


SPECIAL ISSUE PAPER

# Max-flow min-cut algorithm with application to road networks

Varun Ramesh<sup>1</sup>  | Shivaneer Nagarajan<sup>1</sup> | Jason J. Jung<sup>2</sup> | Saswati Mukherjee<sup>3</sup>

<sup>1</sup>Department of Computer Science and Engineering, Anna University, Chennai, India

<sup>2</sup>Department of Computer Science Engineering, Chung-Ang University, Seoul, Korea

<sup>3</sup>Department of Information Science and Technology, Anna University, Chennai, India

**Correspondence**

Varun Ramesh, Department of Computer Science and Engineering, Anna University, Chennai, India.

Email: varun.ceg.95@gmail.com

## Summary

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 lognormal graphs and real-world datasets using the distributed Edmonds-Karp algorithm. The algorithm is deployed on a stand-alone cluster using Spark. In our experiments, we compare the runtime between a single-machine implementation and cluster implementation. We analyze the impact of communication cost on runtime for large lognormal graphs. Further, we record the runtime of the algorithm for various real-world datasets having similar average out-degree and number of edges but different diameters. The observations indicate that for such a set of similar graphs, runtime increases slowly with an increase in diameter. Additionally, we apply this model on large urban road networks to evaluate the minimum number of sensors required for surveillance of the entire network. To validate the feasibility of this extension, we tested the model with large road network datasets having more than 2.7 million edges. Thus, we believe that our model can enhance the safety of today's dynamic large urban road networks.

## KEYWORDS

big data processing, distributed systems, max-flow min-cut algorithm, Pregel

## 1 | INTRODUCTION

Max-flow min-cut problem has a wide variety of applications in various domains including spam site discovery,<sup>1</sup> community identification,<sup>2</sup> and network optimization.<sup>3</sup> For instance, this problem has been previously applied on road networks to assess the reliability of the network.<sup>4</sup> Road networks such as that of California are extremely large, and it has 1 965 206 nodes and 2 766 607 edges.<sup>5</sup> Protection and monitoring of such road networks for safety is a challenging task. Furthermore, to analyze such large data, we find that 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, is an ideal solution for such large computations. Halim et al in 1 study<sup>6</sup> 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.

The classical max-flow algorithms<sup>7</sup> were designed under assumption that the entire graph is small enough fit into main memory. Such algorithms are not directly applicable to run on distributed systems since they require a global view of the entire graph.<sup>6</sup> However, max-flow algorithms such as Edmonds-Karp algorithm<sup>8</sup> and push-relabel algorithm<sup>9</sup> have good distributed settings.<sup>10,11</sup> A detailed

explanation as to why we have chosen the Edmonds-Karp algorithm is presented in Section 4. We adopt the distributed algorithm from 1 study<sup>10</sup> and implement it on a single machine initially. Next, we implement the same on a cluster of 3 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.

Further, we collect a set of graphs from previous studies,<sup>5,12</sup> which have a similar number of edges and average outdegree to analyze the impact of diameter on runtime. We observe that the number of edges, average outdegree, max-flow value, and diameter are important parameters that influence the runtime of the algorithm. 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 and the impact of structural properties are analysed based upon the max-flow min-cut algorithm.

For  $n$  vertices and  $m$  edges, the complexity as achieved in another study<sup>10</sup> is  $O(cm/k) + O(cn \log k)$  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 because of smaller diameter of the graph<sup>13</sup> 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 dataset, and we believe that our proposal can enhance the safety of road networks throughout the world. We have tested our model on real-time road network datasets from another study.<sup>5</sup>

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 Section 3. Various phases of the distributed algorithm are explored in Section 4. Our experiments on a single machine and cluster result are compared in Section 5, along with the investigation of practical communication costs in the cluster. Further, the impact of diameter, average outdegree, and max-flow value on runtime is presented in this section. Extension of the max-flow min-cut implementation to evaluate the minimum number of sensors required to provide surveillance in a large city modeled as a flow graph is performed in Section 6. We summarize in Section 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,<sup>8</sup> the blocking flow algorithm of Diniz,<sup>14</sup> the push-relabel algorithm of Goldberg and Tarjan,<sup>9</sup> and the binary blocking flow algorithm of Goldberg and Rao.<sup>15</sup> The first method developed is the Ford-Fulkerson method.<sup>16</sup> 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)<sup>17</sup> 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<sup>18</sup> provides programmers with an application program interface (API) centered on a data structure called the resilient distributed dataset, a read-only multiset of data items distributed over a cluster of machines, that is maintained in a fault-tolerant manner.<sup>19,20</sup> The availability of resilient distributed datasets 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<sup>21</sup> for iterative operations because of the cost paid by Hadoop for data reloading from disk at each iteration.<sup>22</sup>

Spark provides an API for graph algorithms that can model the Pregel<sup>23</sup> 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 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.<sup>23</sup> 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. MapReduce is sometimes used to mine large graphs,<sup>24</sup> but this can lead to suboptimal performance and usability issues.<sup>23</sup> Pregel has a natural graph API and is much more efficient support for iterative computations over the graph.<sup>23</sup>

### 2.2 | Other related works

General-purpose distributed dataflow frameworks such as MR are well developed for analyzing large unstructured data directly, but direct implementation of iterative graph algorithms using complex joins is a challenging task. GraphX<sup>25</sup> is an efficient graph processing framework embedded within the Spark distributed dataflow system. It solves the abovementioned implementation issues by enabling composition of graphs with tabular and unstructured data. Additionally, GraphX allows users to choose either the graph or the collective computational model based on the current task with no loss of efficiency.<sup>25</sup>

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 1 study.<sup>4</sup> Placing sensors across the road network using the min-cut of the graph is explored in another study<sup>26</sup> 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<sup>3</sup> is a directed graph  $G = (V, E)$  where each edge  $(u, v) \in E$  has a nonnegative capacity  $c(u, v) \geq 1$ . There are 2 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 3 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  $(u, v) \in V - \{S, T\}$ .

Diameter and average degree are 2 important factors used for the analysis of the max-flow algorithm. The distance between 2 nodes in a network is defined as the number of edges needed to reach 1 node from another. The eccentricity of a node can then be defined as the maximal distance from that node to any other node

$$e(u) = \max_{v \in V} d(u, v). \quad (1)$$

The diameter  $\delta$  of a graph equals the longest shortest path in the network, or alternatively, it can be defined as the largest eccentricity of all nodes<sup>12</sup>

$$\delta = \max_{u \in V} e(u) = \max_{u, v \in V} d(u, v). \quad (2)$$

Average degree of a graph is defined as

$$1/|V| \sum_{u \in V} d(u) = 2|E|/|V|. \quad (3)$$

Augmenting path and residual graph are 2 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  $G$  is the 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<sup>9</sup> 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  $O(v^2 \sqrt{e})$ .<sup>27</sup> A parallel implementation of the push-relabel method, including some speedup heuristics applicable in the sequential context, has been studied in another study.<sup>11</sup>

The push-relabel algorithm has a good distributed setting but has the main problem of low parallelism achieved at times.<sup>28</sup> 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.<sup>29</sup> Moreover, other than the degenerate cases, max-flow algorithms, which make use of dynamic tree data structures, are not efficient in practice because of overhead exceeding the gains achieved.<sup>30</sup> 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 lognormal graphs is that they have a diameter close to  $\log n$ .<sup>13</sup> 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.  $w_1, w_2, \dots, w_n$  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:      $L \leftarrow \text{augmentingpath}(S, T)$ 
4:      $\text{Minflow} \leftarrow \min(w_1, w_2, \dots, w_n)$  in  $L$ 
5:     Update Residual Graph and overall  $F_{max}$ 
6:   return  $F_{max}$ 

```

---

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 follow either  $\Delta$ -stepping method<sup>31</sup> or the Pregel's procedure for calculating shortest path. The  $\Delta$ -stepping method has been implemented before in Crobak et al.<sup>32</sup> 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<sup>23</sup> 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 3 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<sup>10</sup>


---

```

1: Make all distance value except source as  $\infty$  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  $(d_i = d_j \text{ and } c_j < c_i)$  :  $m_i$ 
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 MR 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  $R_G$ <sup>10</sup>


---

```

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

```

---

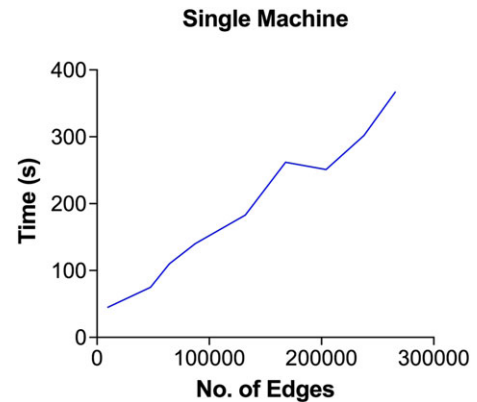
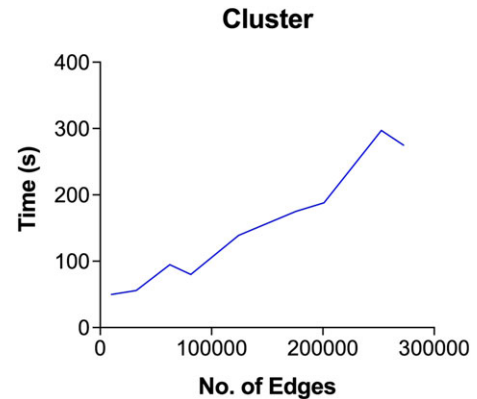
**Runtime analysis.** For a graph with  $n$  vertices and  $m$  edges, cluster size  $k$ , the runtime complexity, is mentioned as follows. For the  $s - t$  shortest path algorithm, the current iteration's distance value for a node will be less than what it receives in the next iteration. Thus, only 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  $O(n \log k)$ . 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 max-flow value  $c$ . Therefore, the overall runtime is as  $O(cm/k) + O(cn \log k)$ .

**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 is bound by the max-flow value  $c$ . The overall communication cost is  $O(cm) + O(cnk)$ . Both runtime and 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 3 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 lognormal distribution of the outdegrees as done in previous literature.<sup>23</sup> 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.<sup>23</sup>

Spark user interface is used to view the timeline of the spark events, Directed Acyclic Graph 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<sup>13</sup> will imply lower practical runtime for the max-flow min-cut algorithm implemented. In Figure 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 both Figures 1 and 2 for instance at 0.18 million edges in Figure 1 and at 0.26 million edges in Figure 2. These irregularities occur because of a decrease in diameter of the graphs, which results in reduction of runtime. The impact of diameter on runtime has been detailed below in this section.

**FIGURE 1** Single-machine analysis**FIGURE 2** Cluster analysis

**Communication cost impact on runtime.** The cluster plot provides great insight into communication cost. It can be observed by comparing the 2 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 is 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 seconds and the max-flow value is 66. Theoretically, the runtime is dominated by  $O(cm)$ , and for this instance, it grows to  $\sim 66 \times 1.3 \times 10^6$  seconds. This worst-case evaluation is much higher than the experimentally observed value of 520 seconds 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.

**Impact of diameter, max-flow value, and average outdegree.** To inspect the effect of graph's topological structure and diameter, we have taken a set of real-world graphs from the literature.<sup>5,12</sup> For this analysis, we have collected graphs with similar average outdegree and number of edges. Similarity between these parameters is an important prerequisite before we make runtime comparisons based on the diameter. This is because numbers of edges and outdegree play a significant role in determining the runtime. As mentioned before, the runtime of the algorithm is  $O(cm/k) + O(cn \log k)$  where the first term dominates in most cases. From these bounds, we can understand that number of edges will impact the runtime. To account for the impact of average outdegree, we can compare instances from Figures 2 and 3. In Figure 2, the average outdegree of the graphs is approximately 127.1, and in Figure 3, the average outdegree of the graphs is approximately 2.2, and all the graphs in Figure 3 have approximately 0.25 million edges. In the former plot, the average runtime observed is 248 seconds where as in the latter, the average runtime is 55 seconds; this drop is due to the reduced average outdegree. Thus, to focus primarily on the impact of diameter on runtime of the algorithm, we collected graphs of similar number of edges and average outdegree but varying diameters.

Figure 3 plots the runtime versus diameter of the graphs. The set of graphs analyzed have approximately 0.25 million edges and aver-

age degree approximately 2.2 with varying diameters. The max-flow value was also fixed so that any variation in runtime can be imputed primarily to diameter. It is observed that runtime slowly increases with an increase in diameter in most cases. Though most real-world graphs have low diameter when compared to number of edges, slightly larger diameter will increase the runtime as the shortest path computation in the residual graph is bounded by the diameter. Interestingly, experiments have shown that apart from average outdegree, diameter, and number of edges, the max-flow value "c" also impacts the runtime of the algorithm but it is less important than the number of edges. In addition, we also observed that the number of iterations consumed by the algorithm was lesser than the max-flow value.

## 6 | APPLICATION OF DISTRIBUTED MAX-FLOW MIN-CUT IMPLEMENTATION

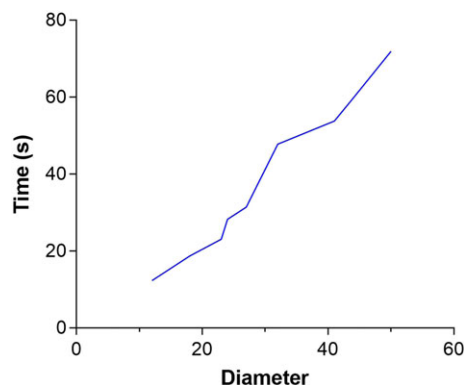
Leading countries use public video surveillance as a major source to monitor population movements and to prevent terrorist activities. The use of closed-circuit television cameras and sensors in the detection of illegal activities, data collection, recording of preincident and postincident image sequences 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.<sup>4</sup> 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 modeling 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 California is enormous. Therefore, we evaluate the min-cut of such graphs using our distributed max-flow min-cut implementation. Sensors or cameras placed in the cut-set will monitor the 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, we added an artificial super source and super sink node 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 analyzed our model on a number of lognormal graphs as well as real-world road networks such as California, Texas, and Pennsylvania. A particular example from the set of randomly generated lognormal graphs had approximately 1.2 million 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.

Table 1 provides the results when our model was applied on large real-world road networks of Texas, California, and Pennsylvania. In



**FIGURE 3** Impact of diameter on runtime



**TABLE 1** Road network dataset analysis

Road Network	Vertices	Edges	Average Outdegree	Diameter	Runtime, s
California	1965206	2766607	2.815	865	529.523
Pennsylvania	1088092	1541898	2.834	794	253.056
Texas	1379917	1921660	2.785	1064	325.327

most of the practical cases, only a smaller subgraph has to be monitored during an attack rather than the entire road network. Based on the parameters such as number of attackers and density of population, the subgraph that has to be monitored is determined. These subgraphs can be of any reasonable size or shape. Hence, to mimic the practical situations, we recorded the runtime for random source and sink combinations in these real-world networks and tabulated the average of different runtimes. A similar approach has been attempted on the road network of New York City in previous study<sup>26</sup> where they evaluate the min-cut set of a selected subgraph. The increase in runtime with the number of edges supports our previous claim in Figure 1 where we observe a similar trend. Additionally, we observe that the real-world graphs have a much smaller average outdegree. As a result of reduction in this factor, the runtime of the algorithm is lesser compared with lognormal graph runtime. Tabulated results indicate the scalability and practical feasibility of our approach.

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 2.7 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 that 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 analyses have been performed experimentally but do not include communication cost observations over a cluster for the max-flow algorithm on Spark. We believe that 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. Further, we examine the impact of the number of edges, average outdegree, max-flow value, and diameter on the runtime of the algorithm. We then propose a method of evaluation and placement of sensors in a large city with millions of road segments, which is modeled as a flow graph. The cut-set of a lognormal graph with approximately 1.2 million edges is approximately 0.000047 times smaller than the total number of edges. Thus, our experiments show a promising and scalable approach to place sensors in large urban road networks based on the distributed max-flow min-cut implementation.

## REFERENCES

1. Saito H, Toyoda M, Kitsuregawa M, Aihara K. A large-scale study of link spam detection by graph algorithms. *Proceedings of the 3rd International Workshop on Adversarial Information Retrieval on the Web*. ACM, Banff, AB, Canada; 2007:45–48.
2. Flake GW, Lawrence S, Giles CL. Efficient identification of web communities. *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, Boston, MA, USA; 2000:150–160.
3. Ahuja RK, Magnanti TL, Orlin JB. Network flows: Theory, algorithms and applications; 1993.
4. 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*. 2016;248(2):396–403.
5. Leskovec J, Krevl A. June 2014. SNAP Datasets: Stanford large network dataset collection. Available from: <http://snap.stanford.edu/data>.
6. Halim F, Yap RHC, Wu Y. A MapReduce-based maximum-flow algorithm for large small-world network graphs. *2011 31st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, Minneapolis, MN, USA; 2011:192–202.
7. Goldberg AV. Recent developments in maximum flow algorithms. *Scandinavian Workshop on Algorithm Theory*. Springer, Stockholm, Sweden; 1998:1–10.
8. Edmonds J, Karp RM. Theoretical improvements in algorithmic efficiency for network flow problems. *JACM (JACM)*. 1972;19(2):248–264.
9. Goldberg AV, Tarjan RE. A new approach to the maximum-flow problem. *JACM (JACM)*. 1988;35(4):921–940.
10. Dancoisne B, Dupont E, Zhang W. Distributed max-flow in spark; 2015.
11. Goldberg AV. Efficient graph algorithms for sequential and parallel computers. *PhD thesis*, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1987.
12. Kunegis J. Konect: the koblenz network collection. *Proceedings of the 22nd International Conference on World Wide Web*. ACM, Rio de Janeiro, Brazil; 2013:1343–1350.
13. Leskovec J, Kleinberg J, Faloutsos C. Graphs over time: densification laws, shrinking diameters and possible explanations. *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*. ACM, Chicago, IL, USA; 2005:177–187.
14. Dinic EA. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math Doll*. 1970;11(5):1277–1280.
15. Goldberg AV, Rao S. Beyond the flow decomposition barrier. *JACM (JACM)*. 1998;45(5):783–797.
16. Ford LR, Fulkerson DR. Maximal flow through a network. *Can J Math*. 1956;8(3):399–404.
17. Dean J, Ghemawat S. Mapreduce: Simplified data processing on large clusters. *Commun ACM*. 2008;51(1):107–113.
18. Apache Spark. 2016. Available from: <http://spark.apache.org/>.
19. Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. *HotCloud*. 2010;10:10–10.
20. Bello-Organ G, Jung JJ, Camacho D. Social big data: recent achievements and new challenges. *Inf Fusion*. 2016;28:45–59.
21. Apache Hadoop. Hadoop; 2009.
22. Gu L, Li H. Memory or time: performance evaluation for iterative operation on hadoop and spark. *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC\_EUC)*. IEEE, Zhangjiajie, Hunan Province, P.R. China; 2013:721–727.

23. Malewicz G, Austern MH, Bik AJC. Pregel: A system for large-scale graph processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, Indianapolis, IN, USA; 2010:135–146.
24. Kang U, Tsourakakis CE, Faloutsos C. Pegasus: A peta-scale graph mining system implementation and observations. *2009 Ninth IEEE International Conference on Data Mining*. IEEE, Miami, FL, USA; 2009:229–238.
25. Gonzalez JE, Xin RS, Dave A. Graphx: Graph processing in a distributed dataflow framework. *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, vol. 14, Broomfield, CO; 2014:599–613.
26. Barnett RL, Bovey DS, Atwell RJ, Anderson LB. Application of the maximum flow problem to sensor placement on urban road networks for homeland security. *Homeland Secur Affairs*. 2007;3(3):1–15.
27. Cheriyan J, Maheshwari SN. Analysis of preflow push algorithms for maximum network flow. *SIAM J on Comput*. 1989;18(6):1057–1086.
28. Kulkarni M, Burtcher M, Inkulu R, Pingali K, Casçaval C. How much parallelism is there in irregular applications? *ACM SIGPLAN Notices*, vol. 44. ACM, Edinburgh, Scotland; 2009:3–14.
29. Cherkassky BV, Goldberg AV. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*. 1997;19(4):390–410.
30. Badics T, Boros E. Implementing a maximum flow algorithm: experiments with dynamic trees. *Network Flows and Matching: First DIMACS Implementation Challenge*. 1993;12:11–15.
31. Meyer U, Sanders P.  $\delta$ -stepping: A parallelizable shortest path algorithm. *J of Algorithms*. 2003;49(1):114–152.
32. Crobak JR, Berry JW, Madduri K, Bader DA. 2007 IEEE Advanced shortest paths algorithms on a massively-multithreaded architecture. *IEEE International Parallel and Distributed Processing Symposium*. IEEE, Long Beach, California USA; 2007:1–8.

**How to cite this article:** Ramesh V, Nagarajan S, Jung JJ, Mukherjee S. Max-flow min-cut algorithm with application to road networks. *Concurrency Computat: Pract Exper*. 2017;29:e4099. <https://doi.org/10.1002/cpe.4099>