FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 41 (2016) No. 1
DE GRUYTER OPEN
DOI: 10.1515/fcds-2016-0002
ISSN 0867-6356
e-ISSN 2300-3405

AN EFFICIENT MONTE CARLO APPROACH TO COMPUTE PAGERANK FOR LARGE GRAPHS ON A SINGLE PC

Tomohiro SONOBE * †

Abstract. This paper describes a novel Monte Carlo based random walk to compute PageRanks of nodes in a large graph on a single PC. The target graphs of this paper are ones whose size is larger than the physical memory. In such an environment, memory management is a difficult task for simulating the random walk among the nodes. We propose a novel method that partitions the graph into subgraphs in order to make them fit into the physical memory, and conducts the random walk for each subgraph. By evaluating the walks lazily, we can conduct the walks only in a subgraph and approximate the random walk by rotating the subgraphs. In computational experiments, the proposed method exhibits good performance for existing large graphs with several passes of the graph data.

Keywords: Graph, Monte Carlo, PageRank, Random Walk

1 Introduction

The PageRank [20] is one of basic metrics for calculating importance of each node in a graph. Originally it used to rank Web pages in the search engine though, today it is applied to other networks such as bioinformatics [18] and image categorization [21].

There are mainly two ways to calculate the PageRank. One is power iteration. PageRank is the stationary distribution of the random walk with random jumping with probability c, called the teleportation probability. In reverse, the walking goes on to outgoing edges from the current node with probability $1 - c$. Assume that we have a weighted directed graph $G = (V, E)$ with n nodes and m edges. We denote a weight of an edge as $w : V \times V \to \mathbb{R}^+$, satisfying $\sum_{v|(u,v)\in E} w(u, v) = 1$ for a node u.

*National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan
†JST, ERATO, Kawarabayashi Large Graph Project, Japan (e-mail:tominlab@gmail.com)

In general, the PageRank π for node v is calculated as follows:

$$\pi(v) = c\delta(v) + (1-c) \sum_{u|(u,v)\in E} \pi(u)w(u,v) \quad (1)$$

Where $\delta(v)$ is a probability to be selected as the destination node from the random jumping and is set to $1.0/n$. The teleportation probability c is set to 0.15 in [20]. This calculation can converge in several dozen of iterations.

The other way is Monte Carlo method [1]. It simulates real random walks with fingerprints and approximates the PageRank with performed random walks. From a node v, one of adjacent nodes of v is selected with the probability proportional to the weight of the edge leading to a given node and then move to the node as a next walk. In [1] the authors proposed several methods, and a random walk starting from each node with stopping at dangling nodes exhibited a good performance from a point of view of approximation. The PageRank value is computed by dividing the number of visits of each node by the total number of the visits. The advantage of the Monte Carlo method is its applicability for non-standard variants of PageRank (e.g. random surfing with "back button" [22]), where the modification of the power method is not obvious or (sometimes) impossible. Besides, the simulation of random walking is easy to parallelize.

The Monte Carlo simulation needs random access for the target graph. If the PC has sufficient physical memory, the graph data can be stored in the memory and we are able to implement the Monte Carlo algorithm in a direct fashion. However, today there are large graphs that consist of over billions of nodes such as social graphs in SNSs. These graphs cannot fit in the physical memory on a single machine. For this issue, there are some distributed approaches [2, 8]. However, while distributed computational resources such as clouds are popularized, there are difficulties such as partitioning of the graph and communication costs between the machines. We have also problems in optimizing and debugging the distributed algorithms.

In this paper, we propose a simple and efficient PageRank calculation technique based on the Monte Carlo method on a single PC. We assume that the used PC has at least sufficient memory capacity for storing the fingerprints (the number of visits) for each node. We utilize a feature of the Monte Carlo random walk such that it is not necessary to simulate the one walk from start to finish in sequence. It is because the fingerprints depend only the number of visits of the nodes, not the order of visits. We call the walker that moves from node to node in the Monte Carlo simulation as an *agent*. The procedure of our proposal consists of mainly four steps: (1) partition the graph into subgraphs such that each subgraph can fit in the physical memory, (2) set a certain number of agents to each node, (3) load one of the subgraphs into the memory, and (4) simulate the random walk for the agents only in the loaded subgraph and go back to the step (3). We repeat the procedure (3) and (4) for all the subgraphs. We denominated it as an *iteration*. In the experiments, we evaluate our method from three aspects: (1) the way of partitioning the graph, (2) the number of divisions of the graph, and (3) the order of loading subgraphs in an iteration. Our method exhibits robust performance, even if we divide the graph randomly.

This paper is organized as follows. We introduce related work in section 2. In

section 3 we explain our proposal in detail and we show experimental results in section 4. We conclude the paper in section 5.

2 Related Work

One of the Monte Carlo based methods to compute the PageRank is introduced in [1]. This method simulates random walks starting at each node and counts the number of visits for each node. The PageRank is calculated by dividing each visit by the number of the total visits. The authors showed that it achieved good approximations of important nodes with only one agent per node. We use this method as basis of our calculation. Fogaras *et al.* [9] calculated the personalized PageRank by distributed computing of Monte Carlo random walks. They only used start and end points of a path of the walk, while [1] used the complete path. Bahmani *et al.* [3] also used the random walk method to calculate the (personalized) PageRank. They incrementally updated the PageRank by adding some fingerprints, which enabled dynamic updates of PageRanks.

There are some existing works for calculating PageRank with efficient I/O. Sarma *et al.* [7] conducted random walks with a fixed length from sampled nodes and merges them in order to achieve longer walks. The PageRanks were approximated through streams of the graph. Bahmani *et al.* [2] improved that work by doubling the walks in the merging process, and applied to the Map/Reduce framework. Chen *et al.* [5] approximated the PageRank only by focusing on subgraphs. They constructed the subgraphs by estimating the influence of each node. Our method only needs a single PC, and does not confine the walks to certain sets of nodes.

Using the structure of the graphs can speedup the efficiency of the calculation. Kamvar *et al.* [11] utilized the block structure of Web graphs by identifying the host-names of the pages. Furthermore, Kohlschütter *et al.* [13] expanded it to a parallel computation, in which the calculations between same hosts were conducted on a single machine and the communication was needed between different hosts. Wicks *et al.* [23] also utilized the block structure to matrix calculation of power iteration. McSherry [17] proposed various heuristics including a method using the block structure. It updated the PageRank by reiterating the calculation two or three times for intra-hosts. Our method divides the given graph into subgraphs in a simple manner, where it can use the structure of the graph in an indirect way.

There are various studies handling large graphs. Some of them can calculate other metrics than the PageRank such as shortest paths, clusters, diameter, radius, and triangles [6]. Pregel [16] is a framework for large graphs, based on the bulk synchronous parallel model. It divides the graph into partitions, and the calculation is conducted in parallel. PEGASUS [12] is a Hadoop-based system. It executes matrix-vector multiplication for calculating the PageRank, random walk with restart [21], connected component, diameter, and radius. GraphChi [15] also aims at handling large graphs on a single PC. It uses the parallel sliding window (PSW) technique to load edge data in physical memory. However, PSW can handle the algorithms with sequential access, not suitable for random access. Turbograph [10] is a graph engine

aiming at using a single PC. Turbograph is build upon FlashSSD storages and fully utilizes the advantages of I/O parallelism.

Our proposal is specialized in the PageRank calculation by the Monte Carlo random walk for large graphs on a single PC. Applications using the random walk simulation on large graphs can benefit from our method.

3 Proposal Method

The Monte Carlo methods to compute the PageRank were proposed in [1]. It simulates real random walks. From each node in the graph, with probability c or when the current node has no adjacent nodes, the walk is stopped. Otherwise, one of adjacent nodes of the current node is randomly selected and the walk moves to that node. We call the walker in this process as an *agent*. For each node the number of visits is recorded and the PageRank is calculated by dividing the number by the total number of the visits of all the nodes. Our method uses this algorithm as a basis.

If our PC has sufficient amount of physical memory that can store the whole graph data (nodes, edges and weights), we can directly store the graph into the memory and execute the algorithm easily. However as large graphs for example social networks and bioinformatics are available, we can no longer keep all the data in the memory. For these large graphs, we will be busy with memory management since the walk needs random accesses to the graph data. Accessing to secondary storages, i.e. hard disk drives (HDDs), can be a major bottleneck of the efficiency of the Monte Carlo simulation. Our challenge is to ease this problem.

In order to reduce read routines from the HDDs, we convert the random walk from "time series" to "location series". In practice, to calculate the PageRank, only we need to do is to count the number of the visits of each node. It means that we don't have to simulate the walk from start to end in sequence. Thus we only have to move the agents at a specific set of nodes in the memory, and execute this process for all the sets. From such a point of view, we are able to focus on a specific part of the graph, and we are free from a complex memory management by reading the subgraphs in rotation.

We assume that we have at least sufficient memory to store the number of visits and agents of all the nodes. Our proposal consists of four steps.

1. partition the graph into subgraphs such that each subgraph can fit in the physical memory
2. set a certain number of agents to each node
3. load one of the subgraphs into the memory
4. simulate the random walk for the agents only in the loaded subgraph and go back to the step 3

We need to read the whole graph to conduct the fourth step for all the subgraphs, and we call it an *iteration* (conducting third and fourth step for all the subgraphs).

In the fourth step, we only focus on the agents in the loaded subgraph. Some agents go to nodes in other subgraphs. For these agents, we postpone the movements to the future process. In other word, we conduct lazy random walks. We finish the fourth step when there is no active agent in the loaded subgraph. Then, we go on to the next subgraph.

In the first step, we divide the graph into subgraphs and record them in the HDDs. In particular, we divide the set of nodes V into disjoint subsets $\{V_1, V_2, ..., V_d\}$ where $V = \bigcup_{i=1}^{d} V_i$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The corresponding edge set E_i of V_i is defined as follows: $E_i = \{(u, v) \mid (u, v) \in E, u \in V_i\}$. Each subgraph $G_i = (V_i, E_i)$ must fit in the available physical memory. In order to achieve efficient walks, each subgraph should be densely connected. In terms of the edge density, we can use a modularity based community detection algorithm [19] for the graph. However it also needs memory capacity and detected communities cannot always fit in the memory. In following experiments, we apply two partitioning methods to divide the graph into subgraphs by reading the whole graph only once. One is random partitioning. In order to divide the graph into d subgraphs, we assign a number ranging 1 to d randomly to each node, and we distribute each edge into the subgraph by considering the assigned number of the source node. The other is a simple Union-Find algorithm. Initiating each node as a subgraph including only one node and its edges, by reading the edges the Union-Find merges two different subgraphs so that the size of the merged subgraph does not exceed the memory limit. After reading the edges, then we merge the small sized subgraphs with a greedy algorithm. Note that the resulting subgraphs by the partitioning process depend on the order of edges in the graph file.

We show the pseudo code of our algorithm in Figure 1. For a given graph, at first the function "calcPageRank" partitions the graph into subgraphs. The constant "D" stands for the desired number of subgraphs for the graph partitioning. The array "agents" stores the number of agents for each node and "visits" stores the number of visits for each node. Then, the "A" agents are placed to each node. Next, the Monte Carlo simulation is conducted for each subgraph. In the following experiments, we shuffle the order of loading subgraphs and we observe that effect for resulting PageRanks. Note that we abbreviate the loading process of each subgraph from the HDDs for brevity. All the agents in the target subgraphs move randomly until they stop with probability c, reach dangling nodes, or move to the nodes in other subgraphs. While the agents move, the number of the visits is recorded in the array "visits". The number of the iterations is given by the constant "I". We observe the convergence of this algorithm for each iteration in the experiment. Finally, the number of the visits of each node is divided by the total number of the visits, which results in the PageRank value. Note that even though there still can be active agents in the graph after "I" iterations, we regard them as stopped agents and add up them to the "visits".

```
// D: the number of (desired) subgraphs
// A: the number of agents per a node
// I: the number of iterations
calcPageRank(Graph g, Num D, Num A, Num I) {
  // partitioning graph (Union-Find or random)
  Graph[] subGraphs = partitionGraph(g, D);
  mergeSubGraphs(subGraphs);
  // declaration of arrays
  N = the number of nodes;
  agents[N];
  visits[N];
  // initialize agents and visits
  for(i = 0; i < N; i++){
    agents[i] = A;
    visits[i] = 0;
  }
  // simulate the walks
  for(i = 0; i < I; i++){
    // shuffle or no shuffle of loading order
    for each Graph sg in subGraphs{
      for each Node n in sg{
        while agents[n] > 0{
          agents[n]--;
          while n is in sg{
            visit[n]++;
            if n is dangling or random(0.0, 1.0) < c:
              break loop;
            Node nextNode = randomly choose from neighbors;
            if nextNode is not in sg:
              agents[nextNode]++;
            n = nextNode
        }
      }
    }
  }
  for each Node n in g{
    visits[n] += agents[n];
    visits[n] /= total number of visits;
  }
  return visits
}
```

Figure 1: Pseudo code of our algorithm

4 Experiment

We conducted computational experiments [1] to confirm the effectiveness of our method. We used a Linux PC with a Xeon CPU running at 2.8 GHz and 64 GB RAM. We simulated the naive Monte Carlo method and compared the results with our method. Note that because the biggest graph used in our experiment could not fit in the 64 GB memory, we used another machine with 96 GB physical memory for the naive Monte Carlo simulation for that graph. We used a GNU C compiler version 4.6.3.

We observed the effects from three points of view: (1) the way of partitioning, (2) the number of subgraphs, and (3) the order of loading subgraphs. For partitioning the graph, we used two methods, random and Union-Find algorithm. For the number of the subgraphs, we assumed that our PC had a physical memory whose size is 1/10 or 1/100 of the one of the target graph. In fact, we set these values to the constant "D" (10 and 100) in the Figure 1 for the partitioning process. Because the behavior of agents depends on the order of loading subgraphs, we shuffled the order and compared with the result without shuffling.

In terms of evaluation, we used three criteria. The first is average L1 error of resulting PageRanks between our method and naive Monte Carlo random simulation. Let $p(v)$ be the PageRank of a node v of naive Monte Carlo (as correct PageRank) and $\hat{p}(v)$ be the PageRank of our method, and the average L1 error is $\sum_i^n |p(v_i) - \hat{p}(v_i)|/n$. The second is concordance rate. We calculated the rate of concordance of top 10000 PageRank nodes with naive Monte Carlo method. This rate is calculated by $a/10000$ where a stands for the number of nodes that appear both in the top 10000 PageRank nodes of the result of the naive Monte Carlo and our method. It is described in [1] that the nodes with high PageRank are immediately approximated after the first iteration in Monte Carlo simulation. In addition, the top 10000 PageRanks are sufficient amount from a practical perspective. The third is residual agent rate. We count residual agents that remained active after each iteration (one pass of the whole graph), and we calculate the rate, (the number of residual agents) / (the number of initial agents). Note that this rate is always 0 in the naive Monte Carlo method when the simulation finishes.

We used some real-world graphs shown in Table 1. We set the number of agents (the constant "A" in the Figure 1) as 100. The number of resulting subgraphs with the Union-Find method is shown in the columns "#sgD10" and "#sgD100" in the Table 1, where "#sgD10" stands for the number of the resulting subgraphs when "D" was set to 10 and "#sgD100" stands for the number of the resulting subgraphs when "D" was set to 100. Since the Union-Find partitioning method could not precisely divide the graph into 10 or 100 subraphs, the number of the subgraphs exceeded the number.

Table 1: Graphs used in experiments. The columns "#sgD10" stands for the number of the resulting subgraphs in the Union-Find partitioning when "D" was set to 10 and "#sgD100" stands for the number of the resulting subgraphs when "D" was set to 100. The column "File Size" indicates the file size of the graph in text format. The wikipedia-en graph is available at `http://konect.uni-koblenz.de/networks/wikipedia_link_en`.

Name	Node	Edge	File Size	Directed	#sgD10	#sgD100
uk-2007-05 [4]	105M	3.7B	63GB	Yes	11	104
friendster [24]	66M	1.8B	31GB	No	11	106
twitter [14]	42M	1.5B	25GB	Yes	11	106
wikipedia-en	27M	0.6B	9.5GB	Yes	13	112

4.1 The Way of Partitioning

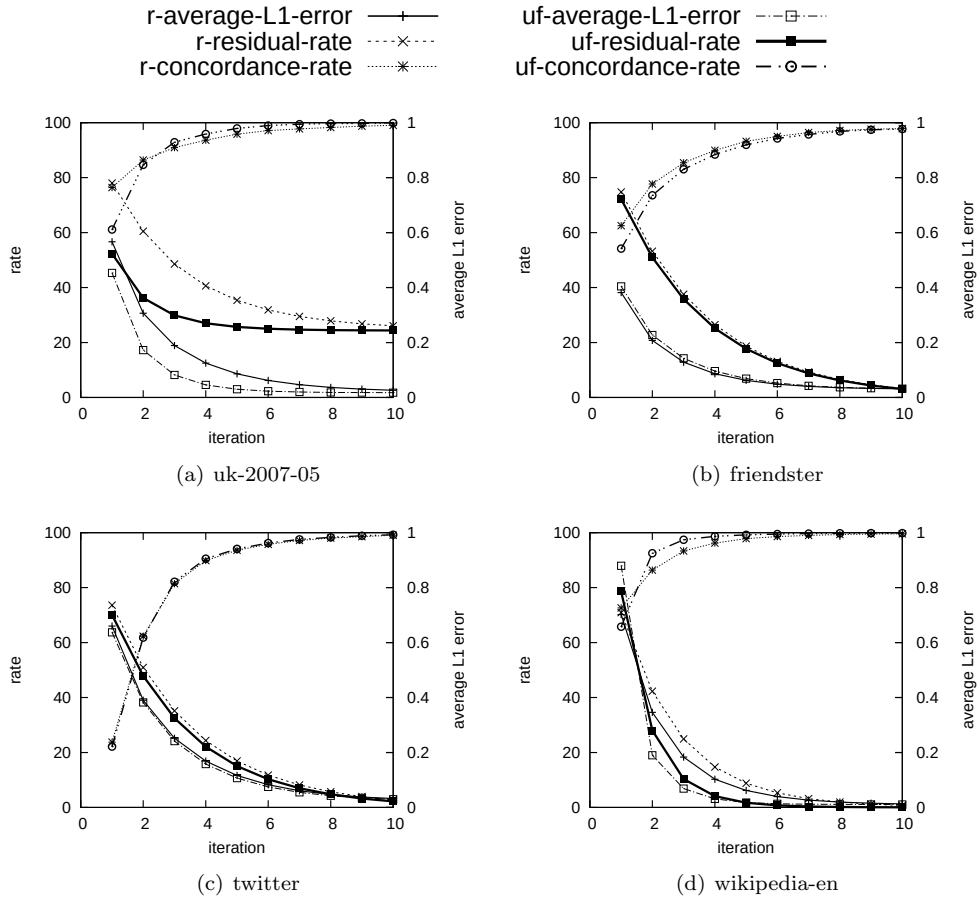
We conducted experiments by using two ways, random and Union-Find, for partitioning the graph. We fixed the number of subgraphs to 10 (or more in the case of the Union-Find method). Naturally, the Union-Find partitioning can generate more densely connected subgraphs than the random partitioning. Intuitively, the walk can be long in a subgraph if the subgraph is densely connected. Thereby, the Union-Find method can approximate the true PageRank more quickly than the random method.

We show the results of the four graphs in Figure 2. As expected, the Union-Find method exhibits lower average L1 error and residual rate except for the friendster graph. In the densely connected graph, the walk can continue longer, and it results in less residual agents and less average L1 error. For the friendster graph, the edges are undirected (i.e. bi-directional) and it can generate denser subgraphs than the other three graphs. This can be one of the reasons why there are little differences on the average L1 error and residual agent rate between the two partitioning methods. We can also see that these differences are large for the largest graph uk-2007-05.

However, the situation is different for the concordance rate. We believe that the order of loading subgraphs affects the accuracy. The agents jump to other subgraphs for each evaluation of a subgraph. Hence the number of agents in the first subgraph is less than in the last subgraph. This bias causes the difference, and it becomes apparent in the Union-Find method at earlier iterations because there are many active agents. We mention the order of subgraphs in subsection 4.3.

Finally, these differences among the three criteria can be uniformized with several iterations. Although we can get more accurate PageRanks with dense subgraphs, we have to face the trade-off between the time for good partitioning and the accuracy.

[1] detailed results are available at `https://sites.google.com/site/tominlab/publications/lmc_fcds_materials`



(a) uk-2007-05

(b) friendster

(c) twitter

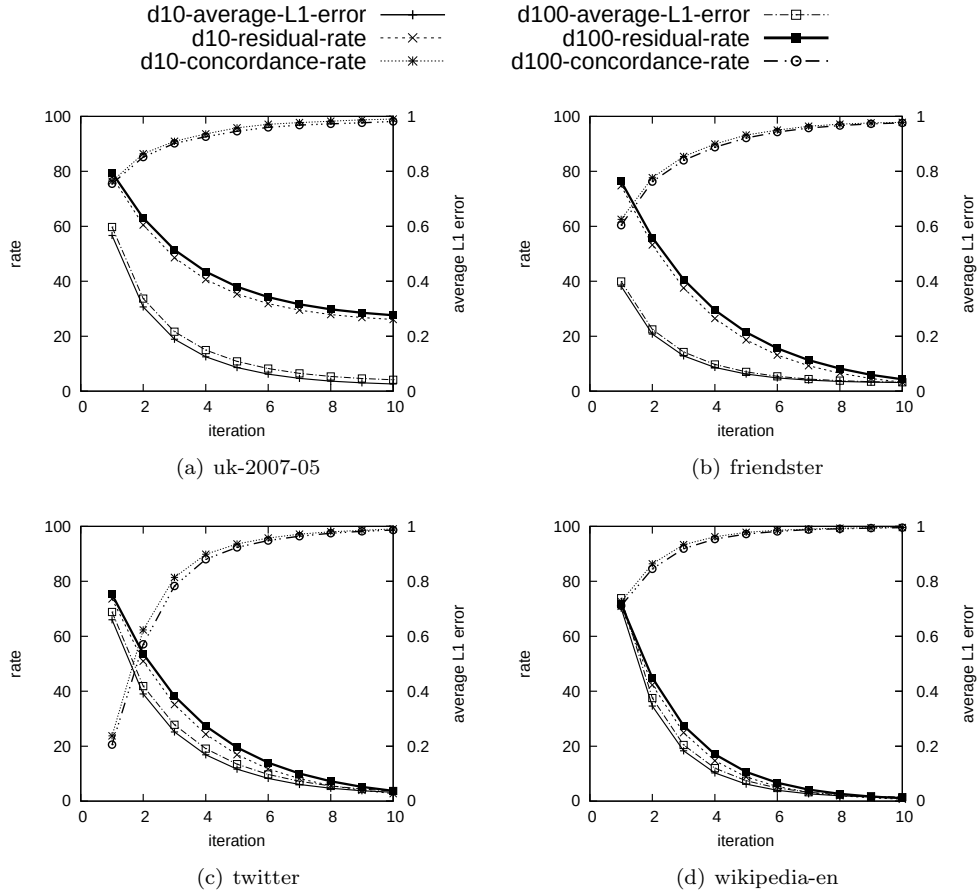
(d) wikipedia-en

Figure 2: The results of partitioning for four graphs. In the legends, the prefix "r" stands for the random partitioning and "uf" stands for the Union-Find partitioning. The number of subgraphs is 10 (or more in the Union-Find). The x-axis indicates the number of iterations. The average L1 error corresponds with the right y-axis and the other two rates correspond with the left y-axis.

4.2 The Number of Subgraphs

We changed the number of subgraphs by setting the parameter "D" in the Figure 1 to 10 and 100. If the number of subgraphs increases, the jumps between the subgraphs can increase. For this reason, the move of agents becomes slow and the approximation of PageRanks can be deteriorated with the increase of the subgraphs.

We show the results in Figure 3. As expected, the average L1 error and residual agent rate are high, and the concordance rate is low for the 100 subgraphs. The accu-



(a) uk-2007-05 (b) friendster

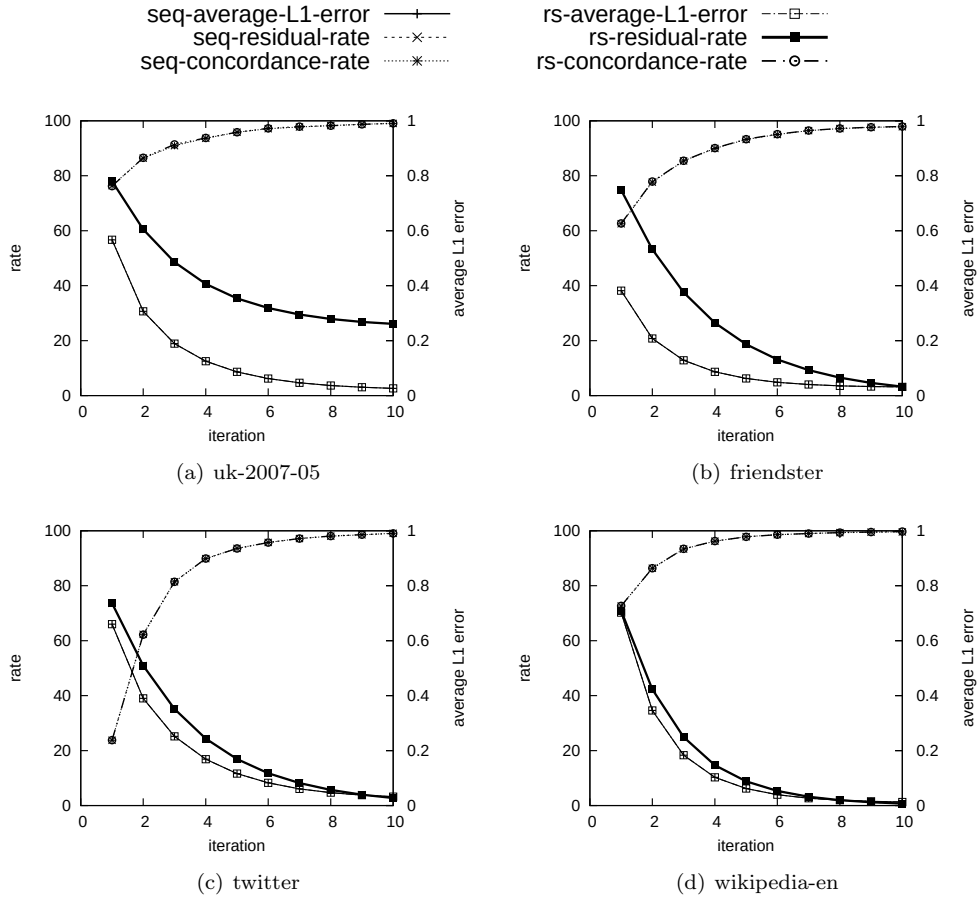
(c) twitter (d) wikipedia-en

Figure 3: The results of the different number of subgraphs for four graphs. In the legends, the prefix "d10" stands for 10 subgraphs and "d100" stands for 100 subgraphs. The partitioning method is random. The x-axis indicates the number of iterations. The average L1 error corresponds with the right y-axis and the other two rates correspond with the left y-axis.

racy can be lessened by increasing the number of subgraphs. However, the differences are relatively small compared to the ratio of the subgraphs (10 : 100). Besides, these differences grow downward with several iterations.

4.3 The Order of Loading Subgraphs

We observed the effect of shuffling the order of loading subgraphs. In the previous experiments, we loaded the subgraphs in sequential order into the physical memory.



(a) uk-2007-05 (b) friendster

(c) twitter (d) wikipedia-en

Figure 4: The results of the different order of loading subgraphs for four graphs. In the legends, the prefix "seq" stands for the sequential order (no shuffle) and "rs" stands for the randomly shuffled order of the subgraphs. The partitioning method is random and the number of subgraphs is 10. The x-axis indicates the number of iterations. The average L1 error corresponds with the right y-axis and the other two rates correspond with the left y-axis.

In other words, we loaded the subgraph numbered 1, 2, 3, ..., d for d subgraphs in each iteration. This order affects the moves of agents. Assume that there is an agent walking along the 3 nodes (a, b, c) that are allocated in subgraph G_1, G_2, and G_3 (a in G_1, b in G_2, and c in G_3). If we load the subgraphs in sequential order, we can completely simulate the walk. However, if we load the subgraphs in the reverse order (G_3, G_2, G_1), we can partially simulate the walk (the agent stops at the node b). We were interested in that effect and we conducted the experiment by changing the orders with sequential and random shuffle. We fixed the partitioning method to

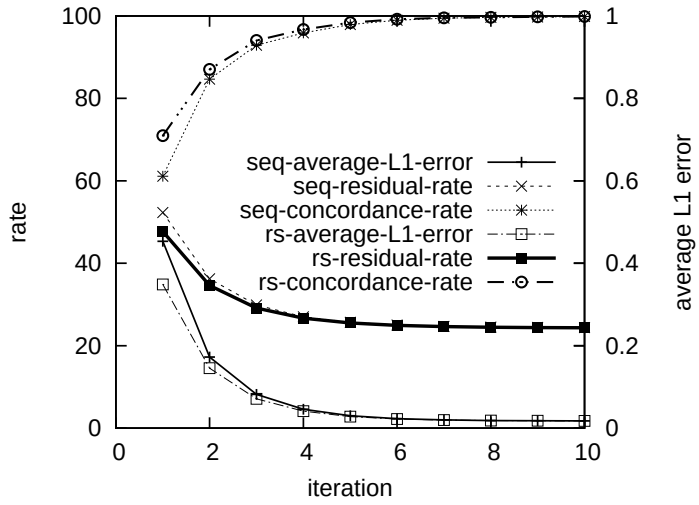


Figure 5: The result of uk-2007-05 with randomly shuffled order. The partitioning method is Union-Find and the number of subgraphs is 10.

the random partitioning and the number of subgraphs to 10.

We show the results in Figure 4. We can see that there is almost no difference in three criteria for four graphs between the two orderings. Since the random partition divides the graph into sparse subgraphs, the length of walk can be short in each subgraph and agents evenly jump to other subgraphs. Thus, if we load the subgraphs in any order, the total number of moves in an iteration can differ only slightly. By contrast, we show the result of Union-Find method in Figure 5 for the uk-2007-05 graph. We can see the difference in the three criteria at the first iteration. This is because the Union-Find method can construct densely connected subgraphs. The agents can jump into other subgraphs that are connected to currently loaded subgraph, not evenly to all the subgraphs. Therefore, the order can affect the moves of the agents more deeply than the random partitioning. Because there are many active agents in earlier iterations, the difference has more impact on the result than later iterations. In sum, although the order of loading subgraphs can affect earlier iterations, we need not mind the order if we conduct several or more iterations.

4.4 The Expected Number of Passes

The expected number of the passes of the naive Monte Carlo random walk is calculated as follows. The total number of the agents is expressed as $A \times n$, where n stands for the number of the nodes in the given graph and A is a constant (set to 100 in the experiment). The average data size for one node (the adjacent nodes of the node) is expressed as r. We assume that the available physical memory is $\frac{1}{D}$ of the graph size

and we do not consider the dangling nodes. Hence the expected number of loading data size for one node from HDDs is $r(1-\frac{1}{D})$. The total loading data size is calculated as follows (note that c is the teleportation probability).

$$\begin{aligned} Anrc(1-\frac{1}{D})\sum_{k=1}^{\infty}k(1-c)^k &= Anrc(1-\frac{1}{D})\frac{1-c}{c^2} \\ &= Anr\frac{1-c}{c}(1-\frac{1}{D}) \end{aligned} \tag{2}$$

Because nr equals the size of the one pass, the expected number of the passes is

$$A\frac{1-c}{c}(1-\frac{1}{D}) \tag{3}$$

In the experiment, for example we used $c = 0.15$, $D = 10$, and $A = 100$. Then, the equation ends up with 510 (passes). Compared with the theoretical number of the passes, our method achieved at least 90 % approximation on the concordance rate with the far fewer passes.

5 Conclusion

We proposed an efficient method to compute PageRanks of nodes in a large graph that cannot fit in physical memory on a single PC. We converted the naive random walk from "time series" to "location series" by moving the agents in a lazy manner. Our method divides the graph into subgraphs that can fin in the memory and simulates the Monte Carlo random walks by focusing on the agents only in the loaded subgraph. In computational experiments, we evaluated our method from three points of views: the ways of partition of the graph, the number of subgraphs, and the loading order of the subgraphs. Although a few percent of agents remained active in the graph, we could obtain a good performance for real-world graphs even though we divided the graph in a simple manner. We also found that we could diminish the drop of accuracy with only several iterations. We believe that conducting graph partitioning methods that trap many agents can boost the efficiency of our method.

Acknowledgement

We appreciate the insightful comments from the FCDS reviewers.

References

[1] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. Monte carlo methods in pagerank computation: When one iteration is sufficient. *SIAM Jounal on Numerical Analysis*, (2):890–904, February 2007.

[2] Bahman Bahmani, Kaushik Chakrabarti, and Dong Xin. Fast personalized pagerank on mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 973–984, 2011.

[3] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized pagerank. *VLDB Endow.*, 4(3):173–184, 2010.

[4] Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, 2004.

[5] Yenyu Chen, Qingqing Gan, and Torsten Suel. Local methods for estimating pagerank values. In *Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management*, pages 381–389, 2004.

[6] Shumo Chu and James Cheng. Triangle listing in massive networks and its applications. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 672–680, 2011.

[7] Atish Das Sarma, Sreenivas Gollapudi, and Rina Panigrahy. Estimating pagerank on graph streams. In *Proceedings of the Twenty-seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 69–78, 2008.

[8] Atish Das Sarma, AnisurRahaman Molla, Gopal Pandurangan, and Eli Upfal. Fast distributed pagerank computation. In *Distributed Computing and Networking*, pages 11–26. 2013.

[9] Dániel Fogaras and Balázs Rácz. Towards scaling fully personalized pagerank. In *Algorithms and Models for the Web-Graph*, pages 105–117. 2004.

[10] Wookshin Han, Sangyeon Lee, Kyungyeol Park, Jeonghoon Lee, Minsoo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 77–85, 2013.

[11] Sepandar Kamvar, Taher Haveliwala, Christopher Manning, and Gene Golub. Exploiting the block structure of the web for computing pagerank. *Stanford University Technical Report*, 2003.

[12] U. Kang and Christos Faloutsos. Big graph mining: Algorithms and discoveries. *SIGKDD Explorations Newsletter*, (2):29–36, April 2013.

[13] Christian Kohlschütter, Paul-Alexandru Chirita, and Wolfgang Nejdl. Efficient parallel computation of pagerank. In *Advances in Information Retrieval*, pages 241–252. 2006.

[14] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, pages 591–600, 2010.

[15] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, pages 31–46, 2012.

[16] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pages 135–146, 2010.

[17] Frank McSherry. A uniform approach to accelerated pagerank computation. In *Proceedings of the 14th international conference on World Wide Web*, pages 575–582, 2005.

[18] Julie L Morrison, Rainer Breitling, Desmond J Higham, and David R Gilbert. Generank: using search engine technology for the analysis of microarray experiments. *BMC bioinformatics*, (1):233, 2005.

[19] Mark E. J. Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

[20] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. 1999.

[21] Jiayu Pan, Hyungjeong Yang, Christos Faloutsos, and Pinar Duygulu. Automatic multimedia cross-modal correlation discovery. In *Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 653–658, 2004.

[22] Marcin Sydow. Random surfer with back step. *Fundamental Informaticae*, 68(4):379–398, 2005.

[23] John Wicks and Amy Greenwald. Parallelizing the computation of pagerank. In *Algorithms and Models for the Web-Graph*, pages 202–208. 2007.

[24] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. In *Proceedings of the ACM SIGKDD Workshop on Mining Data Semantics*, pages 1–8, 2012.

Received 27.01.2015, accepted 30.01.2016