Distributed Max-Flow in Spark

Benoît Dancoisne, Emilien Dupont, William Zhang

June 3, 2015

{benoitd,edupont,wzhang4}@stanford.edu

Stanford University CME323 Final project

Abstract

We describe a distributed algorithm to solve the max-flow problem based on the Edmonds-Karp algorithm. Using k machines, with a graph on n nodes and m edges and a max flow of c, the implementation improves the single machine runtime from $O(cm)$ to $O(cm/k)$. Furthermore, for an appropriate choice of k, the algorithm runs in $O(cn\log(m/n))$. The communication costs incurred are on the order of $O(cm)$, although this bound is much lower for most applications. The algorithm has been implemented in Spark as a `maxFlow` function which takes as argument a graph, source node and target node and returns an RDD with the maximum flow. The implementation can be found at `github.com/benoitdancoisne/SparkMaxFlow`

1 Introduction

The max-flow problem is defined as follows: for a directed graph G with edge capacities and given a source node s and a target node t, what is the maximum flow that can be pushed through from s to t through the edges of G?

For a graph with n nodes and m edges, the fastest current single machine implementation finds a $1-\epsilon$ approximation to the max-flow in $O(\frac{mn^{1/3}}{\epsilon^{11/3}})$ time, where the the big-O notation hides polynomial log terms (see [1]). This algorithm is however very complex and not readily parallelizable. There already exists parallel max-flow algorithms (see e.g. [3] and [6]), but these are complicated and do not lend themselves to an implementation in Spark. In contrast, our distributed algorithm is simple and easily implemented in Spark: it consists in Pregel shortest path search (see [5]) and a MapReduce per iteration. It is based on the Edmonds-Karp algorithm, which runs in $O(\min(cm, m^2n))$ where c is the max-flow in the graph, a runtime which we reduce to $O(cm/k)+O(cn\log k)$ while incurring a communication cost of $O(cm)+O(cnk)$ (for k machines).

2 Edmonds-Karp algorithm

Before presenting the distributed max-flow algorithm, we review the single machine Edmonds-Karp algorithm. We first introduce the concept of a residual graph, which is central to this algorithm.

Definition 2.1. Let $G(V,E)$ be a (directed) graph. For each edge (u,v), define $c(u,v)$ as the edge's capacity. The residual graph $G_f(V,E_f)$ of $G(V,E)$ with respect to a flow f is the graph such, that for every $e \in E$:

- if $f(e) < c(e)$, an edge with capacity $c'(e) = c(e) - f(e)$ is placed in the same direction as e.

Algorithm 1 Edmonds-Karp

While there exists a path from s to t in residual graph R_G:

- Find shortest flow augmenting path P between s and t in R_G
- Find the maximum flow $f_{\max}$ you can push along P
- Update R_G using P and $f_{\max}$

- if $f(e) > 0$, an edge with capacity $c'(e) = f(e)$ is placed in the direction opposite of e.

The algorithm is succinctly outlined in algorithm 1.

For a more detailed description of the algorithm, see [2]. The number of iterations for the algorithm to terminate is at most $\min(c, m(n-1))$ where c is the value of the maximum flow (see e.g. [7] for a proof). In principle, we have no bound on c, but for most large scale graph applications $c \ll m(n-1)$, so for our purposes, we let the number of iterations be upper bounded by c. Note that even then, c is a pessimistic bound. Indeed, this corresponds to the case where exactly one unit of flow ($f_{\max} = 1$) can be shipped at every iteration, which is unlikely (except for unweighted graphs).

The total running time of this algorithm on a single machine is then on the order of $O(cm)$, since finding the shortest admissible path from s to t is usually done via breadth-first-search and thus done in $O(m)$. However, shortest path finding via Pregel should in most cases outperform the breadth-first-search approach.

3 Distributed algorithm

Before the algorithm is presented, some assumptions need to be stated, namely

1. The number of nodes n is such that they can fit on a single machine
2. The number of edges m is such that they cannot fit on a single machine and must be distributed on the cluster
3. The capacities of the edges are integers (this is necessary for convergence of the algorithm)

The reasons for these assumptions will become clear in the following sections. Furthermore, the following data structures are used in our Spark implementation:

1. The graph object of *Spark GraphX* is used to represent the residual graph
2. The flows are stored in an RDD (Resilient Distributed Dataset)
3. The shortest path is stored in an array (since its size is bounded by n)

3.1 Algorithm outline

A high level overview of the whole algorithm is shown in 2. Each part of the algorithm is detailed in the following subsections along with a runtime and communication cost analysis for each of them. Note that most bounds are derived for the worst-case scenarios. Although the average case will tend to be much better, quantifying it is rather difficult and highly dependent on the structure of the graph, especially given that the analysis will be on the residual graph computed at each step, and not the original graph.

Algorithm 2 Outline

Initialization:

- Set flows ($f : E \to \mathbb{R}^+$) in all edges to 0
- Set residual graph R_G equal to initial graph

While there is a flow augmenting path from s to t in R_G:

- Find the shortest augmenting path P between s and t in R_G
- Find capacity $c_{\min}$ along P
- Broadcast P
- Update flows
- Update R_G using P and $c_{\min}$

3.2 Shortest Path using Pregel

In order to find the shortest path in the residual graph, we used the Pregel framework as described in Google's Pregel paper [5]. The implementation was done using the Pregel API for Spark. The algorithm used is shown below in 3.

Algorithm 3 Shortest path using Pregel

Each vertex has an attribute (d, c, id) where d will represent distance from s, c is the minimum capacity the node has seen so far, and id the id of the node from which the previous message was received. Denote with subscripts s and t those attributes relating to the source and the target respectively.

Initialization:

- Initialize all distances d to ∞ except for s (initialized to 0)

While $d_t = \infty$:

- **Message sent by node i:** $(d_i + 1, c_i, i)$ to each neighbor j only if $c_{ij} > 0$
- **Function applied to node j upon receiving message** $(d_i + 1, c_i, i)$:
  - set $d_j \coloneqq \min(d_j, d_i + 1)$
  - if $(d_i + 1 < d_j)$
    - set $id_j \coloneqq i$
    - $c_j \coloneqq \min(c_i, c_{ij})$ where c_{ij} is the capacity along edge (i, j)
- **Merging functions upon receiving** $m_i = (d_i + 1, c_i, i)$ and $m_j = (d_j + 1, c_j, j)$:
  - if $d_i < d_j$: m_i
  - else if ($d_i = d_j$ and $c_j < c_i$): m_i
  - else m_j

Communication cost: Initially, only the source node sends a message to its neighbors. At each

subsequent superstep p, a node i will send out messages to its neighbors only if its state was modified at superstep $p-1$, i.e. if it received a message from a node j such, that $d_j + 1 < d_i$. Therefore, since the distances we send out are incremented by 1 at each superstep, a node that has already sent messages to its neighbors (i.e. has already had its distance attribute modified), will never do so again (since subsequent distances received will always be larger than the current one). As such, we will at most send one message per edge. Each message is of constant size, so the total communication cost is $O(m)$.

Note: this bound is very pessimistic, since the path finding algorithm terminates once the target t has been hit. Since most large graphs have very low diameter compared to m (for instance random graphs and real-world graphs are closer to $\log n$ in diameter as shown in [4]), in the majority of practical applications, the bound will be significantly better.

Runtime: Initializing each vertex is done in $O(n)$. To obtain the runtime for the Pregel procedure, we divide the number of messages sent by the number of machines k. So the total runtime is $O(n)+O(m/k)$.

3.3 Updating the flows

Once the shortest path P has been determined, the maximum flow pushed through P is set to be the smallest capacity of edges in P, i.e. $f_{\max} = c_{\min}$ (the minimum capacity of an edge along a path is equivalently the maximum flow you can ship along the path). Now we update the flows by broadcasting the path P, and incrementing $f(e)$ for every $e \in P$.

Communication cost: We broadcast P, so this is a one-to-all communication. Since the path is at most length n and since there are k machines, this is $O(nk)$.

Runtime: Bittorrent broadcast is done in $O(n\log k)$. For the actual update, we have to iterate over all edges, but since P is broadcast, the process is completely parallel. So with k machines, runtime is $O(m/k) + O(n\log k)$.

Note: as before, the bound n on the number of edges on the shortest path between s and t is very crude. In many large graphs, this will tend to be much lower.

3.4 Updating the Residual Graph

The residual graph is built via a MapReduce operation on the graph obtained from the previous iteration of Edmonds-Karp (see algorithm 4).

Algorithm 4 Building residual graph R_G

Each key value pair is of the form $((i, j) : \text{capacity})$
Map (input: edge; output: edge):

- if P contains edge (i, j) in R_G:
  - emit $((i, j) : c - f_{max})$
  - emit $((j, i) : f_{max})$
- else: emit $((i, j) : c)$ (no changes)

Reduce: sum

Communication cost: The map emits at most 2 edges per edge in the path, but without real knowledge of what edges are in the path, in the naive implementation all edges of the graph need to be looked up. The resulting shuffle size is $O(m)$.

Runtime: We emit at most 2 edges for every edge in graph. The reduce operation will at most have to sum 2 values per edge. Since this is done in parallel we have $O(m/k)$ runtime.

Remark. Since there are at most n edges in the shortest path, by creating a hash of the edge ID's beforehand, we can ship the edges on said path to the correct machines and not cause any communication between worker machines. The above bounds thus should respectively become $O(n)$ and $O(n/k)$ in the worst case, though this was not done in our Spark implementation of the algorithm.

4 Runtime and Communication cost

Note that the analysis in Section 3 was for one iteration. Recall that, as mentioned in Section 2, the number of iterations of the algorithm is c where c is the max-flow/min-cut. Table 1 summarizes the communication costs of the algorithm, and compares the runtime for single machine and distributed versions (both per iteration).

Step	**Cost**
Shortest path	$O(m)$
Broadcast	$O(nk)$
Residual graph update	$O(m)$

(a) Communication cost

Step	**Sequential**	**Distributed**
Shortest path	$O(m)$	$O(m/k)$
Path building	$O(n)$	$O(n)$
Broadcast	0	$O(n\log(k))$
Residual graph update	$O(m)$	$O(m/k)$
Flow update	$O(m)$	$O(m/k)$

(b) Runtime

Table 1: Summary of runtimes and communication cost per iteration

4.1 Analysis

Claim: The total runtime of the algorithm is $O(cm/k) + O(cn\log k)$

Proof. From section 3, adding up each part of the algorithm yields a runtime of $O(m/k) + O(n\log k)$ per iteration. It takes at most c iterations for the algorithm to terminate. So the total runtime is $O(cm/k) + O(cn\log k)$. ■

Claim: The total communication cost of the algorithm is $O(cm) + O(cnk)$

Proof. From section 3, adding up each part of the algorithm yields a communication cost of $O(m)+O(nk)$ per iteration. It takes at most c iterations for the algorithm to terminate. So the total communication cost is $O(cm) + O(cnk)$. ■

Remark. The terms involving m (number of edges) are absolute worst-case scenario bounds. As mentioned before, most large graphs for practical applications have low diameters. Communication cost and runtime per iteration (and thus overall) are therefore usually much better than $O(m)$.

Note that in most cases the first term both for the runtime and communication cost will dominate. In this case the runtime becomes $O(cm/k)$ and recalling the $O(cm)$ runtime of the single machine algorithm, we see that this is completely parallel. More precisely:

- If $k\log k < m/n$ the runtime of the algorithm is $O(cm/k)$ and the communication cost is $O(cm)$
- If $k\log k > m/n$ the runtime of the algorithm is $O(cn\log k)$ and the communication cost is $O(cnk)$

Minimising the expressions for runtime and communication cost we observe that the optimal number of machines is $k = m/n$. In this case we have:

Corollary. If $k = m/n$, the total runtime of the algorithm is $O(cn \log(m/n)) = O(cn \log k)$ and the communication cost is $O(cm)$.

So for a carefully chosen number of machines, the runtime becomes linear in the number of vertices up to a $\log(m/n)$ term, which is a considerable improvement over usual linearity in the number of edges. The communication cost however, is still rather high. But as mentioned previously, this is a very pessimistic bound and will likely be much lower for the majority of applications.

4.2 Experiments

In order to benchmark the performance of the algorithm, we tested our max-flow implementation on random graphs. Said graphs were generated according to a log-normal distribution with the same parameters as those used in Google's Pregel paper [5]. The graphs generated are a good approximation of large scale graphs such as web graphs and social networks. Indeed, a majority of the vertices have a relatively low degree (in our case the mean outdegree is approximately 127), while a few outliers have very high degrees on the order of 10^5. We ran the tests on a single quad core machine with 8GB of RAM.
Figure 1 shows a plot of the runtime of the `maxFlow` function against the number of edges in the graph (m ranging from ten thousand to over 200 thousand). As can be seen, the runtime increases approximately linearly with m, confirming the theoretical results. The runtime is, however, very strongly dependent on the structure of the graph (for a given m, up to an order magnitude of runtime difference was observed). Interestingly, it is observed experimentally that while the maximum flow value c affects the runtime, it is less important than the number of edges m. In addition, while testing the algorithm, it was also observed that the number of iterations was considerably lower than the max-flow value. As previously mentioned, the seemingly high bounds on communication and runtime are almost always orders of magnitude lower in practice.

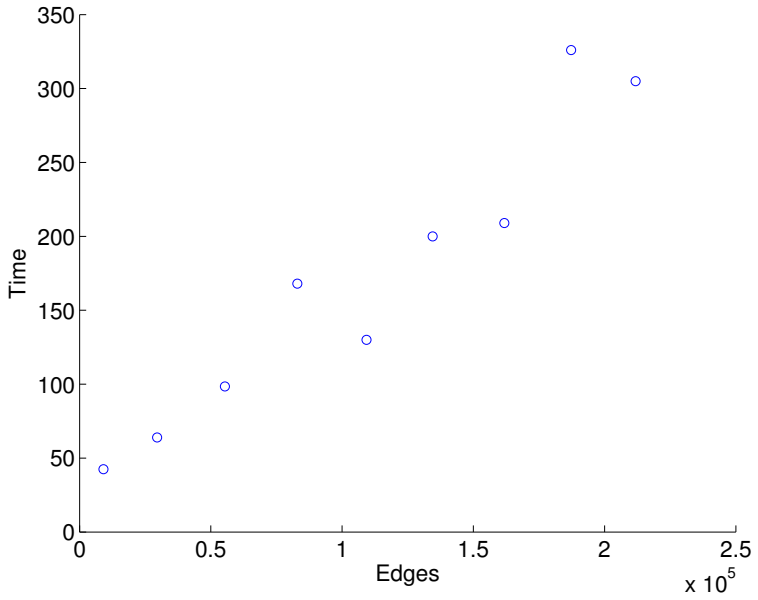

Figure 1: Runtime vs number of edge on a quad-core single machine. Note that the x-axis is in units of hundreds of thousands of edges, and time in seconds. Note that each point of the plot represents an average of several different random graphs with the same number of edges and nodes. Indeed, the runtime is very dependent on the structure of the graph so taking averages was necessary to get useful results.

4.2.1 Largest problem

The largest graph tested had over half a million edges and ran in 702 seconds. Even larger graphs could also have been handled by the `maxFlow` function, but as m grows large we expect the communication costs to be more significant - essentially a non-issue on a single machine, making the comparison less sensible. Note also that, for this graph, we have $m = 587152$ and $n = 2500$ so the ratio $m/n = 235$, i.e. the optimal number of machines would be 235. Provided the ratio stays roughly on the same order of magnitude, this is not an unrealistic number of machines for much larger problems, and should yield a runtime closer to $O(cn \log(m/n))$ (see Section 4), which is a significant improvement.

4.2.2 Verification

To check the correctness of our algorithm, we generated a few random graphs with 100 nodes and 1000 edges, for which we computed the maximum flow using the *Matlab* built-in function `graphmaxflow`. This gave us a reference with which we could compare the results given by our *Scala* implementation. The results coincided.

5 Conclusion

Our distributed implementation of the Edmonds-Karp max-flow algorithm achieves a runtime which is essentially optimal compared to the sequential version ($O(cm) \to O(cm/k) + O(cn \log k)$). Furthermore, for an appropriately chosen number of machines this can be reduced to $O(cn \log(m/n))$, a significant improvement. The algorithm does however suffer from potentially large communication costs which must be taken into account when comparing the actual time needed to execute the algorithm. The communication costs were theoretically found to be bounded by $O(cm)$ although we expect this to be much lower in practice. The algorithm scales with the number of edges m, and is limited by the number of nodes n, as these must fit on a single machine.

The algorithm was implemented in Spark using Pregel-like message passing as well as MapReduces. It was tested on various graphs and showed good performance on large graphs. Unfortunately, the algorithm was not tested on a cluster, so the impact of the communication costs has not been determined experimentally.

The code is available on the following `Github` repository.

References

[1] Paul Christiano, Jonathan A Kelner, Aleksander Madry, Daniel A Spielman, and Shang-Hua Teng. Electrical flows, laplacian systems, and faster approximation of maximum flow in undirected graphs. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 273–282. ACM, 2011.

[2] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2):248–264, April 1972.

[3] Zhipeng Jiang, Xiaodong Hu, and Suixiang Gao. A parallel ford-fulkerson algorithm for maximum flow problem. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, page 70. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013.

[4] Jure Leskovec. CS 224W: Social and Information Network Analysis. University Lecture, 2014.

[5] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[6] Yossi Shiloach and Uzi Vishkin. An o(n2logn) parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[7] Reza Zadeh. CME 305: Discrete Mathematics and Algorithms. University Lecture, 2014.