Distributed Max-flow algorithm

Benoît Dancoisne, Emilien Dupont, William Zhang

{benoitd,edupont,wzhang4}@stanford.edu

Stanford University
CME323 Final project

June 1, 2015
Overview

1. Edmonds-Karp algorithm for max-flow
   - Single Machine Algorithm
   - Distributed Algorithm
   - Details

2. Analysis
   - Communication cost
   - Runtime
Edmonds-Karp algorithm for max-flow

- We increment the flow from $s$ to $t$ by finding a flow-augmenting path.
- We do this by finding a path in the residual graph.
- The total flow is increased by the maximum capacity found on our path.
- Maximal flow is found when there are no more flow-augmenting paths.
- Note that we can lower the flow on a particular edge to favor another path.
Residual graph toy example
Residual graph toy example

![Residual graph diagram](image-url)
Residual graph toy example
Assumptions and Methods

- $n$ vertices: can fit on a single machine
- $m$ edges: too large to fit
- Integer edge capacities
- Use Pregel and MapReduces to distribute
Distributed max-flow

**Initialization:**
- Set flows in all edges to 0
- Set residual graph $R_G$ equal to initial graph

**While there is a path from s to t in $R_G$:**
- Find the shortest path $P$ between $s$ and $t$ in $R_G$
- Find max flow $f_{max}$ you can push along $P$
- Broadcast $P$
- Update flows
- Update $R_G$ using $P$ and $f_{max}$
We use the graph object provided by GraphX to build the residual graph.

Edges and flows are stored in a RDD which will be updated at each iteration (each time we find a path).

The path found in the residual graph is stored in an array of size $O(n)$ that will be broadcasted.
Finding the shortest path in Pregel

- Vertex attribute: \((d, c, id)\)
- \(d\): distance from source \(s\)
- \(c\): minimum capacity the node has seen so far
- \(id\): node from which previous message was received
Finding the shortest path in Pregel

- **Vertex attribute**: \((d, c, id)\)
  - **d**: distance from source \(s\)
  - **c**: minimum capacity the node has seen so far
  - **id**: node from which previous message was received
- Each node propagates its id, the minimum capacity found so far and the distance from the source
Finding the shortest path in Pregel

- Vertex attribute: $(d, c, id)$
- $d$: distance from source $s$
- $c$: minimum capacity the node has seen so far
- $id$: node from which previous message was received
- Each node propagates its id, the minimum capacity found so far and the distance from the source
- Once we reached the target $t$, we can backtrack to find the actual path
Finding the shortest path in Pregel

- **Vertex attribute:** \((d, c, id)\)
  - \(d\): distance from source \(s\)
  - \(c\): minimum capacity the node has seen so far
  - \(id\): node from which previous message was received
- Each node propagates its id, the minimum capacity found so far and the distance from the source
- Once we reached the target \(t\), we can backtrack to find the actual path
- If two paths have the same length, we choose the one with maximum capacity (flow)
Finding the shortest path in Pregel

Communication cost
Because the state of a node is changed once at most, there will be at most one message sent per edge: $O(m)$.

Runtime
Initializing vertices: $O(n)$.
Pregel: $\#\text{messages}/\#\text{machines}$, i.e. $O\left(\frac{m}{k}\right)$. 
## Updating the residual graph

**Algorithm 1** Updating the residual graph \( R_G \)

Each key value pair is of the form \( ((i,j) : c) \) **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
## Updating the residual graph

### Shuffle size

Map operation emits at most 2 values per edge: $O(m)$.

### Runtime

Reduce sums at most 2 values for each edge along the path. But since no a priori knowledge of path: $O\left(\frac{m}{k}\right)$. 
## Communication cost

<table>
<thead>
<tr>
<th>Step</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>Broadcast</td>
<td>$O(nk)$</td>
</tr>
<tr>
<td>Residual graph update</td>
<td>$O(m)$</td>
</tr>
</tbody>
</table>

**Table:** Communication cost
### Runtime

<table>
<thead>
<tr>
<th>Step</th>
<th>Sequential</th>
<th>Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>$O(m)$</td>
<td>$O(m/k)$</td>
</tr>
<tr>
<td>Path building</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Broadcast</td>
<td>0</td>
<td>$O(n \log(k))$</td>
</tr>
<tr>
<td>Residual graph update</td>
<td>$O(m)$</td>
<td>$O(m/k)$</td>
</tr>
<tr>
<td>Flow update</td>
<td>$O(m)$</td>
<td>$O(m/k)$</td>
</tr>
</tbody>
</table>

**Table:** Runtime
Comparison with sequential algorithm

**Number of iterations**

Algorithm terminates after \( \min(c, m(n - 1)) \) iterations where \( c \) is the max-flow. For large graphs usually \( c \ll m(n - 1) \)

**Sequential algorithm**

- Runtime: \( \mathcal{O}(cm) \)

**Distributed algorithm**

- Runtime: \( \mathcal{O}(cm/k) + \mathcal{O}(cn \log k) \)
- Communication cost: \( \mathcal{O}(cm) + \mathcal{O}(cnk) \)
Some experimental results
Conclusion

- Problem scales on $m$ ($n$ has to fit on a single machine)
- Runtime optimal: $O(cm) \rightarrow O\left(\frac{cm}{k}\right)$
- Communication cost potentially high, but not for vast majority of applications
- With optimal $k = m/n$. Runtime: $O(cn \log(m/n))$. Communication cost: $O(cm)$
- Largest graph tested: half a million edges