Technical Report

Department of Computer Science and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 08-024

I/O efficient computation of First Order Markov Measures for Large and Evolving Graphs

Prasanna Desikan and Jaideep Srivastava

July 21, 2008
I/O efficient computation of First Order Markov Measures for Large and Evolving Graphs

Prasanna Desikan and Jaideep Srivastava

Department of Computer Science, University of Minnesota, 200 Union Street SE, 55455 Minneapolis, MN, United States of America
{desikan, srivasta}@cs.umn.edu

Abstract. First order Markov measures, such as PageRank, have gained significance as relevance measure in domains where data is represented as a graph. The large scale of such graphs in real world, such as the World Wide Web has given rise to computing challenges of such measures. In this paper, we address the challenges of computing such First-order Markov measure, considering PageRank as the example of such a measure. We address two challenging computational scenarios for PageRank: (a) computation for a large single graph at a given time instance and (b) incremental computation for large evolving graphs. We achieve efficiency by reducing the problem size and reducing the number of iterations to compute. For (a) we bin the nodes in different partitions and for the subgraph formed by each of these partitions we use the nature of the first-order Markov model to break down the problem of computation. For (b) we propose a method to accommodate the changed edges and nodes into new partitions and existing partitions and identify the subset of partitions for which recomputation is necessary. For each identified partition we use an incremental approach to compute the measure in expedited manner. Our results show significant reduction in time for computing for our approaches to both these problems.

1 Introduction

Web mining research has evolved a long way since its initial categorization into: Web content, Web structure and Web usage [4]. There has been extensive research on each of these categories [5, 19, 20]. Hybrid approaches have been developed and applied for ranking [22] and advanced usage analysis [18]. Our focus is in this paper is applicable to Web structure mining and more generally referred to as link analysis. Link analysis techniques have become one of the principle tools of Web mining. Various link based relevance measures such as PageRank [23] and Kleinberg’s’ hubs and authorities [16] have proven their significance over the years in search applications. PageRank, in particular, has gained significant prominence leading to different variations addressing different scenarios such as ranking based on topic or usage [12, 22].

Efficient computation approach for PageRank first gained attention with a block based technique proposed by Haveliwala [11]. The various aspects related to PageRank computation have been extensively covered in recent surveys [1, 2, 3, 11, 14, 17]. Our work focuses on work reduction techniques to improvise PageRank computation.
for a large single graph at a given time instance and for large evolving graphs. Our earlier works addressed the need for studying evolving graphs [6] and also proposed efficient computation methods for in-memory computations [7, 8]. Study of trends in Web content [24] and Web usage [21] over time has also captured the attention of researchers since then.

**Motivation:** PageRank computation is intensive due to iterative matrix computations of the power method. The computational cost is dominated by two key factors for very large graphs. Firstly, the disk I/O dominates the computation time for a single iteration of PageRank computation. The number of I/Os required for reading the graph and the source and destination vectors largely depend on the memory available and the manner in which the graph is partitioned or binned for computational purposes. Secondly, due to the nature of PageRank computation the numbers of iterations to converge are large and dominate cost. Hence, in order to reduce computation for large graphs, one has to focus on reducing the I/O (by parallelizing the computation or using work reduction techniques) and reducing the number of iterations to converge.

**Contributions:** In this paper we have two key contributions. Firstly, we propose techniques to use our divide and conquer approach [7] on block PageRank [11] to help expedite the process of computing of PageRank for individual bins and thus help in accelerating the convergence of PageRank computation. This approach is very generic and can be applied to any graph without requirement of knowledge of the domain it represents to group nodes together. The approach is independent of the binning strategy and an efficient binning strategy can possibly improve the computational cost further.

In the second part of our work, we address the issue of computation on evolving Web graph. We propose an approach to accommodate the changes to the graph in an evolved time instance to existing partitions as well as new partitions. We then discuss on how the computational time can saved by focusing only on those partitions for which the PageRank has to be locally computed because of the effect of change in the graph. Once a partition has been identified for re-computing, we use incremental PageRank [8] approach to speedup the time to compute for that partition. Our results illustrate the significance of our work in saving the computational cost.

**Paper Organization:** Section 2 describes the overview of PageRank and naive PageRank [23] and block PageRank computation methods [11] that form the basis for the computation in our approach. We have given a brief description of some our earlier work for efficient PageRank computation when the graph fits in the memory [7, 8] that we use in our current approach in Section 3. We refer the reader to those papers for detail descriptions. Section 4 discusses our approaches for improving computing a single large static graph and evolving graphs which are our key contribution. Section 5 presents our results and analysis. We finally conclude and provide future directions of research in Section 6.
2 PageRank Computation

PageRank is a measure for ranking hypertext documents that determines their quality. It was originally developed by Page et al. [24] for the popular search engine, Google [10]. The key idea is that a page has high rank if it is pointed to by many highly ranked pages. PageRank computation is carried out using the iterative power method where the graph is represented as an adjacency matrix. The existence of a stationary vector of PageRank in the iterative power method is guaranteed only if the Web graph is strongly connected and is aperiodic. To ensure the condition of strong connectedness, the dampening factor is introduced, which assigns a uniform probability to jumping to any page. In a graph theoretic sense it is equivalent of adding an edge between every pair of nodes with a transition probability of d/n. The aperiodic property is also guaranteed for the Web graph. Different approaches have been taken to handle dangling nodes, which act as a rank sink, such as to iteratively remove all the nodes that have an outdegree of zero, to remove the dangling nodes while computation initially and add them back during the final iterations of the computation [11], to add self loops to dangling nodes [13, 9] and to add links to all nodes in the graph, G from each of the dangling node. We handle dangling nodes by adding self loops to all nodes. Once these issues are taken care of the convergence of the PageRank vector using power method is guaranteed.

![Diagram of PageRank Computation](image)

Fig. 1. Basic PageRank Computation Approach

Figure 1 illustrates the cost of a PageRank computation using the basic algorithm. The destination vector resides in the memory. During a single iteration, each line read from the link file representing the adjacency matrix, the source node and the list of nodes in its adjacency list is read. Then the contribution of source node to these lists of nodes is then computed using the PageRank score of the source node, stored in the source vector file, from the previous iteration and its outdegree. And then again the
next line is read and the process repeats till all the lines of the link files are read. At the end of this iteration PageRank scores of all the nodes are computed and stored in destination vector in the memory. This destination vector now has to be written to disk so that it can be used as the source vector for the next iteration. Hence the total I/O cost would be: $|V_s| + |L| + |V_D|$.

Haveliwala [11] proposed a memory efficient approach of computing PageRank when the destination vector entirely does not fit in the memory. The key idea behind this approach was to partition the graph in such a way that each partition contains nodes whose PageRank will be computed and the edges among these nodes and also the nodes from other partitions which have edges to nodes in this partition. This ensures that in a partition we have a set of nodes whose all incoming nodes are captured, which translates into the fact that for a given set of nodes the PageRank contribution from all its incoming neighbors are captured. And if the size of these nodes can fit in the memory, regular PageRank algorithm could be applied to compute the PageRank scores of these nodes. And this process has to be repeated for each partition thus ensuring the complete computation of PageRank scores of all partitions. In this approach, a given node’s children are split so that each set in the split belongs to
one of the partitions which are designated for a set of destination nodes to which these children belong to. For example form Figure 2 we can observe, 3→4, 5, 9 is split as 3→4, 5 of partition B and 3→9 of partition C This split results in increased storage in terms of the Adjacency matrix and hence the number of I/Os required for reading the graph increases. Also the source vector has to be read for computation in every partition. If there are ‘n’ number of partitions the total I/O cost is : n[|V_d| + |V_d|] + |L| (1 + δ) where δ account for the number of additional lines in the graph file due to splitting the adjacency list of a node according to the partitions.

3 Theoretical background

We present in this section how we leverage that PageRank is based on first order Markov model for designing a partitioning scheme to improve computation. To compute measure such as PageRank on a graph, if a node belonging to a set of nodes cannot be reached from a node belonging to any other set, then the score on this node would depend only on the nodes of the set to which it belongs. This leads to the idea that PageRank of the nodes belonging to a set A, does not depend on the PageRank of the nodes from another set B if there is no incoming links from set B to set A.

![Fig. 3. Partitioning based of First Order Markov Nature](image)

We make use of this criterion, to divide the graphs into partitions of “red patches” and “yellow patches” such that (i) there is no link that point from any “red patch” to another “red patch” or “yellow patch” and (ii) there are no outgoing links from a “yellow patch” to any of the red patches. Once we partition the graph in such a manner into sets of “red patches” and “yellow patches”, we can then compute the PageRank of nodes in the “red patches” independently. Such an approach has two advantages. Firstly, it reduces the size of the problem by reducing the size of the graph into smaller subgraphs of “red patches” and “yellow patches” each of which may have
different convergent rates. Secondly, since the computation of "red patches" can be carried out independently, this process can be parallelized leading to further optimization and saving on computation time. In Figure 3 the graph $G$ is partitioned into four partitions such that $G_1, G_2, G_3, G_4$ correspond to the "red patches" discussed earlier. $G_2$ corresponds to the "yellow" patch.

The approach to partitioning the graph to satisfy the above conditions is to start with a node and do a 'reverse' BFS (meaning BFS following incoming links) and identify a patch as a 'red patch'. For a single graph, the start node can be random each time we try to identify a "red patch". The graph that remains is considered a "yellow" patch. For incremental computation, the start node for reverse BFS is from the set of changed nodes and we continue till all nodes in the changed set are visited. For further details we refer the reader to the original work [7, 8].

![Fig. 4. Overview of our approach for PR computing for a Single Large Graph](image)

### 4 Our Approach

In our approach to improve the efficiency of I/O based computation of PageRank we firstly categorize the problem based on the two scenarios:
• Single Large Graph
• Evolving Graphs

PageRank computation can be made efficient by parallelizing computations or reducing the work to compute. Our approach to both classes of problems mentioned above is to reduce the amount of work to compute. In order to do so, we consider reducing the key bottleneck of PageRank computation - number of iterations required to converge. For a single iteration all the edges of the graph have to be traversed, and hence the cost of a single iteration cannot be improved. However, the overall number of iterations can be reduced by leveraging the fact certain portions of the graph converge faster and that there are faster approaches to compute PageRank when the graph fits in the memory. In the following subsections we discuss how we approach each class of problem to make the computation more efficient. It should be noted that while our approach for a single graph is similar to the approach proposed by using the block-structure of the web [15], our approach is more generic not restricted to a graphs from the Web where blocks can be approximated as doing the directory structure of the Website. Secondly, our approach relies on our earlier work of using a divide-and-conquer approach [7] and incremental pagerank computation [8] to expedite PageRank computation of the local partitions. Also the related work mentioned [15] does not address the issue of evolving graphs.

4.1 Single Large Graph

Once the memory configuration is known, the graph can be partitioned according to either the number of entries from the destination vector or according to the size of a subgraph that will fit in the memory. Different graph clustering and graph partitioning can be applied to find a partitioning that eventually minimizes the δ factor. However, one should also consider the cost of partitioning the graph with respect to the cost of PageRank computation. Also most graph partitioning and clustering algorithms assume the graph to fit in the memory, which is not true for large graphs such as the Web. We assume the nodes to be binned as the graph is being crawled depending on the predetermined number of nodes the bin can hold. While this may not be the optimal binning strategy this ensures virtually no additional binning or partitioning cost for PageRank computation.

The next step is to compute the PageRank of nodes belonging to a bin (or the subgraph formed due to the binning) using the divide and conquer approach [17]. The divide and conquer approach relies on the fact that certain portions of the graph can be identified such that the PageRank of those portions can be computed independently. This overview is illustrated in Figure 4. The first step is to identify portions of graphs whose PageRank computation is not affected by rest of the graph as there are no incoming links to these portions from the rest of the graph. This can be achieved by doing a reverse BFS from a randomly picked node. For further details, we refer the reader to the original paper [17]. The computation of PageRank for each partition has to be completed to obtain the approximate PageRank Vector which can be used as starting point for the global computation. In our experiments we choose not to use
this approach and compute PageRank of each partition sequentially as the focus of the work is on work-reduction techniques and not parallel computing.

Fig. 5. Capturing graph evolution as changes across old partitions and new partition

A key strategy we apply here is to limit the number of iterations for the computation of local PageRank by specifying a lower threshold for the L1 norm or setting a low threshold for the maximum number of iterations. Finally we take the PageRank vector obtained by the above computations and use that as the starting vector and apply the memory-efficient block PageRank approach proposed by Haveliwala to compute the PageRank. In doing so, the PageRank vector converges at a faster rate and reducing in the overall number of iterations that saves more time than the time spent on computing the PageRank for the individual partitions using the divide-and-conquer ap-
proach. We underline the fact that the approach is generic and it does not rely on grouping nodes by domain. However, at the same time our approach is open to any such grouping that can improve the performance additionally.

4.2 Evolving Graphs

The next focus is on computation of PageRank for evolving graphs. We keep our scope of the problem to changing of graph from one time instance to the next time instance. We do not expand our scope to computing simultaneously the PageRank of series of graphs over a time period.

We propose an incremental computation method for computing PageRank of graph $G$, which has evolved from a time instance $t$ to $t + \Delta t$. We refer to nodes and edges at time instance $t$ as ‘old’ and nodes and edges that have surfaced in time instance $t + \Delta t$ as ‘new’ in our discussions below. The first step is to identify the portions of the graphs that have changed, which can be pipelined with crawling. The changes can be broadly considered into three categories:

**Changes exclusive to new nodes:** For all the new nodes in the new time instance, this case involves only edges between the new nodes. This can be viewed as a new subgraph that is formed due to the evolving graph. This is depicted in Figure 5 by nodes ‘12’, ‘13’, ‘14’ and ‘15’ and the edges between them. These nodes are considered to belong to a new partition ‘D’. The destination vector in this partition will consists only of these new nodes.

**Changes exclusive to old nodes:** These changes include change in the portion of the graph that existed at time instance $t$. These can include addition of new edges within a partition or between partitions or deletion of existing edges. In Figure 5 these are depicted by new edge between node ‘5’ and node ‘6’ and an deleted edge between node ‘1’ and node ‘2’. The only changes these will result is in the change in the link file size of memory efficient PageRank computation which will increase with new edges and decrease with deleted edges. The destination vector of a particular partition will still remain the same if only the edges change. New nodes are added into a new partition and if any nodes are deleted, the size of the destination vector would decrease. However, none of this would impact the fact that the PageRank of these nodes can be computed independently to obtain a global approximation for a good starting vector. The key point to be noted is we would need to recompute local PageRank for all the partitions whose edge information have changed, such as partition A for missing edge between node ‘1’ and ‘2’ and partition B for additional edge between node ‘5’ and node ‘6’ as illustrated in Figure 5.

**Changes involving both old and new node:** The third case involves new edges between the old nodes and new nodes. This can be split further into two cases:

- **Outgoing edge from new node to old node:** This case arises when a new directed edge exists between newly formed nodes to an already existing node. This case is illustrated by the edge between node ‘12’ and node ‘5’ in Figure 5. Such a
scenario does results in the increase in the link file size of the both the new partition and the partition (B) to which the old node (‘5’) belongs to. This would mean we need to re-compute the PageRank scores of the nodes in the partition in the previous time instance (B) in addition to computing the PageRank scores of the new partition (D).

*Incoming edge from old node to new node:* This case arises when there is a new edge from an old node to a new node. This is illustrated by the edge from node ‘8’ to node ‘15’ in Figure 5. This would mean that we need to compute only the PageRank of the new partition (D) to accommodate this change.

In the above paragraphs we have discussed approaches to bin the new nodes into a new partition and how to accommodate all other changes with existing partitions. We have also pointed on how these changes are going to affect the partitions and as a result identify the partitions that would need re-computing. As a final step, we use incremental PageRank algorithm proposed in our earlier work [18] to re-compute the local PageRanks of the partitions that have undergone a change. And we use the basic or memory-efficient approach to compute the local PageRank of the new nodes. We use these results to start as a good approximation to Global PageRank vector. Also, if there are partitions which remain totally unaffected because there has been no changes within the partition and also there is no incoming edge to the partition, we can skip those partitions use the ideas of incremental page rank to compute only for the remaining partitions and scale the values accordingly to obtain the PageRank.

5 Experiments and Results

We briefly discuss our experimental set up, results and our findings in this section. We used GT Graph generator to generate graphs of various sizes for our experiments. The number of edges in a graph is a key parameter, as the GTGraph generates a graph with as specified edges there are only certain numbers that can be the order of the graph to satisfy power-law requirements. Hence, the order of the graphs in our experiments is not nicely rounded numbers following a linear scale, similar to the variation in size of the graph. In this paper we discuss our experiments and results only for power-law graphs as Web graphs follow power-law distribution. In our first set of experiments, we generate graphs of size 20 million to 60 million and partition and bin the nodes into 10 partitions. It should be noted that the size of each partition is not necessarily equal since our binning criteria is based on the number of nodes in the local destination vector.
Table 1. Computation time of PR using different approaches for Single Static Graphs

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
<th>Num Bins</th>
<th>BPR-Iter</th>
<th>BPR-Time</th>
<th>DAC-Time</th>
<th>% Time Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>5572643</td>
<td>2E+07</td>
<td>10</td>
<td>50</td>
<td>5993.55</td>
<td>1042.67</td>
<td>21.21</td>
</tr>
<tr>
<td>6336079</td>
<td>3E+07</td>
<td>10</td>
<td>51</td>
<td>8098.80</td>
<td>1407.60</td>
<td>24.69</td>
</tr>
<tr>
<td>6810171</td>
<td>4E+07</td>
<td>10</td>
<td>55</td>
<td>11645.55</td>
<td>2085.33</td>
<td>26.58</td>
</tr>
<tr>
<td>7132733</td>
<td>5E+07</td>
<td>10</td>
<td>59</td>
<td>15615.33</td>
<td>3128.00</td>
<td>25.55</td>
</tr>
<tr>
<td>7363573</td>
<td>6E+07</td>
<td>10</td>
<td>62</td>
<td>19691.20</td>
<td>4379.20</td>
<td>25.34</td>
</tr>
</tbody>
</table>

Fig. 6. Plot of computation of PageRank using Memory Efficient Approach (Block Page Rank) versus our approach

Table 1 reflects the actual order and size of the graphs and the various run times. Figure 6 shows the perspective of how much we save and what the trend of saving is as the size of the graph increases. Our experiments reveal that the block Page Rank ('BPR') requires more time for PageRank computation than using the divide and conquer approach (DAC) to individual partitions. To arrive at an approximate global PageRank we set the threshold of DAC approach to be 10^4 or maximum of 30 iterations. This enables saving time on computing local partitions but at the same time arrive at a good approximation. Our threshold for convergence of actual PageRank is set to be 10^4. Using our approach we save around 25% of the time required to compute the PageRank using the block PageRank method. This percentage of time saved using our approach (DAC-BPR Total Time) grows slightly with increase in the size of the graphs and seems to stabilize thereafter. This is due to the fact that the total time is dominated by the computation of the PageRank after determining the approximate global PageRank Vector. However, we think two factors that might affect the performance variation is the size of the bin and the number of partitions.
Table 2. Experimental setup for incremental computation

<table>
<thead>
<tr>
<th>Graph (t)</th>
<th>V (First)</th>
<th>E (First)</th>
<th>Graph (t+1)</th>
<th>V (Second)</th>
<th>E (Second)</th>
<th>Bins Affected</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>5572643</td>
<td>2000000</td>
<td>G2</td>
<td>6336079</td>
<td>30000000</td>
<td>10</td>
</tr>
<tr>
<td>G2</td>
<td>6336079</td>
<td>3000000</td>
<td>G3</td>
<td>6810171</td>
<td>40000000</td>
<td>10</td>
</tr>
<tr>
<td>G3</td>
<td>6810171</td>
<td>4000000</td>
<td>G4</td>
<td>7132733</td>
<td>50000000</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3. Computation of PR using various approaches

<table>
<thead>
<tr>
<th>Graph Evolution</th>
<th>BPR</th>
<th>BPR-DAC</th>
<th>IPR</th>
<th>%Save from BPR</th>
<th>%Save from BPR-DAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1-&gt;G2</td>
<td>8098.8</td>
<td>6099.60</td>
<td>5489.64</td>
<td>32.22</td>
<td>10.00</td>
</tr>
<tr>
<td>G2-&gt;G3</td>
<td>11645.33</td>
<td>8549.87</td>
<td>6912.88</td>
<td>40.64</td>
<td>19.15</td>
</tr>
<tr>
<td>G3-&gt;G4</td>
<td>15615.33</td>
<td>11469.33</td>
<td>9596.05</td>
<td>38.80</td>
<td>16.68</td>
</tr>
</tbody>
</table>

Fig. 7. Incremental computation of evolving graphs

For our experiments on evolving graphs, we experimented essentially on three data points. Graph changing size from 20 million to 30 million, graph changing size from 30 million to 40 million and graph changing from 40 million to 50 million (see Table 2). Our results shown in Table 3 indicate that the computation with respect to basic computation of PageRank is significantly lesser – saving from 30% to 40% of the time. However, the time we save from performing a PageRank computation on the new instance using the approach we proposed for single static graph discussed in the previous section is not that high – with savings of approximately 10%. However, we assumed a huge change in the percentage of the size of the graph, and the number of partition being small, the changes affected all partitions in some way. This resulted in computation of PageRank of all partitions. If the percentage of change is less and the graph is initially divided not many more partitions, we could reduce the number of partitions that need to be recomputed drastically and save much more time. We hope more experimentation might shed more light on this issue in the next couple of months.
6 Conclusions and future directions

In this paper we have addressed the problem of efficient computation of PageRank for which the entire graph cannot fit in the memory. In this respect we have addressed two classes of problems: computation for a large single graph and computation for large graphs that evolve. Our approaches are based on the criteria that reducing the number of iterations by starting with a PageRank vector close to the final PageRank vector can save time. While this idea of saving on iterations is itself not new, the way we approach the problem using the novel divide and conquer approach for individual partitions and proposing schemes that are independent of the nature of the graph (power-law or random) and independent of the domain the graph represents (such as the Web) are our key contribution for efficient computing of single large static graph. Our approach for efficient computation of evolving graphs is a key contribution that shows the effectiveness of our scheme to bin the changed portion and use our earlier incremental page rank version for computing local PageRank on individual partitions that have changed. Our experiments show that we save on computation even though our current experiments are not designed to show the best effects of our approach by presenting a more realistic scenario of change. We believe that further improvement can be achieved by parallelizing the computation of individual partitions instead of sequentially computing as we do so in our current experiments. We have demonstrated that we can save time without having to explicitly use any binning strategy such as grouping by domains and computing block ranks. However, domain specific low-cost binning strategies, such as grouping nodes by website domain in the Web domain, can help improve the overall performance as revealed by earlier work [15].

An important area of future research is on an ideal partitioning or binning strategy that helps in improving the computation time. For example, we know that we can find clusters of graphs using graph partitioning approaches it would reduce the number of inter-partition edges and hence the cost of PageRank computation by reducing the δ factor mentioned in section 3. There exists many graph partitioning algorithms that assume the graph to fit in the memory. In our case, if we can fit the graph in the memory, we could use the basic divide and conquer approach to compute the PageRank. Hence, it would be interesting to explore I/O based graph partitioning techniques that can help in saving the overall computation of PageRank like measures. Another key area of future work is to see how some of these techniques can be extended for other link based relevance measures, such as hubs and authorities.

7 Acknowledgements

This work was supported by NASA under award NNX08AC36A.
References

19. B. Liu and K. C. C. Chang, "Editorial: Special Issue on Web Content Mining" SIGKDD Explorations special issue on Web Content Mining, Dec, 2004