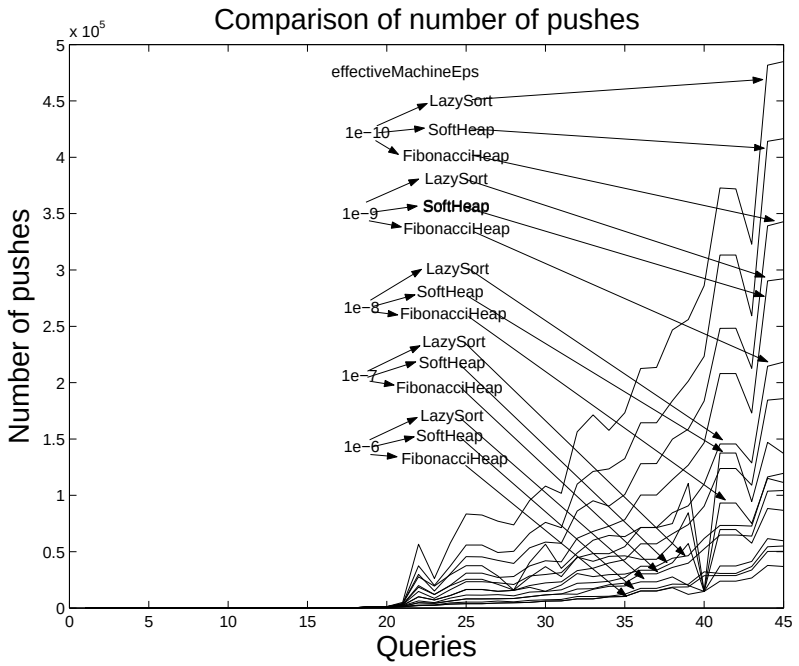


Figure 3.6: Comparison wrt number of pushes for different ϵ

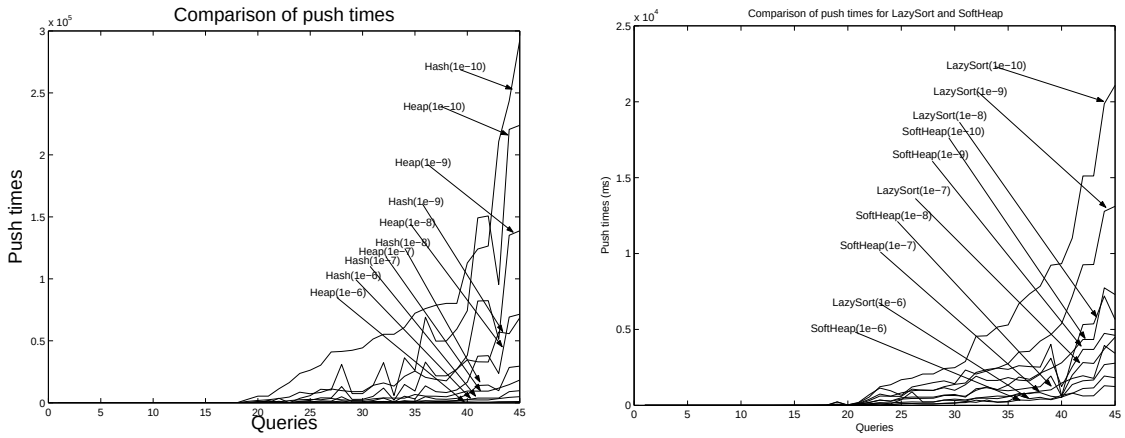


Figure 3.7: (a)Comparison of pushtimes for Hash and FibbonacciHeap (b)Comparison of pushtimes for SoftHeap and the LazySort

is 1.1230 on an average for good queries and 1.4168 for bad queries which implies that on an average a node enters 1.12 times in the heap for good queries and 1.4168 times for bad queries.

So, to improve upon this, we can do either of the 2 things:

1. Somehow save the work done when a node is pushed. Use this saved work, rescaling it by the change in scale of the residual appearing at the node.(Rather than repeatedly doing the pushes from that node. — something like a dynamic blocker.) But then the question is "This work done should be saved for which nodes?"

2. We should order the nodes to be considered for push in some proper way such that the number of times a node re-enters the heap reduces. For this the nodes considered earlier for pushing should have as low Indegree as possible.

Another important factor is the spreading of the residual of the node. Consider a node u with

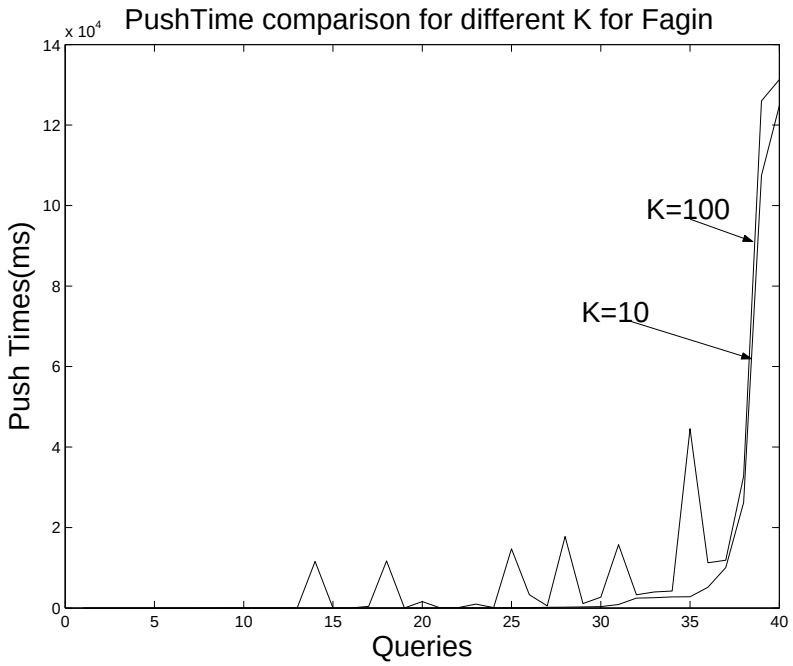


Figure 3.8: Comparison of pushtimes for different K for SoftHeap

residual myRes. If it has high outdegree, its outneighbors(large in number, say m) get "low" residual. As a result this will lead to shorter push paths(say average length a) arising from the authority flow from this node u. If it has low outdegree, its outneighbors(less in number, say n) get "high" residual. As a result this will lead to comparatively longer push paths (say average length b) arising from the authority flow from this node u. But will m nodes each leading to paths of length a take more time or n nodes each leading to paths of length b? Considering that the residual travelling on a path decreases geometrically, having low outdegree seems to cause the push to converge(with some fixed threshold quit or Fagin quit) faster.

Thus, we would a node that is being selected for this push to have these characteristics:(The intuition behind each is sort of based on a greedy approach)

1. high residual

2. low indegree

3. low outdegree. Actually for the entity nodes in our bidirectional graph, outdegree=indegree+1

The next step is to use some function so as to combine these factors.

Using residual*K*indegree as the metric to select the next node for the push does not help. The number of pushes in fact increase for K between 0 and 1. For K=0.1 and using Fagin over a test set of 10000 queries, average numPush=4215.25 as against 3962.7 without considering Indegree.

Using Residual*pow(indegree, K). It provides a better scheduling of nodes for the push algo. On a set of 10000 queries,

1. For NonFagin case: Average numPushes decrease from 21452(K=0) to 21264(K=0.3) and the pushTime decreases from 2877(K=0) to 2109(K=0.3)

2. For Fagin case: Average numPushes decrease from 3962.7(K=0) to 3901(K=0.15)

However the overhead of maintaining an extra HashMap and providing methods to compute the max and the sum of the elements on this map causes the pushTime to increase compared to

the Residual-only(K=0) case.

Other orders too were tried, like Residual*NHS_score and Residual*LAP_score. But neither provide anything better than Residual*pow(indegree,K).

Results for residual*pow(indegree,K):

For non-Fagin case:

K	0	0.1	0.15	0.2	0.3	0.5	1
Avg NumPush	21451.83	21353.39	21316.26	21289.16	21264.81	21357.86	22278.12
PushTIme	2877.37	2145.55	2110.07	2161.73	2108.78	2066.88	2114.22
Ktau	0.94	0.94	0.94	0.94	0.94	0.94	0.94

For Fagin case:

K	-0.2	0	0.1	0.15	0.2	0.5	1	2	5
Avg NumPush	4059.26	3962.7	3979.91	3901.01	3967.15	4004.71	4364.41	5652.89	15518.42
PushTIme	505.55	436.57	478.8	481.21	423.14	475.46	511.7	667.5	1800.29
Ktau	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94	0.94

Results for Residual*NHS_score and Residual*LAP_score:

Method	NonFagin NHS	Fagin NHS	NonFagin LAP	Fagin LAP
Avg NumPush	28578.18	5346.68	2305865.3	2461305.69
PushTime	3103.24	687.01	97525.14	163183.6
Ktau	0.93	0.93	0.93	0.93

3.4 Scalability of the Push algorithm

We studied the scalability of the push algorithm by considering four different temporal subsets of Citeseer metadata. They are as follows:

pre1994 - $|V|=$ 74223 $|E|=$ 287359 IterTime=11.513 sec NodeTextIndexSize= 55M

pre1996 - $|V|=$ 177209 $|E|=$ 752407 IterTime=30.420 sec NodeTextIndexSize= 258M

pre1998 - $|V|=$ 319609 $|E|=$ 1424905 IterTime=61.132 sec NodeTextIndexSize= 139M

pre2000 - $|V|=$ 470029 $|E|=$ 2149461 IterTime=93.440 sec NodeTextIndexSize= 378M

The figure 3.9 shows that choosing a hubset which is about 20% the size of the graph helps to keep the query execution time near 300-400 milliseconds irrespective of the scaling in the size of the graph.

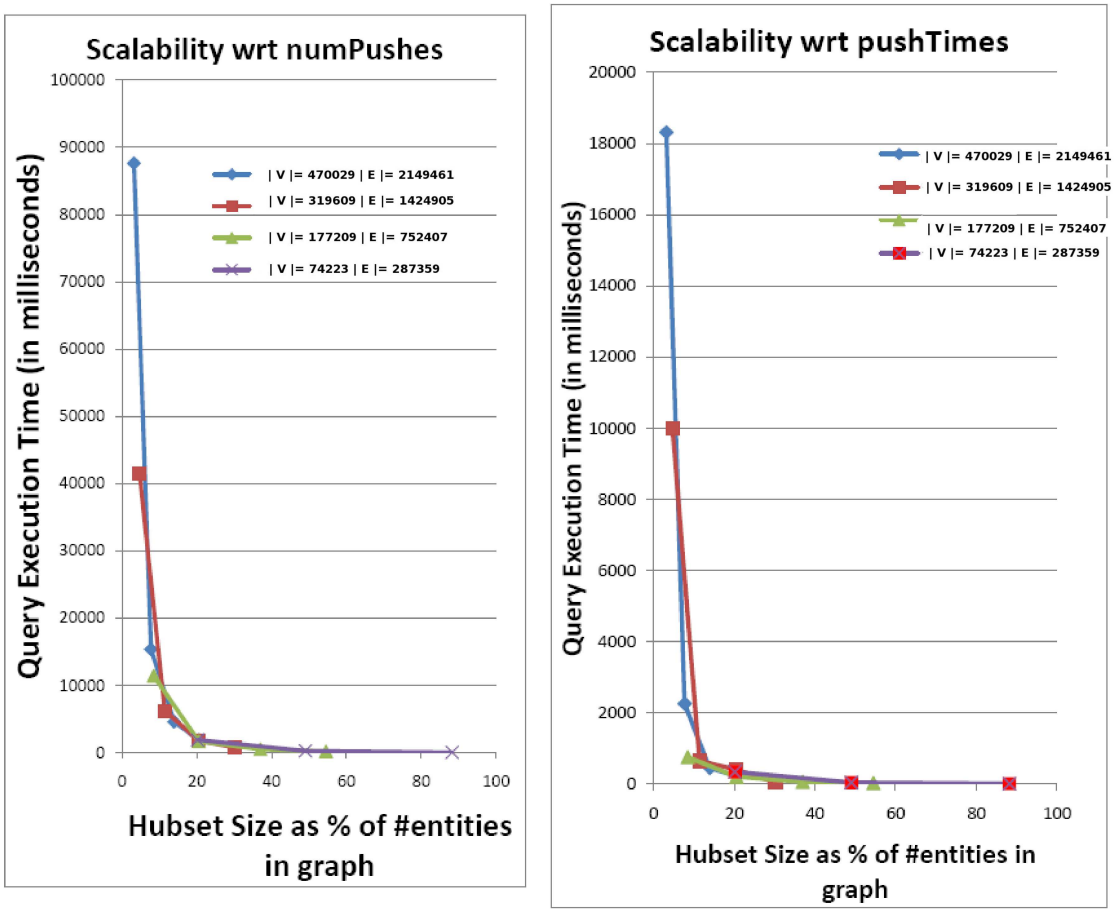


Figure 3.9: Scalability study for Push algorithm

Chapter 4

Fingerprints

4.1 Why use Fingerprints?

Figure 4.1(a) shows that for ObjectRank, as ϵ_{clip} is increased, index size reduces drastically, and results in time being saved during loads inside the push loop. This also reduces query times. It is reassuring to see in Figure 4.1(b) that the savings in space and query time does not happen at the cost of accuracy.

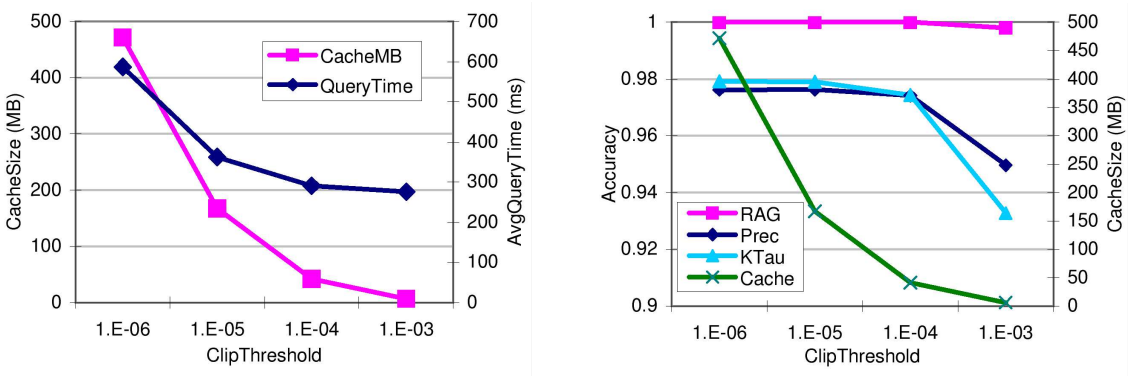


Figure 4.1: (a)Clipping PPVs in H drastically reduce PPV cache size and query processing time. (b)Clipping PPVs has very modest effect on accuracy while reducing PPV cache size drastically.

Clipped PPVs are fairly short and provide substantial accuracy. However PPVs take a huge precomputation time. Recently, Fogaras *et al.* [FR04] and Chakrabarti [Cha07] have proposed that for a node u, PPV_u be replaced by a Monte-Carlo sample FP_u (an vector of small integers) that is faster to compute. Fingerprints (FPs)[FR04] are approximations of PPVs obtained by steady-state distributions of random-restarts. A fingerprint of a vertex u is a random walk starting from u; the length of the walk is of geometric distribution of parameter $1-\alpha$, i.e. after every step the walk ends with probability $1-\alpha$, and takes a further step with probability α.

If N_u is the total number of the fingerprints started from node u , the empirical distribution of the end vertex of these random walks starting from u gives an approximation to the PPV for the node u. To increase the precision, N_u should be large enough. For HubRank, we need to maintain a cache of FingerPrints starting from different nodes. Once we have the hubset selected, we can build an FPCache by starting random walks from each of the hubset nodes. Number of random walks started from each node may be same or different. In their paper, Fogaras *et al.* discuss probabilistic error bounds in the computed value of PPV(u,v) i.e. the v^{th} coordinate of PPV(u) when N_u fingerprints are used. They put up the following claims to choose a value for N_u. In order to guarantee that the PPV of a node(computed using FPs) is not more than $(1+\delta)$

of its actual PPV value, total number of fingerprints should be proportional to $\frac{1}{\delta^2 avgPPV_u}$. They also put forth the following claim. For any vertices u, v,w consider PPV(u) and assume that PPV(u, v) > PPV(u,w). Then the probability of interchanging v and w in the approximate ranking tends to zero exponentially in the number of fingerprints used. Specifically, the total number of fingerprints should be $(N > \frac{1}{(PPV_o(u)-PPV_o(v))^2})$. The values of N_u provided by both of these claims are very huge and impractical. So, we present different numWalk allocation schemes in the next section.

4.2 NumWalk allocation schemes

4.2.1 Fill upto convergence

Algorithm 4 Fill upto convergence

```
List ← TypedWordgraph nodes in Lidstone-smoothed query-log weighted graph conductance(merit) order
numChunk ← ChunkSize
while not all FPs have been computed do
    numwalks ← ChunkSize
    i ← next node in the List
    while TRUE do
        Update FP for i by performing "numChunk" more random walks
        numWalks ← numWalks + numChunk
        if |currFP- prevFP | < 1e-4 then
            break
        end if
        if numWalks > Short.MAX_VALUE then
            break
        end if
    end while
    Save the FP to the disk
end while
```

It was observed that some 93% of word/entities, the FP convergence did not occur before the numWalks reached a count of Short.MAX_VALUE. In this policy, we either compute a decent FP or not at all, unlike linear allocation where many FPs are created with very small numWalks.

FPs are generated by random walks. Greater the number of walks done, theoretically more accurate the FP should be. As N tends to ∞, FP should tend to the actual PPV. However practically we find that FP size goes on increasing as we allocate more number of walks. Also, the L1 norm of the difference of the actual PPV and the FP behaves non-monotonically with respect to increase in the numWalks invested in computing the FP. Figure 4.2(a) shows increase in FP size as we increase the numWalks using a fixed allocation scheme for about 500 nodes. Figure 4.2(b) shows the L1 norm inconsistency.

We have similarly observed that if we keep on bettering a FP by iteratively increasing the numWalks, the $L1$ norm of the difference between the FPs of two consecutive iterations does

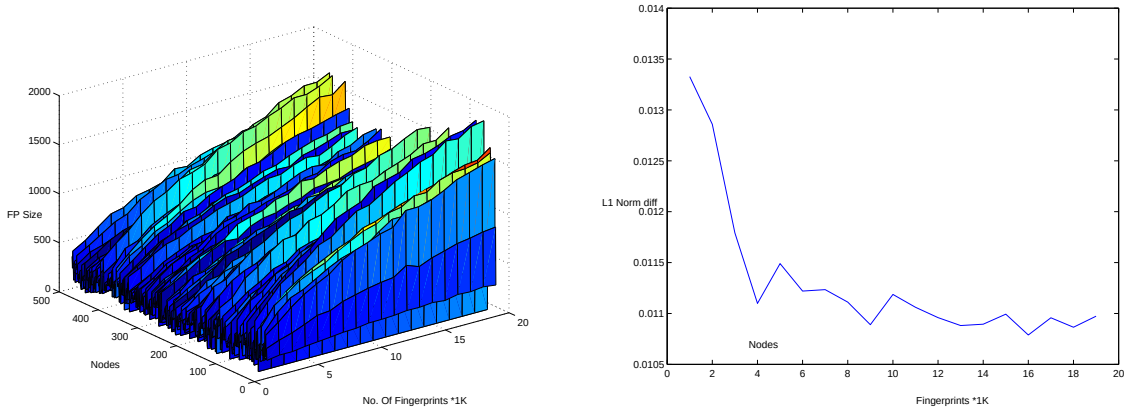


Figure 4.2: (a)FP size increases as we increase numWalks (b)L1 norm between actual PPV and FP

not seem to be converging.

Infact, the fact that for 93% of the nodes, FP convergence was not reached implies that

1. Convergence doesn't really happen for most of the nodes in practice.

2. This scheme becomes equivalent to the uniform allocation scheme where every node is allocated Short.MAX_VALUE number of numWalks.

4.2.2 Linear numWalks allocation

Algorithm 5 Linear numWalks allocation

1: List ← Lidstone-smoothed query-log weighted graph conductance order(NHS)
2: $N \leftarrow$ total numWalks budget
3: **while** not all FPs have been computed **do**
4: i ← next node in the List
5: numwalks = $\frac{NHS\_score_i}{\sum_j NHS\_score_j} N$
6: Compute FP for node i
7: Save the FP to the disk
8: **end while**

Here we distribute this N in proportion to the nodes in the NHS list varying directly with their NHS scores. This supports the fact that the word nodes – i.e. the nodes which can potentially have larger FPs get an allocation of larger number of numWalks. Since more numWalks are allocated to nodes that we expect to have larger FPs, so we expect that the FPs obtained using this allocation would be more accurate.

However because of this kind of allocation, there are nodes at the tail of the NHS ordering that get a very few quota of numWalks. Note that the tail is very large. So effectively, though the nodes at the tail are not very important, we are still wasting some quota computing them. Infact the small quota that each node at the tail gets, results into FPs which are not accurate enough.

In practice linear allocation scheme was found to work better than the saturate-fill or the squared allocation schemes.

4.2.3 Squared numWalks allocation

Fogaras paper says the N_u(total number of numWalks to compute FPs for a node u) should be proportional to $\frac{1}{(\delta^2)(avgPPV_u)}$

We further propose that N should be proportional to $\frac{p_Q^H(h_i)^2}{(\delta^2)(avgPPV_u)}$ where Q is a query word, h_i is a hubnode, H is the hubset and u is any entity node.

The intuition behind this is as follows. Pr(PPV_Q is over-estimated) = P(PPV for a word u with relatively medium PPV values is $(1+\delta)$ times the actual PPV). Now, the relevant part of the approximate score of u is $\sum_i p_Q^H(h_i).\hat{PPV}_{h_i}(u)$ while the relevant part of the exact score is $\sum_i p_Q^H(h_i).PPV_{h_i}(u)$. So the probability of over-estimating PPV_Q is $\Pr[\sigma_i p_Q^H(h_i)\hat{PPV}_{h_i}(u) > (1+\delta)\sum_i p_Q^H(h_i).PPV_{h_i}(u)]$. Considering $p_Q^H(h_i)\hat{PPV}_{h_i}(u)$ as a random variable with mean $p_Q^H(h_i)PPV_{h_i}(u)$ and standard deviation δ, using Poisson distribution, we have $\frac{p_Q^H(h_i)}{\sqrt{N_i}} \approx \delta$. So, N_i should vary as $\frac{(p_Q^H(h_i))^2}{\delta^2}$.

Algorithm 6 Squared numWalks allocation

```
List ← Lidstone-smoothed query-log weighted graph conductance order(NHS)
N ← total numWalks budget
while not all FPs have been computed do
    i ← next node in the List
    numwalks = (NHS_score_i)^2 / Σ_j NHS_score_j  N
    Compute FP for node i
    Save the FP to the disk
end while
```

However with the squared allocation, the skew increases even more. Now we are investing too many walks from favorite nodes, and those do not lead to closer convergence between FP and PPV ,while taking away walks from less favorite nodes. Loading up an FP with numWalks = 1 is not useful as the single random walk can terminate at any node on its path and there are very less chances that PPV value for that node with respect to the origin would be high. Basically, one cannot expect any accuracy out of a single run of a random experiment.

An experiment using the top 10k nodes from the NHS order and a capacity of a total of N=200M numWalks, linear takes 90MB on disk, quadratic takes 78MB because of higher skew.

4.2.4 Cohen-estimate-based numWalk allocation

In this scheme, we consider the nodes in the LAP order and then compute the estimate of their FP sizes using Cohen's reachability estimation algorithm[Coh97]. We then allocate the numWalks among the various nodes in proportion to the size estimates. This is based on the assumption that larger FPs would be more accurate if larger number of numWalks are allocated to them compared to smaller FPs.

In a slightly different approach, after getting the FP sizes, for evaluating FP for a node, we start with a numWalksChunk of 1000 and then increase the number of allocated numWalks in chunks of sizes min(2000,2*fpSizeEst). We go on doing this till the difference in the sizes of the FPs obtained for two consecutive iterations is less than fpSizeConvergence * numWalksChunk.

Algorithm 7 Cohen-estimate-based numWalk allocation

List ← Lidstone-smoothed query-log weighted graph conductance order(NHS)
while not all FPs have been computed **do**
  i ← next node in the List
  numWalkEst=0
  Use Cohen's Reachability Estimation Algorithm to find numReachable upto distance prLoadTol*condToNode. Call it "N" //N should be expected FPSize
  for d = 1,2,... **do**
    //numReach(d): numReachable less than d edges away from nodeId
    if numReach(d) > N **then**
      break
      numReachAtThisDist = numReach(d) - numReach($d-1$)
      $numWalkEst+ = ((\alpha)^d * numReachAtThisDist * log(numReachAtThisDist))$
    end if
  end for
  Compute FP using the numWalkEst
  Save the FP to the disk
end while

where we fixed fpSizeConvergence as 0.1

This results into the following avergae sizes of FPs: Word FPs 4283, Entity FPs 104

However we felt that this size was too huge compared to the sizes of the PPVs truncated at 10^{-3}, which provided better accuracy anyways. So, we thought of truncating the FPs too.

Clipping FPs to a threshold of 10^{-3} results into these averaga sizes: Entity FPs 47, Word FPs 264

This does decrease the size of word FPs a lot. But the accuracy decreases too. Over a sample of first 50 queries of QueryLogTest, Precision decrease from 0.836 to 0.806 on truncation while KTau decreases from 0.712 to 0.658 on truncation.

So, considering clipped FPs was a bad choice. So, we considered the plan of having a hybrid cache. Actually, word FPs were too long while entity FPs were quite short. So we thought of using FPs for entities and PPVs for words.

In general, if the numActive from a node is too huge, then its FP estmation would not be as good as its truncated PPV. So we would like to use FPs for all those nodes which have their numActive below a fixed "threshold" and use PPV for all the nodes with numActive above that threshold.

Observations with this hybrid cache: The figure 4.3(a) shows that even with the hybrid scheme the amount of space required to store the entity FPs and the word PPVs soon crosses the space required to store both words and entities as PPVs. At the break-even point, the accuracy obtained using this hybrid cache is lower than that obtained with an all PPV cache. Still, this scheme works better compared to the linear numWalks allocation scheme.

Figure 4.3(b) shows the accuracy obtained as the total number of numWalks invested are varied, following the Cohen numWalk allocation

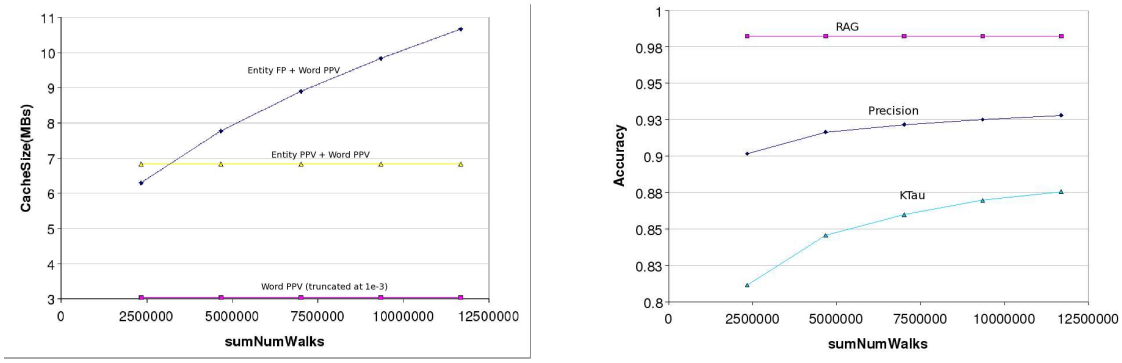


Figure 4.3: (a)Indexing space for Cohen (b)Accuracy for Cohen

4.2.5 Uniform numWalk allocation

Consider each node v reachable from origin o as a bin. Each random walk is like throwing a ball to land in bin v with probability $p_o(v)$. If there are many nodes reachable from o, we have to throw far too many balls until FP_o approximates PPV_o well. After M tosses say m bins have been touched, with counts $c_1, \ldots, c_m > 0$, $\sum c_i = M$. Pretend there is an additional bin "$\perp$" with count $c_\perp = 0$; this stands for an unknown number of untouched bins. Wrt ranking, our primary concern is to make sure that $p_o(\perp) < \epsilon_{\text{trim}}$ with high confidence.

Consider the Bernoulli process of a ball going into $\perp$ or somewhere else. There is a hidden parameter $\theta = \Pr(\perp)$ that we wish to estimate/bound. It is known [Bis06] that $\theta|M$ follows a $\beta(1, M+1)$ distribution, with

$$\mathbb{E}(\theta|M) = \frac{1}{M+2} \quad \text{and}$$
$$\mathbb{V}(\theta|M) = \frac{M+1}{(M+2)^2(M+3)} \approx \frac{1}{M^2} \quad \text{for large } M.$$

From these we can bound $\Pr(\theta > \epsilon_{\text{trim}}|M)$, or judge if $\theta < \epsilon_{\text{trim}}$ at a specified confidence level. E.g., at 99% confidence, $\theta < \mathbb{E}(\theta|M) + 3\sqrt{\mathbb{V}(\theta|M)} \approx 4/M$, so it is adequate to pick $M > 4/\epsilon_{\text{trim}}$.

Using this strategy, we allocated numWalks and computed FPs for entities. We still use PPVs for words. The following figures show the results of this numWalk allocation scheme:

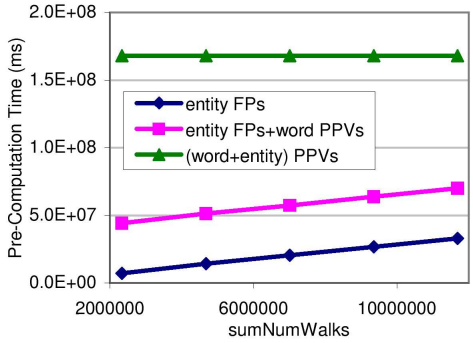


Figure 4.4: Precomputation time saved by using FPs. 'sumNumWalks' is the sum of `numWalks` over all FPs.

Figure 4.5 shows that replacing entity PPVs by FPs results in modest drop of accuracy. If word PPVs were also replaced by FPs, a much larger drop would result, as in [Cha07].

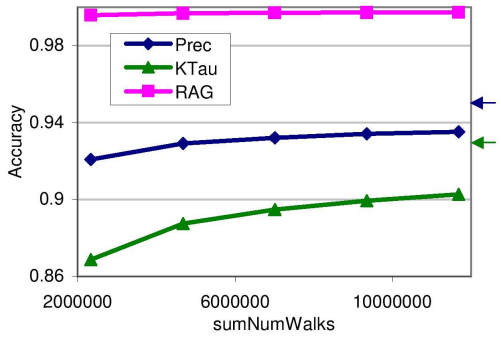


Figure 4.5: Hybrid accuracy is close to PPV-only accuracy (shown with arrows).

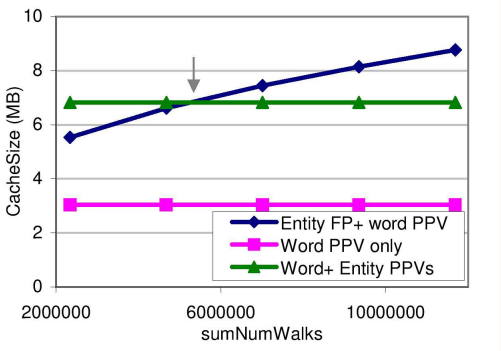


Figure 4.6: Hybrid cache size as more random walks are allocated.

At the breakeven point the accuracy with hybrid cache is almost the same as the accuracy with all PPVs cache.This is better than that obtained when using a hybrid cache developed using the Cohen-based numWalk allocation scheme. But the indexing space and the precomputation time required is far less in the hybrid technique.

Note that we have used doubles to store FPs. But they are generally small numbers and can easily fit even in a two-byte integer. Apart from that, we could do some variable sized encoding and save indexing space.

Chapter 5

Other notions of Proximity

5.1 Different notions of proximity

We looked at the graph-conductance-based notion of proximity. The literature provides some other notions of proximity too. Here we survey some of them and then present a push-based solution to compute topK proximal nodes from a particular node in the graph where proximity is defined using the concept of escape probability as defined in [TKF07] and reviewed below. Figure 5.1 shows some of the possible ways in which nodes 's' and 't' can be connected in a graph. Below we list down different ways of defining proximity.

1. Graph theoretic distance:

It is the the sum of the edge weights of edges which are part of the shortest path connecting two nodes. This is like saying s is more proximal to t than to u because t is 3 friends away while u is 5 friends away. Using this metric, t_1 is as proximal from s_1 as is t_2 from s_2 or t_3 from s_3. However, proximity should consider number of common friends and quality of friends too. Thus t_2 is more proximal to s_2 rather than t_1 to s_1 as s_2 and t_2 share 2 common friends while s_1 and t_1 share just one. t_3 is not as close to s_3 as t_1 is to s_1 because t_3 is related to s_3 only through a very famous friend, thereby denoting a very weak relationship between s_3 and t_3.

2. Maximum Network flow:

If we assume the same capacity for all the edges, then proximity based on max network flow will be high between two nodes which are connected by multiple paths. However max flow disregards the path lengths. Thus, max flow truly identifies that s_2 is more closer to t_2 compared to the proximity between s_1 and t_1. However, it doesn't differentiate among the degree of friendship between two friends who are 3 friends away versus friends who are five friends away. Thus t_4 is as proximal to s_4 as t_1 is to s_1.

Also max flow based proximity is bottleneck dependent. Thus, though there are many pathways added between s_5 and t_5, still s_5 will be as proximal to t_5 as s_4 is to t_4.

3. Using PPVs as in the previous chapters as a measure of proximity is also a good enough choice. However, consider the example (f) in figure 5.1. Here we expect that s_6 is more closer to t_6 than to t_7. t_7 is a celebrity and so the friendship of s_6 to t_6 is more important than friendship of s_6 to t_7. However, if we compute PPV with respect to s_6(assuming all edges of same type and same importance), then it turns out the PPV of t_7 is greater than that of t_6.

4. Tong *et al.* [TKF07] define a new measure – the escape probability from node i to node j as ep_{ij}, probability that the random particle starts from node i will visit node j before it returns to node i. Using this measure, we observe that s_6 is more proximal to t_6 than to t_7. Now consider the elementary graphs shownin Figure 5.2. Intituitively, proximity from A to C should be higher in (a) rather than in (b). This is because in (b), A is connected to C via a

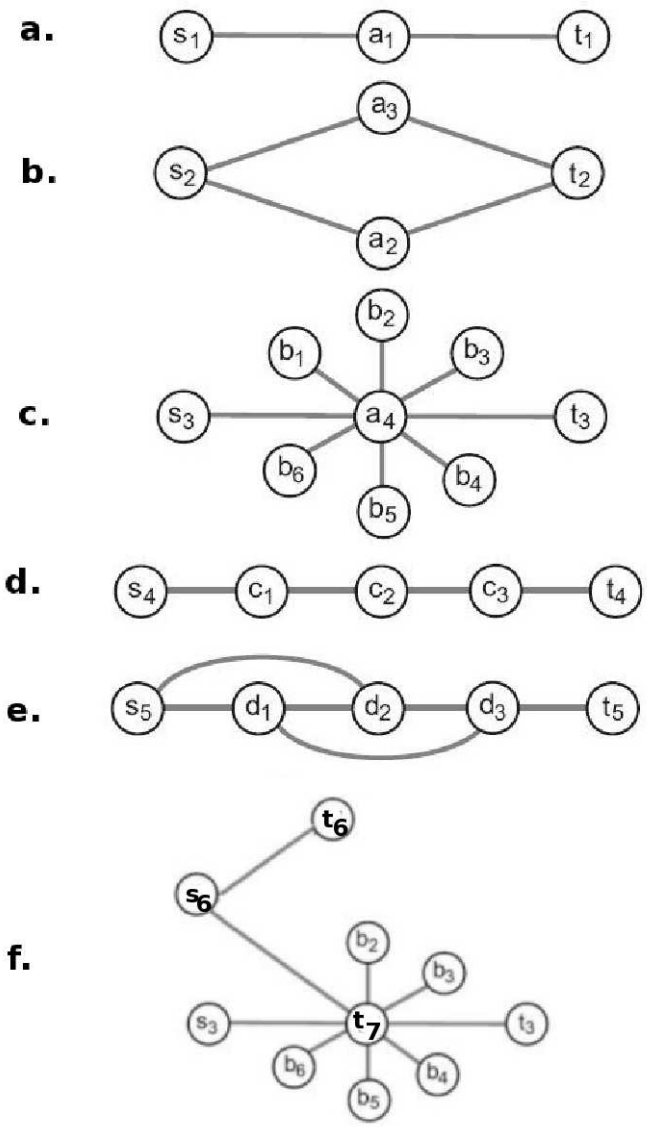


Figure 5.1: Collection of graphs

node B which in turn has noisy elements D and E attached to it. However escape probability from A to C is the same in both the cases. So, a remedy is to have a universal sink node to which each of the graph nodes are connected. This is something like our dummy node in the TypedWordGraph architecture. The sink node ensures that the path from source node to the destination node is taxed according to the number of nodes met in the path. Thus, now due to the introduction of the sink node, each of the nodes D and E pass away some authority to the sink, thereby decreasing the proximity from A to C in (b) while in (a), sink node takes some authority just from node B. Thus now proximity from A to C is more in (a) than in (b).

Thus proximity suggested by Tong *et al.* is like sink augmented effective conductance. One of the problems with this definition of proximity is that it breaks monotonicity i.e. adding paths decreases proximity as more taxing happens.

They explain that this proximity measure can be computed by solving the following linear

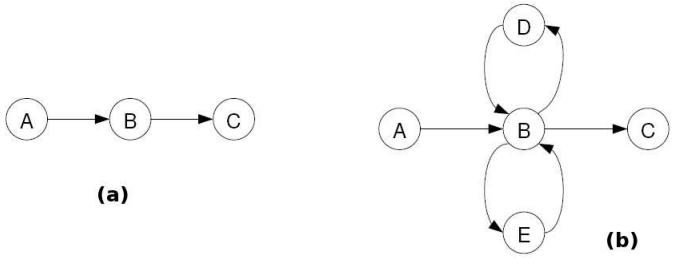


Figure 5.2: Need for sink: Pizzaboy problem

system:

$v_k(i,j) = \sum_{t=1..n} c.p_{k,t}.v_t(i,j) \quad [k \neq i,j]$

$v_i(i,j) = 0$ and $v_j(i,j) = 1$

where $v_k(i,j)$ is the probability that a random walk started from node k reaches node j before node i.

Solution to this linear system leads to

$$Prox(i,j) = c^2 \; P(i,Z)G''P(Z,j) + cp_{ij} \tag{5.1}$$

where $G'' = (I - cP(Z,Z))^{-1}$

$Z = V - \{i,j\}$

and $P = D^{-1}W$ where D is the diagonal matrix of node-outdegrees and W is the weight matrix. Thus P is a row-stochastic matrix.

Tong *et al.* also present an algorithm where they use directed proximity to compute the importance scores for the CePS algorithm which was introduced by Tong and Faloutsos in [TF06]. CePS is an algorithm used to compute center-piece subgraphs. As we show in the next section, we can use psuh to compute directional proximity which implies that push can also be used to compute center-piece subgraphs.

5.2 Directional Proximity using TopK push

We propose that computation of excape-probability-based proximity can be considered as an application of the BCA push algo. By using push, we compute a vector Q as

$$Q = (1-c)P(i,Z)G'' \tag{5.2}$$

Then, using equations 5.1 and 5.2, we can write proximity as $Prox(i,j) = \frac{c^2}{1-c}Q.P(Z,j) + cp_{ij}$

We propose, the application of Fagin's topK threshold algorithm when computing topK proximal nodes from i. At any point of time, we maintain a heap of nodes with topK proximity values. When doing a push, to compute the proximity of a node j from node i, the worstcase proximity is given by $(\widehat{prox(i,j)})_w = \sum_l (currentscore_l) * \frac{c^2}{1-c}P(l,j) + cp_{ij}$. Now in the further iterations of the push algorithm, potentially the entire residual can flow into the node which is an in-neighbor of the destination node and the edge connecting it to the destination node carries the maximum conductance out of all the inLinks of the destination node. So, in the bestcase, proximity is given by $(\widehat{prox(i,j)})_b = \sum_l (currentscore_l) * \frac{c^2}{1-c}P(l,j) + sumRes * \frac{c^2}{1-c} max_l \; P(l,j) +$

cp_{ij} where $currentscore_l$ is the value of Q_l as it is being computed using push. If the best case value goes below the minimum proximity value in the heap, we stop the push.

We ran this proximity algorithm over the pre1994 graph to get the most proximal nodes to the node number 38752(the topmost node in the selected hubset). This node has 673 entries in the PPV truncated to a minimum entry of 10^{-6} Without Fagin Quit the total 673 Proximity computations took 5377.0 pushes, while with Fagin Quits,

Fagin TopK=10 : 4280.0 pushes

Fagin TopK=20 : 4683.0 pushes

Fagin TopK=30 : 4729.0 pushes

Fagin TopK=40 : 4744.0 pushes

Further, we can improve the bounds for the Fagin's TA.

We have assumed that entire of the authority in the residual vector flows into the $max_l\ P(l,j)$ node. However, as explained in the section 3.2, we know that at max $(1-\alpha)myRes+\alpha^2(myRes)+\alpha(sumRes-myRes)$ can flow in into any node. So, we consider this bound on the auhority flow into $max_l\ P(l,j)$ node and assume that the remainder of the authority in the residual flows into the $2ndmax_l\ P(l,j)$ node. This leads us to the following expression as the new bound on Proximity.

$\widehat{prox(i,j)} = \sum_l(currentscore_l) * \frac{c^2}{1-c}P(l,j) + \frac{c^2}{1-c}max_l\ P(l,j)[(1-c)^2 myRes_{max_l P(l,j)} + csumRes] + \frac{c^2}{1-c}\ 2ndmax_l\ P(l,j)[1-(1-c)^2\ myRes_{max_l P(l,j)}\ -\ c\ sumRes] + cp_{ij}$.

However, this might require some more query processing time for the maintenance of various intermediate values. If there is a node l with $P(l,j)! = 0$, it is not necessary that l is reachable from i. If l is not reachable from i, then the push will finally result in $Q(l) = 0$. Therefore, when considering $max\ P(l,j)$ or $2ndmax\ P(l,j)$ we should ensure that $l \in \{reach\ from\ i\}$. Then, quit if $\widehat{prox(i,j)}$ <root of heap, where heap is the minheap of topK proximity values computed so far.

5.3 Another way of computing proximity using TopK push

In their paper [TKF07], Tong *et al.* also present a representation of directional proximity in terms of PPV values, as follows:

$prox(i,j) = \frac{(1-c)r_{ij}}{r_{ii}*r_{jj}-r_{ij}*r_{ji}}$

To find topK proximal nodes to node i, naively, we need to do n pushes and find all PPVs. We first present an algorithm as to how to terminate these n pushes early using Fagin's TopK.

First compute full PPV for node i using full push (terminating only when sumResidual has reached $\epsilon/\mid V \mid$).

$prox(i,j) = \frac{(1-c)}{\frac{r_{ii}}{r_{ij}}r_{jj}-r_{ji}}$

Let $\frac{r_{ii}}{r_{ij}} = A$.

Therefore, $prox(i,j) = \frac{(1-c)}{Ar_{jj}-r_{ji}}$

Now best value of $prox(i,j)$ at any point of computation of PPV for node j using push is given by:

$$\widehat{prox(i,j)} = \frac{(1-c)}{A*\ currentscore\ [r_{jj}]\ -\ best\ [r_{ji}]} \tag{5.3}$$

where best $[r_{ji}]$ = currentscore $[r_{ji}]\ +\ (1-c)^2\ myRes_i\ +\ c\ sumRes$

If $\widehat{prox(i,j)}$ < root of minheap, do Fagin Quit.

We don't expect Fagin Quits in this form to provide much benefits. Fagin Quits used to make a difference in HubRank because word PPVs were involved in it and each word PPV actually takes a large number of pushes. But for entity PPV computations, anyways the number of pushes are far too less, for Fagin quit to make a substantial difference.

So, we need to somehow establish a relation between proximity computations for two destinations(say j and k) We perform the proximity computations from node i in the decreasing PPV_i order. Consider nodes j and k such that $r_{ij} > r_{ik}$. i.e. We start push from j before k.

The question to answer is as follows. Can we use $prox(i,j)$ to cause faster quits when computing $prox(i,k)$? i.e. can we actually use the knowledge from the previous proximity computations from node i to strengthen beliefs for a stronger topK bound when computing proximity values in the future?

Say, the heap has k nodes in it and smallest value is D.

So, we know that $\frac{(1-c)r_{ij}}{r_{ii}*r_{jj}-r_{ij}*r_{ji}} < D$

Quit if $best\frac{(1-c)r_{ij}}{r_{ii}*r_{jj}-r_{ij}*r_{ji}} < D$.

$r_{ii} > \frac{r_{ij}}{r_{jj}}(\frac{1}{D(1-c)} + r_{ji})$

Therefore,

$\frac{(1-c)}{\frac{r_{ii}}{r_{ik}}r_{kk}-r_{ki}} < \frac{(1-c)}{\frac{r_{ij}}{r_{jj}}(\frac{1}{D(1-c)} + r_{ji})\frac{r_{kk}}{r_{ik}} - r_{ki}} = \frac{1}{Ar_{kk} - r_{ki}}$

Thus, Quit if

$$\frac{(1-c)}{A' \; currentscore[r_{kk}] \; - \; best[r_{ki}]} \; < \; D \tag{5.4}$$

Combining equations 5.3 and 5.4, we can use either of them as the quit condition depending on which is greater A or A'.

where $A' = \frac{r_{ij}}{r_{jj}}(\frac{1}{D(1-c)} + r_{ji})\frac{1}{r_{ik}}$

Thus, we have looked into different ways in which we can apply our topK BCA framework to the computation of escape-probability-based proximity computation. Which of them works best can be known only after each of the above techniques are implemented. We leave the implementation as future work.

Chapter 6

Handling Graph Updates

6.1 Introduction

Updations to the graph can be of different kinds:

1. Updation of the weights of one or more edges(edge types in our case)
2. Addition of a new node resulting into a potential addition of new edges
3. Deletion of a node from the graph resulting into a potential deletion of existing edge

Updates to graph structures are a very common phenomenon. Consider a social network. Update of type (1) occurs when a person changes the degree of his friendship with one or more of his friends. Update of type (2) happens when a new person enters a network and becomes a friend and/or extends friendship to other existent members of the network. Updates of type (3) happen when a member leaves the network.

In a graph-representation of the web, updates happen when a crawler crawls some more pages or when some pages are reported to be stale or some pages have been reported as abuse and so need to be removed from crawler's index.

Our aim is to note the change in the relative ranking of the nodes among the topK nodes in the PPVs of various nodes, given a perturbation E in the values of the conductance matrix C.

Let $\hat{C} = C + E$ be the new conductance matrix. Our goal is to find out the new $\hat{p} = (\alpha\ \hat{C} + (1-\alpha)1'.r)\hat{p}$

where $p = (\alpha\ C + (1-\alpha)1'.r)\ p$

6.2 Using push to compute matrix inverse

Updations need that the inverse of the matrix $(I - \alpha C)$ be computed. So lets first have a look at how we can use push to find inverse of a matrix.

PageRank formulation: $p_r = (1-\alpha)(I - \alpha C)^{-1} r$

Say $r_i = \delta_i$

So, we compute $p_{r_i} = (1-\alpha)(I - \alpha C)^{-1}\ r_i$ for all i.

Now we augment these column vectors.

$[p_{r_1}..p_{r_n}] = (1-\alpha)(I - \alpha C)^{-1}[r_1..r_n]$

$[r_1..r_n]$ is the identity matrix

So $(I - \alpha C)^{-1} = (1/(1-\alpha))[p_{r_1}..p_{r_n}]$

This would take n calls to push.

Since push is far better than $O(n^2)$, this would be efficient than $O(n^3)$

6.3 Graph Updation algorithm

In [CDK+03], Chein *et al.* discuss a method to perform updates to the PageRank vector caused due to perturbations in the graph conductance matrix. They find out a part of the graph which would be affected due to change in weight of edge between two nodes and then convert it into a single node in the transformed graph. They then perform PageRank computations on this transformed graph and then compute the full pagerank vector. However this takes long computation time.

We propose the following. Say we have all the PPVs with respect to all the entity nodes on a graph G with n nodes. Now, we want to compute all the entity PPVs for the graph G' obtained by the addition of a new node. If a new node is added into the graph, we handle this update in two steps: step 1 converts the matrix C to a matrix

$C_1 = \begin{bmatrix} C & 0 \\ 0 & 0 \end{bmatrix}$

and then step 2 converts C_1 to C_2 where C_2 reflects the changes values according to the inLinks and the outLinks of the newly added node.

If we have all the entity PPVs on the graph G, we have $(I - \alpha C)^{-1} = A^{-1}$ as $P = (1 - \alpha)(I - \alpha C)^{-1}$ where P is the matrix obtained by augmenting all the PPVs. Now using the Block Matrix Inversion Lemma,

$(I - \alpha C_1)^{-1} = \begin{bmatrix} A^{-1} & 0 \\ 0 & 1 \end{bmatrix}$

Now the $(n+1)$th node can have inlinks and outlinks to other nodes in the graph. So, the last row of $(I - \alpha C_1)$ has to undergo large changes. Also note that when a new outLink appears from a node, some of the outLinks from this node have their weights changed.

So, there would be some rows [= no of inLinks to node n+1] which would need change alongwith the last column.

Sherman Morrison formula says:

$(A + uv')^{-1} = A^{-1} - \frac{A^{-1}uv'A^{-1}}{1+v'A^{-1}u}$

where both u and v are column vectors.

Each such uv' update takes $O(n^2)$ time

So, the total time that it would take to create a set of all entity PPVs for the new graph G' from all the PPVs of the graph G would be = O(n^2 * (no of inLinks to the newly added node +1))

For a update of the k^{th} row, v' should be new_row-old_row and k^{th} element of u should be 1, other $u(.)$ being 0.

For the update of the last $(n+1)^{th}$ column, v' should be all 0s except the last element that should be 1. u should be set to the weights on different outLinks from node $n+1$.

This will give $(I - \alpha\hat{C})^{-1}$

Multiply by an appropriate r to get the corresponding PPV for the new graph

In general, Shermann Morrison formula can be written as $(X - USV)^{-1} = X^{-1} + X^{-1}U\hat{\Lambda}VX^{-1}$ where $\hat{\Lambda} = (S^{-1} - VX^{-1}U)^{-1}$

If it takes substantially less time to compute the low rank approximation i.e. the matrices U, S and V out of the perturbation matrix E, then this formulation may be better than the multiple applications of the uv' updates.

For handling updates involving deletion of a node from the graph, first use the SM formula(normalizing the weights on every row to sum up to 1 except the $(n+1)^{th}$ column). Then, use the block matrix inversion lemma

Using this graph updation mechanism we can find changes in topK proximal nodes in the graph. Once we have the list of the new PPVs, we can compute the new proximity values by

$prox(i,j) = \frac{(1-c)r_{ij}}{r_{ii}\ r_{jj}\ -r_{ij}\ r_{ji}}$

Using the bounds on the changes in the PPV values we would be able to find a bound on the changes in proximity values or the proximity rankings, given the conductance matrix perturbations.

6.4 Updates to the hubset PPVs

Suppose we do not have all the PPVs on the original graph. Given K hubset PPVs of the graph G, how do we get PPV values for the K hubnodes in the graph G'? i.e. We have the partially filled A^{-1} matrix. Say we have the K columns of A^{-1}. Say we re-arrange them such that they form the first K columns of the matrix A^{-1}

Find a PPV for $r = [0\ 0\ 0\ 0\ 0\ \frac{1}{n-K}\ \frac{1}{n-K}\ \frac{1}{n-K}\ \frac{1}{n-K}\ \frac{1}{n-K}]'$ i.e. K zeroes followed by $n-K$ ones, normalized. This returns a PPV Q. So, now the PPV matrix for the old graph G is $[PPV_1\ PPV_2\ ...\ PPV_k\ Q\ Q\ Q\ Q\ Q]$ i.e. a matrix with last $(n-K)$ column vectors equal to Q.

Now we can use Sherman Morrison formula to get the PPVs after updates to C. Though specifying an error bound to this formulation is difficult, but it has been found to work well in computing the new PPV values for the topK PPVs in practice.

We took 200 samples of random matrices each of order 100 and using topK=30, we tried introducing random perturbations. We observe that the maximum difference between the actual PPV and the computed PPV among the topK over the transformed graph is of the order of 10^{-3}. Thus, approximating the PPVs for the non-hubnodes by a PPV computed considering a teleport to the non-hubnodes, is a good idea.

6.5 Batched(Lazy) updates

We don't want to update the Personalized PageRank vectors, always when there is a perturbation in the graph conductance matrix. We should first be able to check if this update causes too much variation in the final PPV values. If the change is within acceptable precision, we can postpone applying the update thus avoiding running the updates for it. We are interested in the bound on the change so that we can perform the tasks more efficiently.

Change in the PPV can be estimated using the following from [CDK$^+$03]

$|\pi - \hat{\pi}| <= \kappa |E|$

where E is the perturbation in P such that P represents the Markov chain $\pi = P\pi$ and κ is the condition number of the perturbation matrix E and the norms can be any standard norm such as L_1, L_2 or L_∞.

If we have a stream of updates to be applied, then we can group those particular updates in a batch which require changes to the same column or the same row. This helps in grouping multiple Sherman- Morrison updates to the same row or the column into a single update.

Such batching of updates can save us time if the application does not need the steady state vectors to be always accurate.

6.6 Using the matrix updation procedure to compute the PPVs for the graph G

Say the graph G contains n nodes. You want to compute all the entity PPVs for G.

1. Coarsen the original graph to obtain a new reduced graph G'.
2. Compute all entity PPVs for this reduced graph G' using either push or direct PageRank.
3. Get $(I - \alpha\ \tilde{\mathrm{C}})^{-1}$ where $\tilde{\mathrm{C}}$ is the conductance matrix for this reduced graph.
4. Now use the recipe mentioned above to add nodes one by one or in batches to this graph till you get the $(I - \alpha C)^{-1}$ where C is the conductance matrix for the graph G.

This would help:

1. If graph G is too huge to fit into the memory and so PPVs can't be computed using PageRank.
2. This might also reduce the total time required to compute all the PPVs.

In [KK95], Karypis and Kumar present different ways of coarsening the graphs. The first approach is to find a maximum matching in the graph and then collapse the vertices into a multinode. They use a maximal matching so that a large number of nodes can be collapsed into multinodes at the same time. While collapsing we can take care that they don't collapse multinodes that contain hubnodes. Thus finally we would be left with a graph which contains approximately number of nodes equal to the hubset size.

Another way of coarsening the graph is by creating multinodes that are made of groups of vertices that are highly connected. [KK95] discusses both these techniques and ways to obtain maximum matching in detail.

6.7 Graph updates using the HubRank framework

Let H' be the set of nodes whose out-degree and/ or outweights have changed as a result of an update to the graph. Given such changes and the PPVs of the entity nodes on older graph, we want to compute the PPVs on the changed graph.

From the push framework, we express PPV with respect to origin o as $P_{\delta_o} = N_{H,\delta_o} + \sum_{h \in H} B_{H,\delta_o}(h).PPV_h$

We compute the PPVs for all the nodes in H' on the new graph as well as on the old graph using iterative PageRank or by using push.

Now, consider the expression of PPV with respect to origin o and hubset H':

$(P_{\delta_o})_{old} = N_{H',\delta_o} + \sum_{h \in H'} B_{H,\delta_o}(h).(PPV_h)_{old}$

The above system of equations consists of R equations where R is the size of $(P_{\delta_o})_{old}$. We have observed that PPV sizes are quite small (upto around 100) when truncated. So, we can efficiently solve the above system of R equations to get $N_{H',o}$ and $B_{h,o}(h)$ where $h \in H'$

Now, $(P_{\delta_o})_{new} = N_{H',\delta_o} + \sum_{h \in H'} B_{H,\delta_o}(h).(PPV_h)_{new}$

Since the hubset H' remains the same, so N_{H',δ_o} and $B_{H,\delta_o}(h)$ will remain the same. So, we can use the already computed values to compute $(P_{\delta_o})_{new}$. Overall, we need to compute PPVs for nodes whose outweights change - not for old and the new graph, once per graph update.

Then to compute a PPV with respect to origin node o, we need to solve a system of equations with the number of equations equal to the PPVsize. Given that the number of variables for which we need to solve the system is equal to the number of equations, we can obtain the PPVs for the new graph.

6.8 Graph updates and communities

In [TFP06], the authors propose that proximity between any two nodes in a graph can be expressed in terms of relevance scores obatined by running random walk with restarts on the weighted graph. They propose to do this fast by exploiting two important properties shared by many real graphs. They exploit the linear correlations among the rows and the columns of the adjacency matrix by low-rank matrix approximations and the community structure by graph partitioning.

They propose that the normalized graph weight matrix W can be decomposed into two parts – the blocked matrix normalized W_1 which represents the community structure and the matrix normalized W_2 which represents the inter-community links. W_1 can be represented as follows:

$$\begin{pmatrix} W_{1,1} & 0 & \dots & 0 \\ 0 & W_{1,2} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & W_{1,k} \end{pmatrix}$$

Note that each of the components are also normalized. Note that the graph is divided into k different community structures.

By block matrix inversion lemma, $(Q_1)^{-1} = (I - cW_1)^{-1}$ can be written as

$$\begin{pmatrix} (Q_{1,1})^{-1} & 0 & \dots & 0 \\ 0 & (Q_{1,2})^{-1} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & (Q_{1,k})^{-1} \end{pmatrix}$$

Then they propose the following algorithm to compute the ranking vector r_i for a starting vector e_i

In the above algorithm, we can compute different Q_i^{-1} using our push algorithm, thus making the algorithm even faster. Authors of the METIS paper propose different partitioning schemes.

Now suppose that a new node gets inserted into the graph. So, first we need to decide as to which community we can place it in. We look at the communities surrounding that node and put it in the one such that the inter-community links are as less as possible. So, now say we have decided to put it in the m^{th} community. We would like to update Q_m^{-1} which can easily be done by our updation algorithm in the section 6.3. This updates the W_1 part of the W matrix. Next to update the W_2 part, we need to update the SVD USV of the W_2 matrix. To do this, we can use

$$X + AB' = [U\ A] \begin{bmatrix} S & 0 \\ 0 & I \end{bmatrix} [V\ B]'$$

where AB' is just a rank 1 update. This is handled in [Bra06]. The paper also deals with perturbations of rank greater than 1. In [MCW95], authors present some more techniques for efficient SVD updates.

Algorithm 8 PPV Computation Algorithm

Input: Normalized weighted matrix W and the starting vector e_i

Output: The ranking vector r_i

1: Pre-computational Stage(Off-line):
2: Partition the graph into k partitions by METIS[KK95]
3: Decompose W into two matrices:$W = W_1 + W_2$ according to the partition result, where W_1 contains all within-partition links and W_2 contains all cross-partition links.
4: Let $W_{1,i}$ be the i^{th} partition, denote W_1 as shown above
5: Compute and store $(Q_{1,i})^{-1} = (I - cW_{1,i})^{-1}$ for each partition i
6: Do low rank approximation for $W_2 - USV$
7: Define Q_1^{-1} as shown above. Compute and store $\Lambda = (S^{-1} - cVQ_1^{-1}U)^{-1}$
8: Query Stage (On-Line):
9: Output $r_i = (1-c)(Q_1^{-1}e_i + cQ_1^{-1}U\Lambda VQ_1^{-1}e_i$

Using these two update steps we can update both W_1 and W_2 and then get the updated ranking vectors for any starting vector e_i.

Now, if we keep on doing such updates for multiple times, then the partitioning may get disturbed a lot. So, we need to run the partitioning algorithm after regular intervals. When to run the partitioning algorithm, can be decided by observing the norm of the W_2 matrix. Greater the norm of the "inter-community" W_2 matrix, worse the quality of the current partitioning. So, we can set a threshold on the norm of W_2 matrix and the partitioning algorithm is fired whenever that threshold is reached.

Thus, we looked at different ways in which updates to the Personalized PageRank vectors can be handled using the topK BCA framework. We also proposed how the updation algorithm might be used to compute PPVs faster. Especially in the comuunity based graph settings, where we expect the size of each community to be far smaller than the size of the whole graph, matrix inverses involved won't be too huge to be stored. Implementing the proposed work in this chapter and making the algorithms space-wise and time-wise efficient is left to future work.

Chapter 7

Summary and Conclusions

In this report, we first reviewed related concepts like ObjectRank, Personalized Pagerank, Fingerprinting, Directional Proximity. We then used an entity-word graph structure which incorporated teleport as inherent edges and where the teleport happens via the word nodes thus providing a representation of the query in the graph. Then we reviewed the linearity theorem[JW03], the distribution theorem, and the concept of hubset selection[Pat07].

We then reviewed the push algorithm with hubset and presented some implementation details with regards to different data structures viz FibonacciHeap, HashMap, SoftHeap, Logarithmic Bins, LazySort. We presented results in the form of comparisons between execution timings and accuracy of ObjectRank and our HubRank system using the push algorithm. We also presented results on comparisons of the variants of push algorithm. We also studied topK with its sloppiness and reported the benefits of the same.

PPVs required huge pre-computation time. So, we approximated them using FPs. To compute FPs, we need to have a sound numWalk allocation scheme. So, we presented five different schemes viz. Saturate-Fill, Linear, Squared, Cohen-based, Uniform, of which the uniform allocation was found to work well in practice. Then we looked at a different way of expressing proximity. After describing the need for such a proximity measure, we show that the proximity computation algorithm can be implemented as a topK push algorithm. We then presented topK bounds in terms of the previous accurate computations, in a hope to be able to obtain tighter bounds.

Then we looked at the problem of graph updations and studied it from a variety of angles. We presented an updation algorithm using Sherman Morrison formula and block matrix inversion lemma. We also implemented the topK updation algorithm and found that it works practically though providing theoretical error bounds on it seems difficult. We also mention how to do updates in case there is a good blocked structure to our graph. Using updation algorithm, we saw how we can compute PPVs for all the entities in the graph very quickly.

Thus, we have tried to establish a sound and efficient indexing and ranking framework for proximity search on entity-relationship graphs which can help us answer queries of the form "Entities near keywords biochemistry, Information, Retrieval". Our search system is both time-wise and space-wise efficient and almost as accurate as the ObjectRank system, which we use as a baseline. Through the updation algorithm, we have also tried to attack the problem of maintaining these indexes in the light of updates to the graph. To accomplish this task, we used a variety of available techniques like the SoftHeap data structure, the BCA algorithm, the fingerprints and then came up with some useful techniques like topK BCA algorithm, hybrid caches of PPVs and Fingerprints. The dynamics of the push algorithm are difficult to understand and depend on the order in which we select the nodes to be pushed. We then expressed escape-

probability-based proximity as an application of the topK BCA algorithm. The scalability study that we presented shows that topK BCA is indeed a very important technique particularly for huge graphs.

Appendix A

Accuracy measures

We measure two score-related and two rank-related indicators of quality [FR04], comparing the test algorithm with "full-precision" ObjectRank.

L_1 error If p_{δ_u} is the true full-precision PPV for node u, and we estimate $\hat{p}_{\delta_u}$, $\|\hat{p}_{\delta_u} - p_{\delta_u}\|_1$ is a reasonable first number to check. However, it is not scale-free. i.e., for a larger graph, we must demand a smaller difference. Moreover, it is not a faithful indicator of *ranking* fidelity [LM05].

Precision at k p_{x_u} induces a "true" ranking on all nodes v, while $\hat{p}_{x_u}$ induces a distorted ranking. Let the respective top-k sets be T_k^u and $\hat{T}_k^u$. Then the precision at k is defined as $|T_k^u \cap \hat{T}_k^u|/k \in [0,1]$. Clipping at k is reasonable, because, in applications, users are generally not adversely affected by erroneous ranking lower in the ranked list.

Relative average goodness (RAG) at k Precision can be excessively severe. In many real-life social networks, near-ties in Pagerank values are common. If the *true scores* of $\hat{T}_k^u$ are large, our approximation is doing ok. One proposal is (note that $\hat{p}_{x_u}$ is not used):

$$RAG(k,u) = \frac{\sum_{v \in \hat{T}_k^u} p_{x_u}(v)}{\sum_{v \in T_k^u} p_{x_u}(v)} \in [0,1]$$

Kendall's τ Node scores in a PPV are often closely tied. Let exact and approximate node scores be denoted by $S_u^k(v)$ and $\hat{S}_u^k(v)$ respectively, where the scores are forced to zero if $v \notin T_k^u$ and $v \notin \hat{T}_k^u$. A node pair $v, w \in T_k^u \cup \hat{T}_k^u$ is *concordant* if $(S_u^k(v) - S_u^k(w))(\hat{S}_u^k(v) - \hat{S}_u^k(w))$ is strictly positive, and *discordant* if it is strictly negative. It is an *exact-tie* if $S_u^k(v) = S_u^k(w)$, and is an *approximate tie* if $\hat{S}_u^k(v) = \hat{S}_u^k(w)$. If there are c, d, e and a such pairs respectively, and m pairs overall in $T_k^u \cup \hat{T}_k^u$, then Kendall's τ is defined as

$$\tau(k,u) = \frac{c-d}{\sqrt{(m-e)(m-a)}} \in [-1,1].$$

Unlike Fogaras *et al.*, we do not limit to pairs whose scores differ by at least 0.01 or 0.001, so our τ is typically smaller.

Appendix B

Notations

The following table shows the notations that we have used in this report. Unless otherwise stated specifically, the symbols stand for the descriptions as given below.

r,t	Teleport vector
p_r	PageRank vector with respect to a teleport vector r
δ_u	Column vector with 1 at index u and 0 otherwise
C	Transpose of the normalized adjacency matrix, such that columns of C add upto 1
p_r	PageRank vector corresponding to the tepeport vector r.
α,c	walking probability = 1 - teleport_probability = 0.8(default)
V	set of all entity nodes in the ER graph
N_u	total number of the fingerprints started from node u
$PPV(u, v)$	the v^{th} coordinate of PPV(u)
Prox(i,j)	Directed Proximity from node i to node j
$P(i, j)$	probability that random surfer moves from node i to node j
D	diagonal matrix of node outdegrees
W	weight matrix for the graph
ϵ	PageRank termination criteria – 1e-6
q	residual vector

References

[Ber07] Pavel Berkhin. Bookmark-coloring approach to personalized pagerank computing. *Internet Mathematics*, 3(1), January 2007. Preprint.

[BHP04] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. Objectrank: Authority-based keyword search in databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *VLDB*, pages 564–575. Morgan Kaufmann, 2004.

[Bis06] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[BNH+02] Gaurav Bhalotia, Charuta Nakhe, Arvind Hulgeri, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[Bra06] Matthew Brand. Fast low-rank modifications of the thin singular value decomposition. Technical Report TR 2006-059, 2006.

[CDK+03] Steve Chien, Cynthia Dwork, Ravi Kumar, Daniel R. Simon, and D. Sivakumar. Link evolution: Analysis and algorithms. *Internet Mathematics*, 1(3):277–304, 2003.

[Cha00] Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. *Journal of the ACM*, 47(6):1012–1027, 2000.

[Cha07] Soumen Chakrabarti. Dynamic personalized pagerank in entity-relation graphs. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 571–580, New York, NY, USA, 2007. ACM Press.

[Coh97] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[FR04] D. Fogaras and B. Rácz. Towards fully personalizing PageRank. In *Proceedings of the 3rd Workshop on Algorithms and Models for the Web-Graph (WAW2004), in conjunction with FOCS 2004.*, 2004.

[JW03] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 271–279, New York, NY, USA, 2003. ACM Press.

[KK95] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, 1995.

[LM05] Ronny Lempel and Shlomo Moran. Rank-stability and rank-similarity of link-based web ranking algorithms in authority-connected graphs. *Information Retrieval*, 8(2):245–264, 2005.

[MCW95] B.S. Manjunath, S. Chandrasekaran, and Y.F. Wang. An eigenspace update algorithm for image analysis. In *SCV95*, page 10B Object Recognition III, 1995.

[Pat07] Amit Pathak. Indexing and query processing for text and structured search. Technical report, IIT Bombay, 2007.

[PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[SZ89] Karen A. Stephenson and Marvin Zelen. Rethinking centrality: Methods and examples. *Social Networks*, 11:1–37, 1989.

[TF06] Hanghang Tong and Christos Faloutsos. Center-piece subgraphs: problem definition and fast solutions. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 404–413, New York, NY, USA, 2006. ACM Press.

[TFP06] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Fast random walk with restart and its applications. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pages 613–622, Washington, DC, USA, 2006. IEEE Computer Society.

[TKF07] Hanghang Tong, Yehuda Koren, and Christos Faloutsos. Fast direction-aware proximity for graph mining. In *KDD '07: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, New York, NY, USA, 2007. ACM Press.

[WS03] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 266–275, New York, NY, USA, 2003. ACM Press.