Billion-Node Graph Challenges

Yanghua Xiao, Bin Shao
Fudan University, Micorosoft Research Asia
shawyh@fudan.edu.cn, binshao@microsoft.com

Abstract

Graph is a universal model in big data era and finds its wide applications in a variety of real world tasks. The recent emergence of big graphs, especially those with billion nodes, poses great challenges for the effective management or mining of these big graphs. In general, a distributed computing paradigm is necessary to overcome the billion-node graph challenges. In this article, we elaborate the challenges in the management or mining on billon-node graphs in a distributed environment. We also proposed a set of general principles in the development of effective solutions to handle billion-node graphs according to our previous experiences to manage billion-node graphs. The article is closed with a brief discussion of open problems in billion-node graph management.

1 Introduction

Many large graphs have emerged in recent years. The most well known graph is the WWW, which now contains more than 50 billion web pages and more than one trillion unique URLs [5]. A recent snapshot of the friendship network of Facebook contains 800 million nodes and over 100 billion links [6]. LinkedData is also going through exponential growth, and it now consists of 31 billion RDF triples and 504 million RDF links [7]. In biology, the genome assembly problem has been converted into a problem of constructing, simplifying, and traversing the de Brujin graph of the read sequence [8]. Each vertex in the de Brujin graph represents a k-mer, and the entire graph in the worst case contains as many as 4^k vertices, where k generally is at least 20.

We are facing challenges at all levels from system infrastructures to programming models for managing and analyzing large graphs. We argue that a distributed memory system has the potential to meet both the memory and computation requirements for large graph processing [4]. The reasons are as follows. One distinguishing characteristic of graphs, as compared to other forms of data, is that data accesses on graphs have no locality: As we explore a graph, we invoke random, instead of sequential data accesses, no matter how the graph is stored. To address this problem, a simple solution is to put the graph in the main memory. The size of required memory space to host the topology of a web-scale graph is usually on the terabyte scale. It is still unlikely that it can be stored in the memory of a single machine. Thus, a distributed system is necessary. On the other hand, the computation power required by applications on web-scale graphs are often far beyond the capacity of a single machine. A distributed system is beneficial as graph analytics is often computation intensive. Clearly, both memory and computation demand an efficient distributed computing platform for large graphs.

In this paper, we advocate the use of distributed memory systems as a platform for online graph query processing and offline graph analytics. However, management of billion-node graphs in a distributed paradigm is still very challenging. In this article, we will first elaborate the challenges to manage or mine billion-node graphs in a distributed environment, especially for the distributed memory systems. We further propose a set of general principle to overcome the billion-node graph challenges according to our previous experience to develop scalable solutions on billion-node graphs.

2 Challenges

Even given an efficient distributed memory system, the management of billion-node graph is still challenging. There are two major challenges: *scalability* and *generality*. The two challenges are even more remarkable when the graphs are complicated. However, most real graphs are usually complicated and notoriously known as complex networks. Next, we will first elaborate the two challenges. Then we discuss the difficulties caused by the complicated structure of real graphs.

2.1 Billion-node Graph Challenges

Scalability Most current graph algorithms are designed for small, memory-based graphs. Most of memory based graph algorithms rely one subtile/tricky strategies to produce an optimal or near optimal solution. Consider the graph partitioning problem. A class of local refinement algorithms, most of which originated from the Kerninghan-Lin (KL) algorithm [9], bisect a graph into even size partitions. The KL algorithm incrementally swaps vertices among partitions of a bisection to reduce the edge-cut of the partitioning, until the partitioning reaches a local minimum. The local refinement algorithms are costly, and are designed for memory-based graphs only.

There are two general ideas to improve the scalability of the well-turned memory based algorithm for small graphs. The first is *multiple level* processing following the *coarsening-then-conquer* idea [12]. In *coarsening-then-conquer* idea, we repeatedly coarsen a graph until it is small enough then run corresponding heavyweight well-tuned algorithms on the small coarsened graphs. The second is leveraging parallelism. That is distributing the computation and data onto different machines and use multiple machines to achieve a speedup. However, both of two ideas face great challenges when processing billion-node graphs.

- First, consider the coarsening-then-conquer idea. We use graph partitioning as an example problem to discuss the challenge this idea is confronted with when handling billion-node graphs. Multi-level partitioning algorithms, such as METIS [11] coarsens a large graph by *maximal match* and apply algorithms such as KL and FM on the small graph. However, the assumption that (near) optimal partitions on coarsened graphs implies a good partitioning in the original graph may not be valid for real graphs, when maximal match is used as the underlying coarsening mechanism. Metis actually spent grate efforts refine the partitioning on coarsened graphs. In general, it is not trivial design an appropriate coarsening mechanisms to coarsen a graph without sacrificing useful information in the network.

- Second, let's investigate the challenge to use parallelism when processing billion-node graphs. Both data parallelism and computation parallelism is not trivial for graph data. To partition a graph into k machines to minimize the number of cut edges (i..e, communication cost) is a classical NP-hard problem. Thus, data parallelism is not trivial to be used. To distribute computation over different machines is neither easy. Because the logics of most graph operations or algorithms is inherently dependent, leading to poor computation parallelism. Many graph problems are proved to be P-Complete (i.e. hard to parallelize) problems, such as DFS (depth-first search).

Generality It is important to develop a general purpose infrastructure where graphs can be stored, served, and analyzed, especially for web-scale graphs. Current graph solutions are not built on top of a general-purpose graph infrastructure. Instead, they are designed exclusively and subtly for the purpose of specific tasks. Many solutions are sufficiently tuned for the specific problem settings and specific graph data. As a result, the assumptions, principles or strategies that achieve good performance on certain tasks or graphs are not necessarily effective on other problems or graphs. For example, for the graph partitioning problem, ParMetis [13] can work on graphs of tens of millions of nodes. However, it requires that the graph is partitioned *two-dimensionally*, that is, the adjacency list of a single vertex is divided and stored in multiple machines. This helps reduce the communication overhead when coarsening a graph. However, such a design may be disruptive to other graph algorithms. Even if we adopt this approach, the cost of converting data back and forth is often prohibitive for web-scale graphs.

Then, the question is, is any infrastructure currently available appropriate for web-scale graphs? MapReduce is an effective paradigm for large-scale data processing. However, MapReduce is not the best choice for graph applications [14, 22]. Besides the fact that it does not support online graph query processing, many graph algorithms for offline analytics cannot be expressed naturally and intuitively. Instead, they need a total rethinking in the "MapReduce language." For example, graph exploration, i.e., following links from one vertex to its neighbors, is implemented by MapReduce iterations. Each iteration requires large amount of disk space and network I/O, which is exacerbated by the random access pattern of graph algorithms. Furthermore, the irregular structures of the graph often lead to varying degrees of parallelism over the course of execution, and overall, parallelism is poorly exploited [14, 22]. Only algorithms such as PageRank, shortest path discovery that can be implemented in vertex-centric processing and run in a fixed number of iterations can achieve good efficiency. The Pregel system [15] introduced a vertex-centric framework to support such algorithms. However, graph partitioning is still a big challenge. We are not aware of any effective graph partitioning algorithms in MapReduce or even in the vertex-centric framework.

The generality issue deteriorates when the diversity of graph data and graph computations is taken into account. There are many kinds of graphs, such as regular graph, planar graphs, ER random graphs [31], small-world graphs, scale-free graphs [19]. Graph algorithms' performance may vary a lot on different types of graphs. On the other hand, there are a variety of graph computations such as path finding, subgraph matching, community detection, and graph partitioning. Each graph computation itself even deserves dedicated research; it is nearly impossible to design a system that can support all kinds of graph computations. There are also two completely different scenarios for graph computations: offline analytics and online query answering. They have different requirements on the performance and effectiveness of the graphs algorithms or solutions. All these diversity factors interleave with each other to make the situation even worse.

2.2 Complicated Structures of Real Big Graphs

Most real graphs are notoriously known as complex networks. Complex network means that the real graphs have a structure that is far away from the regularly simple structure that can be explained by simple network growth dynamics or modeled by simple models such as regular graphs, or ER networks. Next, we will discuss some of the complex structural characteristics of real graphs including scale free (power-law degree distribution) [19], small world *small-world* and *community structure*, as well as their influence on the billion-node graph management.

- *Scale free.* Most real graphs have a power law degree distribution, which is known as the scale free property of real graphs. The power law degree distribution implies that most vertices have small degrees, while some vertices of large degree do have a great opportunity to exist. These vertices of highest degree are usually referred to as *hubs*. There are two implications of power law degree distribution on big graph management. First, a distributed solution is easily trapped in load unbalance. If the hub vertices are not carefully taken care of, the machines hosting them tend to be the bottleneck of the system since hubs have more logical connections to other vertices and have a large probability to receive or send message from or to neighbors. The second implication is that the graph is more heterogeneous in terms of degree when compared to synthetic graph models such as ER model [31], where most vertices tend to have the average degree. The heterogeneous degrees make it difficult to design a universal strategy to process vertices of different degrees.

- *Small world.* Most real life networks are small-world networks, i.e., the average shortest distance is quite small. In most social networks, any two persons can reach each other within six steps. It is also shown that real life networks *shrink* when they grow in size, that is, their diameters become smaller [20]. As a result, the graph exploration in a big graph becomes extremely challenging since exploration of six steps or less will need to traverse almost the entire graph. The traverse of a billion-node graph usually costs serval hours. As a result, any online query answering that needs to explore the graphs on a billion-node graph might incur unaffordable cost.

- *Community structure.* Most real-life complex networks, including the Internet, social networks, and biological neural networks, contain community structures. That is, the networks can be partitioned into groups within which connections are dense and between which connections are sparse. The community structure makes the divide-and-conquer possible. However, partitioning a billion-node graph itself is non-trivial. And in many cases, the communities are not necessarily of equal size, which makes the balanced load distributions harder. On the other hand, real graphs usually tend to have an unclear community structure, that is the boundaries of communities are not clear and two communities in many real cases are hard to be separated. As a result, although the existence of community structure provides an opportunity, it is still challenging to leverage the community structure to develop effective solutions.

There are also many other properties of real graphs that pose great challenges to graph management, such as dynamically evolving, containing heterogeneous information. This article is only a limited discussion of these challenges. The complicated structures not only poses great challenge for us to manage big graph but also provide us new opportunities to build effective solutions to manage billion-node graphs if we are well aware of their existence and fully leverage these properties. Some recent approaches take into consideration the power-law degree distribution and community structure exhibited by many real life networks to develop effective distributed storage solutions [3] or reduce communication costs [2] in distributed graph processing.

3 Principles

Next, we present a set of general principles to overcome the billion-node graph challenges according to our previous experience to manage billion-node graphs. All these principles reflect our choice of different options in developing solutions. Most of our principles are intuitively correct. However, how to realize them in real problems and on big graphs is non-trivial. Clearly, it is only a limited set of effective principles. More effective principles are coming when we have more practices to process billion-node graphs. In the following text, we use detailed examples to showcase each principle. All the principles are discussed with respect to billion-node graph challenges.

3.1 Principle 1: Lightweight models are preferred

Billion-node scale demands lightweight models. In many real applications, the real requirement can be satisfied by problems models of different complexity. In general, when the graph is big, the models of less complexity is preferred. However, no free lunch in general. The lightweight models usually come with the cost of the sacrifice of effectiveness, which however is usually acceptable when processing billion-node graphs. For most heavyweight problems on graphs, there are usually corresponding lightweight models. For example, to answer the shortest distance query, distance oracle that uses a precomputed data structure to estimate the shortest distance approximately is usually more lightweight compared to the exact shortest distance query. To find a community around a vertex, the community search [1] is more lightweight than community mining. To find the landmarks of a graph, approximate betweenness is more lightweight than the exact betweenness. Next, we use shortest distance query as an example problem to elaborate this principle.

Distance oracle. Shortest distance queries are one of the fundamental queries on graphs. Exact shortest distance query answering on billion node graphs either by offline pre-computation or online computation are unaffordable. For the online computation, it takes hours for the Dijkstra algorithm (on weighted graph) or bread-first search (on unweighted graphs) to find the shortest distance between two nodes [25] in a billion-node graph. Alternatively, by offline computation we can pre-compute and store the shortest distances for all pairs of nodes, and use table lookups to answer shortest distance queries at the time of the query. Clearly, this approach requires quadratic space, which is prohibitive on large graphs.

Our experience shows that rather than exact shortest distance query, *distance oracle* is a more realistic model to answer shortest distance queries on billion node graphs. A distance oracle is a pre-computed data structure that enables us to find the (approximate) shortest distance between any two vertices in constant time. A distance oracle is feasible for billion node graphs if it satisfies the following criteria:

1. Its pre-computed data structure has a small *space complexity*. For billion-node graphs, we can only afford linear, or sub-linear pre-computed data structures.

2. Its construction has a small *time complexity*. Although distance oracles are created offline, for billion-node graphs, we cannot afford algorithms of quadratic time complexity.

3. It answers shortest distance queries in *constant time*.

4. It answers shortest distance queries with high *accuracy*.

We built a distance oracle by an embedding based approach which will be elaborated in the next principle. The result shows that our distance oracle is a realistic solution for the online shortest distance query on billion-node graphs. Our distance oracle takes about 20 to 70 hours to construct distance oracle on billion node graphs. Since it is built offline, this performance is acceptable. The query response is also quite efficient. It only takes less than 1 ms to answer the shortest distance query on a billion node graph. The accuracy results show that for geodesic distances less than 14 (which account for the majority of the shortest paths), our distance oracles generate distances with absolute error consistently less than 0.3 and relative error consistently less than 8%.

3.2 Principle 2: Approximate solutions are usually acceptable

For billion-node graph, we can not accept the algorithms or solutions with super-linear complexity. In general, only linear or near-linear (such as $O(N \log N)$) is acceptable for billion-node graphs. However, the efficiency usually comes with the sacrifice of accuracy. For billion-node graphs, minor sacrifice of accuracy usually is acceptable. Next, we will show how we use embedding based solution to answer shortest distance queries.

Embedding based distance oracle Graph embedding projects the elements of graphs (such as nodes or edges) into a geometric space so that the key properties of the original graph is preserved in the geometric space. There are a lot of graph embedding solutions due to the popularity of deep learning models which usually requires graphs as input. In this paper, we only consider the embeddings that is critical for big graph management.

Specially, we will discuss how to use embedding to build an efficient-yet-effective distance oracle. By this case study, we show that embedding is a promising transformation operation that can be used as an offline step to derive the essential information of the graph.

To build a distance oracle, we embed a graph into a geometric space so that shortest distances in the embedded geometric space preserve the shortest distances in the graph. Then, the shortest distance query can be answered by the coordinate distance. Let c be the dimensionality of the embedded space. The space complexity of the coordinate based solution is $\Theta(c \cdot |V|)$. The query can be answered in $\Theta(c)$ time, independent of the graph size.

Graph embedding is the key to the success of distance oracles of this kind. State-of-the-art approaches [26, 27] first select a set of landmark vertices using certain heuristics (for example, selecting vertices of large degrees). For each landmark, BFS is performed to calculate the shortest distance between each landmark and each vertex. Then, we fix the coordinates of these landmarks using certain global optimization algorithms based on the accurate distances between landmarks. Finally, for each non-landmark vertex, we compute its coordinates according to its distances to the landmarks. Finding the coordinates of landmarks and non-landmark vertices is formulated as an optimization problem with the objective of minimizing the sum of squared errors:

$$\sqrt{\sum_{(u,v)} (|c(u) - c(v)| - d(u,v))^2} \tag{1}$$

where $c(u)$ is the coordinates of vertex u. Usually, this kind of optimization problem can be solved by the *simplex downhill* method [28]. Given the coordinates of two vertices, their shortest distance can be directly estimated by the geometric distance between them. The distance calculation can be further improved by certain low-bound or upper-bound estimation.

Experimental results on real graphs show that embedding based distance oracle can find shortest distance for an arbitrarily given pair of nodes with the absolute error almost consistently less than 0.5, which is significantly better than other competitors [2].

3.3 Principle 3: Local information is usually helpful

A graph can be logically or physically organized. In either way, locality of the graph does matters in handling billion-node graphs. The subgraphs aground a certain vertex is a logically local graph. In distributed graph computing, the subgraphs induced by the vertices and their adjacent list in a certain machine is a physically local graph. Fully leveraging the locality of these graphs alleviates us from the complexity of the entire graph. By focusing on the local graphs, we can build effective and scalable solutions. To illustrate this principle, we will showcase how to use local graphs in local machines to estimate the betweenness of vertex.

Betweenness computation by Local graph based vertex Betweenness is one of the important measures of vertex in a graph. It is widely used as the criteria to find important nodes in a graph. Betweenness has been empirically shown to be the best measure to select landmarks for shortest distance query or shortest path query [29]. For a vertex v, its betweenness is the fraction of all shortest paths in the graph that pass through v. Formally, we have:

$$bc(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2}$$

where σ_{st} is the number of shortest paths between s and t, and $\sigma_{st}(v)$ is the number of shortest paths in σ_{st} that pass through v.

It is not difficult to see that the exact calculation of betweenness requires to enumerate all pairs of shortest path. The fastest algorithm needs $O(|V||E|)$ time to compute exact betweenness [30]. It is prohibitive for a billion-node graph. Thus an *approximate betweenness* is a more appropriate choice than the exact betweenness on big graphs. However, it is non-trivial to develop an effective-yet-efficient approximate betweenness estimation solution in a distributed environment.

We propose an efficient and effective solution to compute approximate betweenness for large graphs distributed across many machines. A graph is usually distributed over a set of machines by hashing on the vertex id. The hash function distribute a vertex as well as its adjacent list to a certain machine, which allows to build a local graph consisting of vertices and their relationship that can be found from the same machine. We show a local graph in Example 1. Instead of finding the shortest paths in the entire graph, we find the shortest paths in each machine. Then, we use the shortest paths in each machine to compute the betweenness of vertices on that machine. We call betweenness computed this way *local* betweenness. Clearly, the computation does not incur any network communication. After we find local betweenness for all vertices, we use a single round of communication to find the top-k vertices that have the highest local betweenness value, and we use these vertices as landmarks.

Theoretical results show that the shortest distance estimated from local graphs has a upper bound on the error of the exact shortest distance. This allows us to use the local shortest distance to approximate exact betweenness. Experimental results on real graphs shows that contrary to the perception that local betweenness is very inaccurate because each machine only contains a small portion of the entire graph, it turns out to be a surprisingly good alternative for the exact betweenness measure. Please refer to for the detailed results.

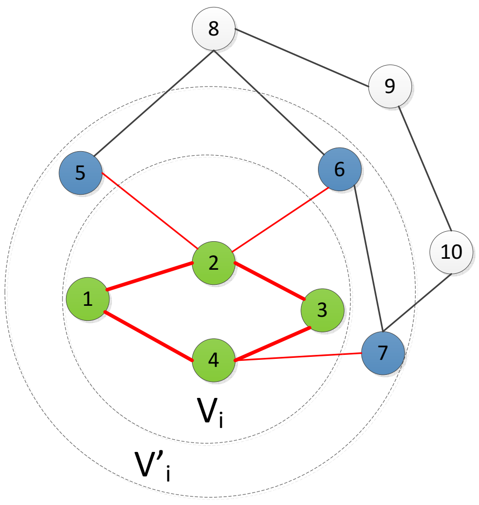


Figure 1: The local graph, the extended local graph, and the entire graph, from the perspective of machine i.

Example 1 (Local graphs): Consider a graph with 10 vertices, as shown in Figure 1. Assume machine i contains 4 vertices, that is, $V_i = \{1, 2, 3, 4\}$. The graph in the innermost circle is the *local graph*. Machine i also has information about vertices $5, 6$, and 7, as they are in the adjacency lists of vertex 2 and 4. The graph inside the second circle, except for the edge $(6, 7)$, is the *extended local graph*. Machine i does not realize the existence of vertices $8, 9$, and 10. Note that edge $(6, 7)$ does not belong to the extended local graph since none of its ends belongs to V_i.

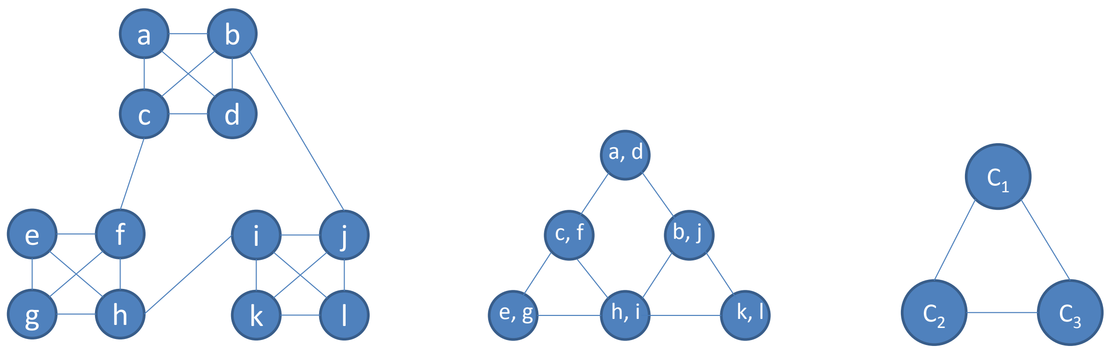
3.4 Principle 4: Fully leverage the properties of real graphs

We have shown that real big graphs usually exhibit a lot of structural characteristics. These characteristics on the one hand pose great challenge to build scalable solutions on graphs. On the other hand, if we are fully aware of these characteristics, we have a great opportunity to boost the performance of the solutions. Next, we show two cases in which we build effective solutions by carefully leveraging the properties of real graphs. The first is partitioning a billion graph by employing the community structure of real big graphs. The second is reducing the communication cost for a distributed bread-first-search by employing the power-law degree distribution of real graphs.

3.4.1 Graph partitioning based on community structure

As we have claimed that a distributed system is necessary to manage a billion-node graph. To deploy a graph on a distributed system, we need to first divide the graph into multiple partitions, and store each partition in one machine. How the graph is partitioned may cause significant impact on load balancing and communication. Graph partitioning problem thus is proposed to distribute a big graph onto different machines. Graph partitioning problem seeks solution to divide a graph into k parts with approximately identical size so that the edge cut size or the total communication volume is minimized in a distributed system. The problem of finding an optimal partition is NP-Complete [21]. As a result, many approximate solutions have been proposed [9, 10], such as KL [9] and FM [10]. However, these algorithms in general are only effective for small graphs. For a large graph, a widely adopted approach is to "coarsen" the graph until its size is small enough for KL or FM. The idea is known as multi-level graph partitioning, and a representative approach is METIS [11].

METIS uses maximal match to coarsen a graph. A maximal match is a maximal set of edges where no two edges share a common vertex. After a maximal match is found, METIS collapses the two ends of each edge into one node, and as a result, the graph is "coarsened.". Maximal match based coarsening however is unaware of the community structure of real graphs. Thus it is quite possible to break the community structure in the coarsened graphs. As a result, a maximal match may fail to serve as a good coarsening scheme in graph partitioning. For example, the coarsened graph shown in Figure 2(b) no longer contains the clear structure of the original graph shown in Figure 2(a). Thus, partitions on the coarsened graph cannot be optimal for the original graph. METIS use subtle refinement to compensate for the information loss due to coarsening, leading to extra cost.



(a) A graph (b) Coarsened by maximal match (c) Coarsened by LP

Figure 2: An example graph and its coarse-grained graph

A community-structure aware coarsening is proposed in the billion-graph partitioning solution MLP (multiple-level propagation). MLP uses *label propagation* (LP) [23, 24] to find the community of a graph and then coarsen a big graph. Compared to maximal match, LP is more semantic-aware and is able to find the inherent community structure of the graph, as illustrated in Figure 2(c). Thus, the coarsened graph preserves the community structure of the original graph. Consequently, the locally closed nodes tend to be partitioned into the same machine. MLP thus has no necessary for extra refinement, which saves time cost and makes MLP a good choice to partition a billion-node graph.

The experimental results on the synthetic graphs with embedded communities show that MLP can effectively leverage the community structure of graphs to generate a good partitioning with less memory and time [3]. In contrast, METIS, which is based on the maximal matching method, is not community-aware when it coarsens a graph, thus heavily relying on costly refinement in the uncoarsening phase to ensure the solution quality. As a result, METIS incurs more time and space costs.

3.4.2 Network IO reduction based on the power-law degree distribution

Bread-first search is one of the most important operations on graphs. Billion-node graph management calls for distribute BFS algorithms. *Level-synchronized BFS* is one of typical distributed algorithm to compute shortest distances in a distributed environment. Level-synchronized BFS proceeds level by level starting from a given vertex. At level l, each machine i finds vertices of distance l to r. Then, we locate their neighboring vertices. If a neighboring vertex has not been visited before, then it has distance $l + 1$ to r. However, since the neighboring vertices are distributed across all machines, only their host machines (the machines they reside on) know their current distances to r. Thus, we ask their host machines to update the distance of these neighboring vertices. Each remote machine i, upon receiving messages from all of other machines, updates their distances to either $l + 1$ if they have not been visited before, or keeps their current distances (equal to or less than l) unchanged. After each level, all the machines synchronize, to ensure that vertices are visited level by level.

In the above naive BFS algorithm, the cost is dominated by sending messages to remote machines to update vertices' distances to landmarks. We reduce the communication cost by caching a subset of vertices on each machine. The opportunity comes when we notice that many distance update requests in level-synchronized BFS actually are wasteful. Specifically, a distance update request for vertex u is wasteful if the shortest distance of u to the starting node is already known. Avoiding such wasteful requests can reduce the communication cost. A straightforward way to do this is to cache each vertex's current distance on each machine. Then, we only need to send messages for those vertices whose current distances to r are ∞. In this manner, we can reduce the communication cost. However, for billion node graphs, it is impossible to cache every vertex on every machine. Then, the problem is: *which vertices to cache?*

Intuitively, the number of remote messages we saved by caching vertex v is closely related to the degree of v. The larger the degree of v, the more messages can be saved. Since real networks are usually scale free, that is, only a small fraction of vertices have large degree. This allows us to save a significant number of network messages by caching the top-K vertices of highest degree. We use an empirical study [2] to obtain a more intuitive understanding about the effectiveness to save network IOs by caching the hub vertices. The study uses the scale-free graphs [32] whose degree follows the distribution as follows:

$$deg(v) \propto r(v)^R \tag{3}$$

where $r(v)$ is the degree rank of vertex v, i.e., v is the $r(v)$-th highest-degree vertex in the graph, and $R < 0$ is the rank exponent.

The simulation results are given in Figure 3 with R varying from -0.6 to -0.9 (many real networks' rank exponent is in this range) [2]. From the simulation, we can see that by caching a small number of hub vertices, a significant number of communications can be saved. For example, when $R = -0.9$, caching 20% of the top-degree vertices can reduce communications by 80%.

4 Conclusion

In this paper, we elaborate the challenges to mange big graphs of billion nodes in the distributed memory system. We propose a set of general principles to develop efficient algorithmic solutions to manage billion-node graphs based on our previous experience to process billion-node graphs. Although these principles are effective, we

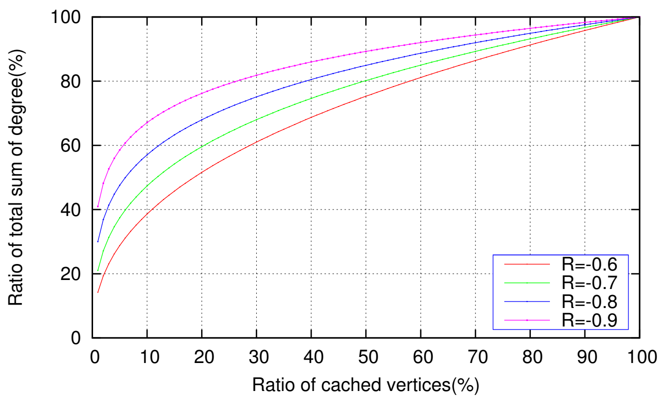


Figure 3: Benefits of caching hub vertices on scale free graphs

still have many open problems in billion-node graph management. The most challenging open problem might be the design of a universal solution to manage big graphs. Currently, we can only develop efficient solutions that are well tuned for specific tasks or specific graphs. In general, we still have a long way to overcome the billion-node graph challenges.

Acknowledgement This paper was partially supported by National Key Basic Research Program of China under No.2015CB358800, by the National Natural Science Foundation with No.61472085, U1509213, by Shanghai Municipal Science and Technology Commission foundation key project under No.15JC1400900, by Shanghai Municipal Science and Technology project under No.16511102102, No.16JC1420401.

References

[1] W. Cui, Y. Xiao, H. Wang, Y. Lu, and W. Wang, "Online search of overlapping communities," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 277–288.

[2] Z. Qi, Y. Xiao, B. Shao, and H. Wang, "Toward a distance oracle for billion-node graphs," *Proc. VLDB Endow.*, vol. 7, no. 1, pp. 61–72, Sep. 2013.

[3] L. Wang, Y. Xiao, B. Shao, and H. Wang, "How to partition a billion-node graph," in *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, Eds. IEEE, 2014, pp. 568–579.

[4] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 505–516.

[5] http://www.worldwidewebsize.com/.

[6] http://www.facebook.com/press/info.php?statistics.

[7] http://www.w3.org/.

[8] D. R. Zerbino et al., "Velvet: algorithms for de novo short read assembly using de bruijn graphs." *Genome Research*, vol. 18, no. 5, pp. 821–9, 2008.

[9] B. Kernighan et al., “An efficient heuristic procedure for partitioning graphs,” *Bell Systems Journal*, vol. 49, pp. 291–307, 1972.

[10] C. M. Fiduccia et al., “A linear-time heuristic for improving network partitions,” in *DAC ’82*.

[11] G. Karypis et al., “Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0,” Tech. Rep., 1995.

[12] G. Karypis et al., “Analysis of multilevel graph partitioning,” in *Supercomputing ’95*.

[13] G. Karypis et al., “A parallel algorithm for multilevel graph partitioning and sparse matrix ordering,” *J. Parallel Distrib. Comput.*, vol. 48, pp. 71–95, 1998.

[14] A. Lumsdaine et al., “Challenges in parallel graph processing,” *Parallel Processing Letters*, pp. 5–20, 2007.

[15] G. Malewicz et al., “Pregel: a system for large-scale graph processing,” in *SIGMOD ’10*.

[16] A. Abou-Rjeili et al., “Multilevel algorithms for partitioning power-law graphs,” in *IPDPS ’06*.

[17] B.Shao et al., ”The trinity graph engine,” *Microsoft Technique Report, MSR-TR-2012-30*.

[18] P. Erdős et al., “Graphs with prescribed degrees of vertices (hungarian).” *Mat. Lapok*, vol. 11, pp. 264–274, 1960.

[19] A.-L. Barabasi et al., “Emergence of scaling in random networks,” vol. 286, pp. 509–512, 1999.

[20] J. Leskovec et al., “Graphs over time: Densification laws, shrinking diameters and possible explanations,” in *KDD ’05*.

[21] M. R. Garey et al., “Some simplified np-complete problems,” in *STOC ’74*.

[22] K. Munagala et al., “I/O-complexity of graph algorithms,” in *SODA ’99*.

[23] M. J. Barber et al., “Detecting network communities by propagating labels under constraints,” *Phys.Rev.E*, vol. 80, p. 026129, 2009.

[24] X. Liu et al., “How does label propagation algorithm work in bipartite networks?” in *WI-IAT ’09*.

[25] A. Das Sarma, S. Gollapudi, M. Najork, and R. Panigrahy, “A sketch-based distance oracle for web-scale graphs,” in *WSDM ’10*.

[26] X. Zhao, A. Sala, H. Zheng, and B. Y. Zhao, “Fast and scalable analysis of massive social graphs,” *CoRR*, 2011.

[27] X. Zhao, A. Sala, C. Wilson, H. Zheng, and B. Y. Zhao, “Orion: shortest path estimation for large social graphs,” in *WOSN’10*.

[28] J. A. Nelder and R. Mead, “A simplex method for function minimization,” *Computer Journal*, 1965.

[29] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis, “Fast shortest path distance estimation in large networks,” in *CIKM ’09*.

[30] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of Mathematical Sociology*, 2001.

[31] aErdős, P. and Gallai, T., “Graphs with prescribed degrees of vertices (Hungarian).,” *Mat. Lapok*,1960.

[32] M. Faloutsos, P. Faloutsos, and C. Faloutsos, “On power-law relationships of the internet topology,” *SIGCOMM Comput. Commun. Rev.*, 1999.