Max-flow Min-cut Algorithm in Spark with Application to Road Networks

Varun Ramesh[1(✉)], Shivanee Nagarajan[1], and Saswati Mukherjee[2]

[1] Department of Computer Science and Engineering, Anna University, Chennai, India
varun.ceg.95@gmail.com, shivanee.ceg@gmail.com
[2] Department of Information Science and Technology, Anna University, Chennai, India
msaswati@auist.net

Abstract. The max-flow min-cut problem is one of the most explored and studied problems in the area of combinatorial algorithms and optimization. In this paper, we solve the max-flow min-cut problem on large random graphs with log-normal distribution of outdegrees using the distributed Edmonds-Karp algorithm. The algorithm is implemented on a cluster using Spark. We compare the runtime between a single machine implementation and cluster implementation and analyze the impact of communication cost on runtime. In our experiments, we observe that the practical value recorded across various graphs is much lesser than the theoretical estimations primarily due to smaller diameter of the graph. Additionally, we extend this model theoretically on a large urban road network to evaluate the minimum number of sensors required for surveillance of the entire network. To validate the feasibility of this theoretical extension, we tested the model with a large log-normal graph with ∼1.1 million edges and obtained a max-flow value of 54, which implies that the minimum-cut set of the graph consists of 54 edges. This is a reasonable set of edges to place the sensors compared to the total number of edges. We believe that our approach can enhance the safety of road networks throughout the world.

1 Introduction

Max-flow min-cut problem has a wide variety of applications in various domains including spam site discovery [28], community identification [11], and network optimization [1]. For instance, this problem has been previously applied on road networks to assess the reliability of the network [27]. Road networks such as that of California are extremely large and it has 1,965,206 nodes and 2,766,607 edges [24]. Protection and monitoring of such road networks for safety is a challenging task. Furthermore, to analyze such large data, it is impractical to use an extremely expensive machine equipped with voluminous storage and processing capabilities. We posit that distributed algorithm on a cluster of commodity machines, as attempted earlier [20], is an ideal solution for such large computations. Halim *et al.* in [20] used Hadoop to implement the max-flow algorithm. In similar vein, in this paper, we have chosen Spark over Hadoop to implement a distributed max-flow min-cut algorithm on a cluster.

V. Ramesh and S. Nagarajan contributed equally.

J.J. Jung and P. Kim (Eds.): BDTA 2016, LNICST 194, pp. 12–22, 2017.
DOI: 10.1007/978-3-319-58967-1_2

Classical max-flow algorithms [14] are not feasible for real world graphs as they require the entire graph to be in memory for computation. However, max-flow algorithms such as Edmonds-Karp algorithm [10] and push-relabel algorithm [16] have good distributed settings [7, 13]. A detailed explanation as to why we have chosen the Edmonds-Karp algorithm is presented in Sect. 4. We adopt the distributed algorithm from [7] and implement it on a single machine initially. Next, we implement the same on a cluster of three machines and analyze the runtime. The analysis of communication costs when iterative graph algorithms are implemented on a cluster is essential. This is because the graph will be partitioned across cluster nodes and there will be considerable communication costs amongst them to exchange information about neighboring vertices. Thus, in addition to runtime of such an algorithm, communication costs play a vital role in the overall efficiency. To the best of our knowledge, this is the first implementation of its kind on Spark where the runtime and communication costs are experimentally compared between a cluster and a single machine, based upon the max-flow min-cut algorithm.

For n vertices and m edges, the complexity as achieved in [7] is $O(cm/k) + O(cnlogk)$ with k machines and c being the max-flow value. The communication cost expected theoretically is $O(cm) + O(cnk)$. But we have noted experimentally on our cluster that the communication costs are much lesser due to smaller diameter of the graph [23] and the runtime in practice is much more efficient.

We propose a model based on our max-flow min-cut implementation to evaluate the minimum number of sensors required for surveillance of an urban road network, which is modelled as a flow graph. In a flow network, the max-flow value obtained is the minimum cut-set—the smallest total weight of edges which if removed would disconnect the source from the sink. The ratio of the cut-set to the total number of edges in the graph is then investigated to assess the feasibility of our approach. This model provides the advantage of scalability to cover the range of multiple cities, where the edge set may have over 5 million edges. An appropriate cluster size can handle such large data set and we believe that our proposal can enhance the safety of road networks throughout the world.

The rest of the paper is organized as follows. Section 2 provides background and related work. Analysis of the Edmonds-Karp algorithm and issues in the distributed settings of other algorithms are explained in Sect. 3. Various phases of the distributed algorithm are explored in Sect. 4. Our experiments on a single machine and cluster result are compared in Sect. 5, along with the investigation of practical communication costs in the cluster. Theoretical extension of the max-flow min-cut implementation to evaluate the minimum number of sensors required to provide surveillance in a large city modelled as a flow graph is performed in Sect. 6. We summarize in Sect. 7.

2 Background and Related Works

Over the years, many efficient solutions to the maximum flow problem were proposed. Some of them are the shortest augmenting path algorithm of Edmonds and Karp [10], the blocking flow algorithm of Dinic [9], the push-relabel algorithm of Goldberg and Tarjan [16] and the binary blocking flow algorithm of Goldberg and Rao [15]. The first

method developed is the Ford-Fulkerson method [12]. The algorithm works by finding an augmenting path as long as one exists and then sending the minimal capacity possible through this path.

MapReduce (MR) [8] is a programming model and an implementation for processing large datasets. MapReduce programs read input data from disk, map a function across the data, reduce the results of the map, and store reduction results back in the disk. Apache Spark [29] provides programmers with an API centered on a data structure called the resilient distributed dataset (RDD), a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant manner [30]. The availability of RDDs helps in implementing iterative algorithms as needed in our case where the dataset is visited multiple times in a loop. MapReduce has a linear dataflow pattern and Spark was developed to improve this dataflow pattern. Spark is more suitable than Apache Hadoop [19] for iterative operations because of the cost paid by Hadoop for data reloading from disk at each iteration [18].

Spark provides an API for graph algorithms that can model the Pregel [25] abstraction. Pregel was specially developed by Google to support graph algorithms on large datasets. Having mentioned about the Pregel API, it is essential that we briefly give an overview of Pregel's vertex centric approach below, and describe how it handles iterative development using supersteps.

2.1 Overview of Pregel Programming Model

Pregel computations consist of a sequence of iterations, called supersteps. During a superstep the framework invokes a user-defined function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex V and a single superstep S. It can read messages sent to V in superstep $S-1$, send messages to other vertices that will be received at superstep $S+1$, and modify the state of V and its outgoing edges [25]. Edges are not the main focus of this model and they have no primary computation. A vertex starts in the active state and it can deactivate itself by voting to halt. It will be reactivated when it receives another message and the vertex has primary responsibility of deactivating itself again. Once all vertices deactivate and there are no more messages to be received by any of the vertices, the program will halt.

Message passing is the mode of communication used in Pregel as it is more expressive and fault tolerant than remote read. In a cluster environment, reading a value from a remote machine can incur a heavy delay. Pregel's message passing model reduces latency by delivering messages in batches asynchronously. Fault tolerance is achieved through checkpointing and a simple heartbeat mechanism is used to detect failures. MR is sometimes used to mine large graphs [21], but this can lead to suboptimal performance and usability issues [25]. Pregel has a natural graph API and is much more efficient support for iterative computations over the graph [25].

2.2 Other Related Works

General purpose distributed dataflow frameworks such as MapReduce are well developed for analyzing large unstructured data directly but direct implementations of iterative graph algorithms using complex joins is a challenging task. GraphX [17] is an

efficient graph processing framework embedded within the Spark distributed dataflow system. It solves the above mentioned implementation issues by enabling composition of graphs with tabular and unstructured data. Additionally, GraphX allows users to choose either the graph or collective computational model based on the current task with no loss of efficiency [17].

Protection of Urban road networks using sensors is an important but demanding task. Applying the max-flow min-cut algorithm under the circular disk failure model to analyze the reliability of New York's road network has been performed in [27]. Placing sensors across the road network using the min-cut of the graph is explored in [3] using GNET solver. In contrast, we use our spark based max-flow min-cut implementation to place sensors efficiently across the network and discuss about practical issues such as scalability.

3 Maximum Flow Algorithm

Before we move to the distributed model, we need to understand the implications of the Edmonds-karp algorithm on a single machine setting. Additionally, we discuss issues in achieving parallelism in the push-relabel algorithm.

3.1 Overview of Max-flow Problem

A flow network [1] $G = (V,E)$ is a directed graph where each edge $(u,v) \in E$ has a non-negative capacity $c(u,v) \geq 1$. There are two end vertices in a flow network: the source vertex S and the sink vertex T. A flow is a function $f : V \times V \rightarrow R$ satisfying the following three constraints for all *u, v*:

1. The flow along an edge $f(u,v) \leq c(u,v)$ otherwise the edge will overflow.
2. The flow along one direction say $f(u,v) = -f(v,u)$ which is the opposite flow direction.
3. Flow must be conserved across all vertices other than the source and sink, $\sum f(u,v) = 0$ for all $(u,v) \in V - \{S,T\}$.

Augmenting path and residual graph are two very important parameters of the max-flow problem. An augmenting path is a simple path — a path that does not contain cycles. Given a flow network $G(V,E)$, and a flow $f(u,v)$ on G, we define the residual graph R with respect to $f(u,v)$ as follows.

1. The node set v of R and G are same.
2. Each edge $e = (u,v)$ of R is with a capacity of $c(u,v) - f(u,v)$.
3. Each edge $e' = (v,u)$ is with a capacity $f(u,v)$.

The push-relabel algorithm [16] maintains a preflow and converts it into maximum flow by moving flow locally between neighboring nodes using push operations. This is done under the guidance of an admissible network maintained by relabel operations. A variant of this algorithm is using the highest label node selection rule in $O(v^2\sqrt{e})$ [4]. A parallel implementation of the push-relabel method, including some speed-up heuristics applicable in the sequential context has been studied in [13].

The push-relabel algorithm has a good distributed setting but has the main problem of low-parallelism achieved at times [22]. This means that among hundreds of machines which are computing the flow on Spark only a few of them may be working actively and further its high dependency on the heuristic function is not always suitable [5]. Moreover, other than the degenerate cases, max-flow algorithms, which make use of dynamic tree data structures, are not efficient in practice due to overhead exceeding the gains achieved [2]. On the contrary, the Edmonds-Karp algorithm works well without any heuristics and also has a high-degree of parallelism in this low diameter large graph settings. This is because of many short paths existing from the source to sink. One more primary advantage of using log-normal graphs is that they have a diameter close to *logn* [23]. In the Edmonds-Karp algorithm mentioned below (see Algorithm 1), input is Graph $G = (V, E)$ with n vertices and m edges. $\{S, T\} \in V$ are the source and sink respectively. L is the augmenting path stored as a list. $w_1, w_2 ... w_n$ are the weights of the edges in the path. F_{max} is the flow variable which is incremented with the Min-flow value in every iteration.

Algorithm 1. Edmonds-karp algorithm for a single machine

```
1: procedure MAX-FLOW(S, T)
2:     while augmentingpath do
3:         L ← augmentingpath(S, T)
4:         Minflow ← min(w1, w2, ...wn)  in  L
5:         Update Residual Graph and overall Fmax
6:     return Fmax
```

We observe that for a graph G with n vertices and m edges the algorithm has a runtime complexity of $O(nm^2)$. On a single machine, if we use the bidirectional search to find the augmenting path then the time taken will be reasonable. But, if the maximum flow value is large then this will affect the runtime negatively and thus using the Pregel Shortest path approach is useful for this procedure.

4 Distributed Max-flow Model

In the distributed model, the first step is to calculate the shortest augmenting path from source to sink. The shortest augmenting path can either follow Δ-stepping method [26] or the Pregel's procedure for calculating shortest path. The Δ-stepping method has been implemented before in [6]. Given that the Pregel shortest path has an acceptable runtime, we implement the latter in Spark. After this step, we find the minimum feasible flow and broadcast this across the path which is stored as a list. The residual graph is then updated and the overall flow is incremented according to the flow value obtained.

Shortest Path using Pregel. Google's Pregel paper [25] provides with a simple algorithm to implement the single source shortest path method. In case of the max-flow problem, the S-T shortest path has to be found which is a slight variation of the single source shortest path problem.

Each vertex has three attributes - d the distance from the source S, c being the minimum capacity seen till now by that vertex and id is the identifier of the node from which the previous message had arrived. The shortest path P is obtained as output from Algorithm 2 and is stored as a list.

Algorithm 2. Shortest path using Pregel [7]

1: Make all distance value except source as ∞ and source to 0.
2: **while** $d_t = \infty$ **do**
3: **Message sent by node** i **with attributes** $(d_i + 1, c_i, i)$ to each neighbor j only if $c_{ij} > 0$.
4: **Function applied to node** j **upon receiving message** $(d_i + 1, c_i, i)$:
5: – set $d_j := min(d_j, d_i + 1)$
6: – if $(d_i + 1 < d_j)$
7: * set $id_j := i$
8: * $c_j := min(c_i, c_{ij})$ where c_{ij} is the capacity along edge (i, j)
9: **Merging functions upon receiving** $m_i = (d_i + 1, c_i, i)$ *and* $m_j = (d_j + 1, c_j, j)$:
10: – if $d_i < d_j : m_i$
11: – else if $(d_i = d_j$ *and* $c_j < c_i) : m_i$
12: – else m_j

Flow Updation. The minimum capacity of the all the edges in the shortest path is the maximum flow that can be pushed through the path. The minimum value c_{min} is determined and the flow value from source to sink is incremented with c_{min}. The shortest path is also broadcasted.

Residual Graph Updation. This step includes MapReduce operation on the graph obtained from the last iteration. The key-value pair for this step is of the form $((i, j) : capacity)$.

Algorithm 3. Building Residual Graph R_G [7]

1: **Map** (input: edge,output: edge)
2: if P contains edge $(i; j) in R_G$:
3: – emit $((i, j) : c - f_{max})$
4: – emit $((j, i) : f_{max})$
5: else: emit $((i, j) : c)$ (no changes)
6: **Reduce:** sum

Runtime Analysis. For a graph with n vertices and m edges, cluster size k, the runtime complexity is mentioned as follows. For the $s - t$ shortest path algorithm, the current iteration's distance value for a node will be less than what it receives in the next iteration. Thus, only once a node emits messages outwards. For each edge we associate it with a message and thus the worst-case complexity is $O(m)$. If k machines are used in the cluster, then the complexity becomes $O(m/k)$. The step to initialize each vertex to infinity takes $O(n)$. The broadcast of the minimum flow value takes $O(nlogk)$. The next

step of flow updation and residual updation in the worst case would take $O(m)$, because each edge will be passed in the iteration. Similarly, if k machines are used the worst case evaluates to $O(m/k)$. The number of iterations is bound by the maximum-flow value c. Therefore the overall run time as: $O(cm/k) + O(cnlogk)$.

Communication Cost. The communication cost of the shortest path $s-t$ method is $O(m)$. Broadcast is done to a list of at most n nodes over k machines. This leads to $O(nk)$. Similarly, the number of iterations are bound by the max-flow value c. The overall communication cost is $O(cm)+O(cnk)$. Both runtime as well as communication cost include $O(m)$ as a major factor and this will dominate the runtime of the algorithm in most cases. But this runtime is pessimistic and would rarely occur.

5 Experimental Analysis

The experimental setup is as follows. We used Spark 1.6.1 in a cluster of three machines. Each of the machine had 8 GB RAM, quad-core, intel i5 processor and ran Ubuntu 14.04 LTS. The large random graphs generated have a log-normal distribution of the outdegrees as done in [25]. The mean outdegree is 127.1, while some outliers have a degree larger than hundred thousand. Such a distribution enables the randomly generated graph to resemble real-world large-scale graphs [25].

Spark UI is used to view the timeline of the spark events, DAG visualization and real time statistics. The timeline view provides details across all jobs, within a single job and within a single stage. As these are random graphs, the results had a major dependence on the structure of the graph. Averages of results were taken for each of the test case. Runtime of the algorithm will have a high dependence on the diameter of the graph as the shortest path computation which returns a path from source to sink will scale proportionally to the diameter. A lower diameter for real-world graphs [23] will imply lower practical runtime for the max-flow min-cut algorithm implemented. In Fig. 1, single machine line plot depicts that there is an almost linear increase in runtime as the number of edges increase. Minor irregularities are observed in the single machine plot for instance between 0.18 million edges and 0.22 million edges. This is due to the decrease in the diameter of the graph which results in reduction of the runtime.

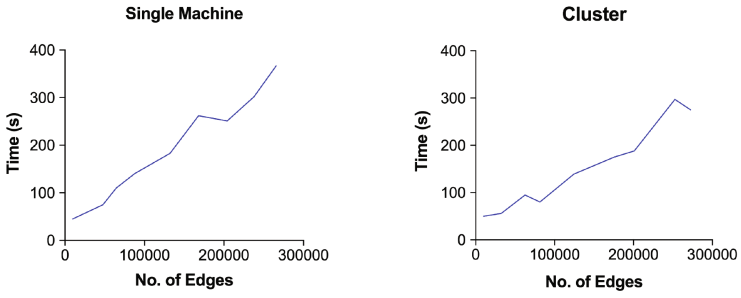

Fig. 1. Runtime analysis between single machine and cluster deployment

The cluster plot provides great insight into communication cost. It can be observed by comparing the two plots that for a similar number of edges, the cluster in most cases will provide a better runtime than a single machine as expected. But in few cases, in the initial stages, when the number of edges are about 0.05 million edges in the graph, the single machine implementation has a better runtime than the cluster. This is because communication costs prevail only in the cluster setup increasing the runtime and not in case of the single machine setup.

Further, to prove that the communication cost observed practically is much lesser than the theoretically estimated value we can look at a specific instance. For a graph with around 1.3 million edges we observed that the average runtime is about 520 s and the max-flow value is 66. Theoretically, the runtime is dominated by $O(cm)$ and for this instance it grows to $\sim 66 \times 1.3 \times 10^6$ s. This worst-case evaluation is much higher than the experimentally observed value of 520 s giving a clear indication that the worst-case bounds are very pessimistic. This creates the need for a tight upper bound which incorporates graph's diameter and topological structure.

6 Application of Distributed Max-flow Min-cut Implementation

Leading countries use public video surveillance as a major source to monitor population movements and to prevent terrorist activities. The use of Closed-circuit television (CCTV) cameras and sensors in the detection of illegal activities, data collection, recording of pre and post incident image sequences along road networks will contribute to the protection of urban population centers. Hence, it is essential for sensors to be placed in this network to detect unlawful activities. Reliability of road networks in New York city using the max-flow min-cut problem has been attempted recently [27]. Our implementation can be extended to such real world scenarios as mentioned above.

Two critical parameters in designing a system for detecting such unforeseen activities are determining how many sensors would be needed and where they should be located. We address these issues by modelling the road network as a flow graph and then the minimum cut set of the flow graph gives the optimal number of cameras or sensors to be placed.

The road network of a metropolitan city such as New York is enormous. Therefore, we process such graphs using our distributed max-flow min-cut implementation and sensors or cameras placed in the cut-set will monitor the entire network. To validate the feasibility of our approach, we evaluate the size of the min-cut and the *min-cut/edges* ratio. Smaller min-cut set implies that our approach can be practically implemented and is cost-efficient. Similar theory can be extended to a communication network to protect from a spreading virus by disconnecting the graph using the min-cut obtained for the graph.

To model the road network as a flow graph, an artificial super source and super sink node is added to the graph. Road segments are the edges and junction points between them are vertices of the graph. All edge capacities in the graph are set to unity. To facilitate the analysis, we generated a number of large graphs with log-normal distribution of outdegrees which resembles large-scale real-world road networks. An example from this set is a graph which has 11,48,734 edges and 10,000 vertices. On applying the

distributed max-flow min-cut algorithm, the min-cut set obtained had 54 edges which is 0.000047 times the number of edges in the graph. This graph was solved in 462 s. Based on our results, we hypothesize that our implementation is scalable and practically applicable for large urban road networks.

To summarize, our approach evaluates the exact number of sensors required to monitor the road network and is cost-efficient when compared to placing sensors randomly across the city. The ratio of the cut-set to the edges indicates that for large real-world graphs the cut-set obtained will be a reasonable number compared to the edges and hence, this is a practically feasible solution. The largest graph tested on our cluster had more than 1.5 million edges. Larger graphs spanning over multiple cities could have in excess of 5 million edges if modelled as a flow graph. Therefore, we believe by choosing an appropriate number of machines in the cluster such large graphs can be easily processed in reasonable time.

7 Conclusion

In this paper, we implemented the distributed version of the Edmonds-Karp algorithm using Spark. In prior efforts only single machine analysis have been performed experimentally but do not include communication cost observations over a cluster for the max-flow algorithm on Spark. We believe this is the first attempt where the runtime and the communication cost based upon the maximum flow algorithm, have been compared across a single machine and a cluster experimentally. Communication cost is observed to have a considerable impact on the runtime. We then propose a method of evaluation and placement of sensors in a large city with millions of road segments, which is modelled as a flow graph. The cut-set of a log-normal graph with 11,48,734 edges is 0.000047 times smaller than the total number of road segments. Thus, our experiments show a promising and scalable approach to place sensors in a large urban road networks based on the distributed max-flow min-cut implementation.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network flows: theory, algorithms, and applications (1993)
2. Badics, T., Boros, E.: Implementing a maximum flow algorithm: experiments with dynamic trees. Netw. Flows Matching First DIMACS Implement. Chall. **12**, 43 (1993)
3. Barnett, R.L., Sean Bovey, D., Atwell, R.J., Anderson, L.B.: Application of the maximum flow problem to sensor placement on urban road networks for homeland security. Homel. Secur. Aff. **3**(3), 1–15 (2007)
4. Cheriyan, J., Maheshwari, S.N.: Analysis of preflow push algorithms for maximum network flow. SIAM J. Comput. **18**(6), 1057–1086 (1989)
5. Cherkassky, B.V., Goldberg, A.V.: On implementing the push-relabel method for the maximum flow problem. Algorithmica **19**(4), 390–410 (1997)
6. Crobak, J.R., Berry, J.W., Madduri, K., Bader, D.A.: Advanced shortest paths algorithms on a massively-multithreaded architecture. In: 2007 IEEE International Parallel and Distributed Processing Symposium, pp. 1–8. IEEE (2007)
7. Dancoisne, B., Dupont, E., Zhang, W.: Distributed max-flow in spark (2015)

8. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
9. Dinic, E.A.: Algorithm for solution of a problem of maximum flow in a network with power estimation. Sov. Math. Dokl. **11**(5), 1277–1280 (1970)
10. Edmonds, J., Karp, R.M.: Theoretical improvements in algorithmic efficiency for network flow problems. J. ACM (JACM) **19**(2), 248–264 (1972)
11. Flake, G.W., Lawrence, S., Giles, C.L.: Efficient identification of web communities. In: Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 150–160. ACM (2000)
12. Ford, L.R., Fulkerson, D.R.: Maximal flow through a network. Can. J. Math. **8**(3), 399–404 (1956)
13. Goldberg, A.V.: Efficient graph algorithms for sequential and parallel computers. Ph.D. thesis, Massachusetts Instutute of Technology, Department of Electrical Engineering and Computer Science (1987)
14. Goldberg, A.V.: Recent developments in maximum flow algorithms. In: Arnborg, S., Ivansson, L. (eds.) SWAT 1998. LNCS, vol. 1432, pp. 1–10. Springer, Heidelberg (1998). doi:10.1007/BFb0054350
15. Goldberg, A.V., Rao, S.: Beyond the flow decomposition barrier. J. ACM (JACM) **45**(5), 783–797 (1998)
16. Goldberg, A.V., Tarjan, R.E.: A new approach to the maximum-flow problem. J. ACM (JACM) **35**(4), 921–940 (1988)
17. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2014), pp. 599–613 (2014)
18. Lei, G., Li, H.: Memory or time: performance evaluation for iterative operation on Hadoop and Spark. In: 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), pp. 721–727. IEEE (2013)
19. Apache Hadoop: Hadoop (2009)
20. Halim, F., Yap, R.H., Yongzheng, W.: A MapReduce-based maximum-flow algorithm for large small-world network graphs. In: 2011 31st International Conference on Distributed Computing Systems (ICDCS), pp. 192–202. IEEE (2011)
21. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: a peta-scale graph mining system implementation and observations. In: 2009 Ninth IEEE International Conference on Data Mining, pp. 229–238. IEEE (2009)
22. Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., Casçaval, C.: How much parallelism is there in irregular applications? In: ACM Sigplan Notices, vol. 44, pp. 3–14. ACM (2009)
23. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, pp. 177–187. ACM (2005)
24. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection, June 2014. http://snap.stanford.edu/data
25. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135–146. ACM (2010)
26. Meyer, U., Sanders, P.: δ-stepping: a parallelizable shortest path algorithm. J. Algorithm. **49**(1), 114–152 (2003)
27. Otsuki, K., Kobayashi, Y., Murota, K.: Improved max-flow min-cut algorithms in a circular disk failure model with application to a road network. Eur. J. Oper. Res. **248**(2), 396–403 (2016)

28. Saito, H., Toyoda, M., Kitsuregawa, M., Aihara, K.: A large-scale study of link spam detection by graph algorithms. In: Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web, pp. 45–48. ACM (2007)
29. Apache Spark: Apache spark^{TM} is a fast and general engine for large-scale data processing (2016)
30. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. HotCloud **10**, 10 (2010)