Graph Algorithms

Counting Triangles
Transitive Closure
Counting Triangles

Bounds on Numbers of Triangles
Heavy Hitters
An Optimal Algorithm
Why Care?

1. Density of triangles measures maturity of a community.
   - As communities age, their members tend to connect.
2. The algorithm is actually an example of a recent and powerful theory of optimal join computation.
We need to represent a graph by data structures that let us do two things efficiently:

1. Given nodes u and v, determine whether there exists an edge between them in $O(1)$ time.
2. Find the edges out of a node in time proportional to the number of those edges.

Question for thought: What data structures would you recommend?
Let the graph have N nodes and M edges.

- $N \leq M \leq N^2$.

**One approach**: Consider all N-choose-3 sets of nodes, and see if there are edges connecting all 3.
- An $O(N^3)$ algorithm.

**Another approach**: consider all edges $e$ and all nodes $u$ and see if both ends of $e$ have edges to $u$.
- An $O(MN)$ algorithm.
  - Therefore never worse than the first approach.
To find a better algorithm, we need to use the concept of a *heavy hitter* – a node with degree at least $\sqrt{M}$.

**Note**: there can be no more than $2\sqrt{M}$ heavy hitters, or the sum of the degrees of all nodes exceeds $2M$.

- Impossible because each edge contributes exactly 2 to the sum of degrees.

- A *heavy-hitter triangle* is one whose three nodes are all heavy hitters.
Finding Heavy-Hitter Triangles

- First, find the heavy hitters.
  - Determine the degrees of all nodes.
  - Takes time $O(M)$, assuming you can find the incident edges for a node in time proportional to the number of such edges.
- Consider all triples of heavy hitters and see if there are edges between each pair of the three.
  - Takes time $O(M^{1.5})$, since there is a limit of $2\sqrt{M}$ on the number of heavy hitters.
Finding Other Triangles

- At least one node is not a heavy hitter.
- Consider each edge $e$.
  
  - If both ends are heavy hitters, ignore.
  
  - Otherwise, let end node $u$ not be a heavy hitter.
  
  - For each of the at most $\sqrt{M}$ nodes $v$ connected to $u$, see whether $v$ is connected to the other end of $e$.

- Takes time $O(M^{1.5})$.
  
  - $M$ edges, and at most $\sqrt{M}$ work with each.
Both parts take $O(M^{1.5})$ time and together find any triangle in the graph.

For any $N$ and $M$, you can find a graph with $N$ nodes, $M$ edges, and $\Omega(M^{1.5})$ triangles, so no algorithm can do significantly better.

- **Hint**: consider a complete graph with $\sqrt{M}$ nodes, plus other isolated nodes.

- Note that $M^{1.5}$ can never be greater than the running times of the two obvious algorithms with which we began: $N^3$ and $MN$. 
Parallelization

- Needs a constant number of MapReduce rounds, independent of N or M.
  1. Count degrees of each node.
  2. Filter edges with two heavy-hitter ends.
  3. 1 or 2 rounds to join only the heavy-hitter edges.
  4. Join the non-heavy-hitter edges with all edges at a non-heavy end.
  5. Then join the result of (4) with all edges to see if a triangle is completed.
Transitive Closure

Classical Approaches
Arc + Path => Path
Path + Path => Path
“Smart” Transitive Closure
Strongly Connected Components
Different algorithms for the same problem can be parallelized to different degrees.

The same activity can (sometimes) be performed for each node in parallel.

A relational join or similar step can be performed in one round of MapReduce.

**Parameters:** $N = \# \text{ nodes}$, $M = \# \text{ edges}$, $D = \text{ diameter}$. 

---

**Issues Regarding Parallelism**
The Setting

- A directed graph of $N$ nodes and $M$ arcs.
- Arcs are represented by a relation $\text{Arc}(u, v)$ meaning there is an arc from node $u$ to node $v$.
- Goal is to compute the transitive closure of $\text{Arc}$, which is the relation $\text{Path}(u, v)$, meaning that there is a path of length 1 or more from $u$ to $v$.
- **Bad news**: TC takes (serial) time $O(NM)$ in the worst case.
- **Good news**: But you can parallelize it heavily.
Why Transitive Closure?

- Important in its own right.
  - Finding structure of the Web, e.g., strongly connected “central” region.
  - Finding connections: “was money ever transferred, directly or indirectly, from the West-Side Mob to the Stanford Chess Club?”
  - Ancestry: “is Jeff Ullman a descendant of Genghis Khan?”
- Every linear recursion (only one recursive call) can be expressed as a transitive closure plus nonrecursive stuff to translate to and from TC.
Classical Methods for TC

Warshall’s Algorithm
Depth-First Search
Breadth-First Search
1. Path := Arc;
2. FOR each node u, Path(v,w) += Path(v,u) AND Path(u,w); /*u is called the pivot */

- Running time $O(N^3)$ independent of M or D.
- Can parallelize the pivot step for each u (next slide).
- But the pivot steps must be executed sequentially, so N rounds of MapReduce are needed.
A pivot on u is essentially a join of the Path relation with itself, restricted so the join value is always u.

- Path(v,w) += Path(v,u) AND Path(u,w).

But (ick!) every tuple has the same value (u) for the join attribute.

- Standard MapReduce join will bottleneck, since all Path facts wind up at the same reducer (the one for key u).
Skew Joins

- This problem, where one or more values of the join attribute are “heavy hitters” is called *skew*.
- It limits the amount of parallelism, unless you do something clever.
- But there is a cost: in MapReduce terms, you communicate each Path fact from its mapper to many reducers.
  - As communication is often the bottleneck, you have to be clever how you parallelize when there is a heavy hitter.
The trick: Given Path(v,u) and Path(u,w) facts:

1. Divide the values of v into k equal-sized groups.
2. Divide the values of w into k equal-sized groups.
   - Can be the same groups, since v and w range over all nodes.
3. Create a key (reducer) for each pair of groups, one for v and one for w.
4. Send Path(v,u) to the k reducers for key (g,h), where g is the group of v, and h is any group for w.
5. Send Path(u,w) to the k reducers for key (g,h), where h is the group of w and g is any group for v.

k times the communication, but $k^2$ parallelism
Mapping Path Facts to Reducers

$k = 3$

Path($v, u$) group 1
Path($u, w$) group 1
Path($u, w$) group 2
Path($u, w$) group 3
Path($v, u$) group 2
Path($v, u$) group 3

Notice:
every Path($v, u$) meets every Path($u, w$) at exactly one reducer.
Depth-First Search

- *Depth-first search* from each node.
- $O(NM)$ running time.
- Can parallelize by starting at each node in parallel.
- But depth-first search is not easily parallelizable.
- Thus, the equivalent of $M$ rounds of MapReduce needed, independent of $N$ and $D$. 
Breadth-First Search

- Same as depth-first, but search breadth-first from each node.
- Search from each node can be done in parallel.
- But each search takes only D MapReduce rounds, not M, provided you can perform the breadth-first search in parallel from each node you visit.
- Similar in performance (if implemented carefully) to “linear TC,” which we will discuss next.
Linear Transitive Closure

- Large-scale TC can be expressed as the iterated join of relations.
- Simplest case is where we
  1. Initialize Path(U,V) = Arc(U,V).
  2. Join an arc with a path to get a longer path, as:
     Path(U,V) += \text{PROJECT}_{UV}(\text{Arc}(U,W) \ JOIN \ Path(W,V))
     or alternatively
     Path(U,V) += \text{PROJECT}_{UV}(\text{Path}(U,W) \ JOIN \ Arc(W,V))
     - Repeat (2) until convergence (requires D iterations).
Join-project, as used here is really the composition of relations.

Shorthand: we’ll use $R(A,B) \circ S(B,C)$ for $\text{PROJECT}_{AC}(R(A,B) \text{ JOIN } S(B,C))$.

MapReduce implementation of composition is the same as for the join, except:

1. You exclude the key $b$ from the tuple $(a,b,c)$ generated in the Reduce phase.

2. You need to follow it by a second MapReduce job that eliminates duplicate $(a,c)$ tuples from the result.
Joining Path with Arc repeatedly redoes a lot of work.

Once I have combined Arc\((a,b)\) with Path\((b,c)\) in one round, there is no reason to do so in subsequent rounds.

- I already know Path\((a,c)\).

At each round, use only those Path facts that were discovered on the previous round.
Path = ∅;
NewPath = Arc;
while (NewPath != ∅) {
    Path += NewPath;
    NewPath(U,V) =
        Arc(U,W) o NewPath(W,V));
    NewPath -= Path;
}
**Example: Seminaive TC**

<table>
<thead>
<tr>
<th>Arc</th>
<th>U</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path</th>
<th>NewPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial:</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>12, 13, 23, 24</td>
</tr>
<tr>
<td>Path += NewPath</td>
<td>12, 13, 23, 24</td>
</tr>
<tr>
<td></td>
<td>12, 13, 23, 24</td>
</tr>
<tr>
<td>Compute NewPath</td>
<td>12, 13, 23, 24</td>
</tr>
<tr>
<td></td>
<td>13, 14</td>
</tr>
<tr>
<td>Subtract Path</td>
<td>12, 13, 23, 24</td>
</tr>
<tr>
<td></td>
<td>14</td>
</tr>
<tr>
<td>Path += NewPath</td>
<td>12, 13, 14, 23, 24</td>
</tr>
<tr>
<td></td>
<td>14</td>
</tr>
<tr>
<td>Compute NewPath</td>
<td>12, 13, 14, 23, 24</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

Done
Each Path fact is used in only one round.
In that round, Path(b,c) is paired with each Arc(a,b).
There can be $N^2$ Path facts.
But the average Path fact is composed with $M/N$ Arc facts.
- To be precise, Path(b,c) is matched with a number of arcs equal to the in-degree of node b.
Thus, the total work, if implemented correctly, is $O(MN)$. 
Each round of seminaive TC requires two MapReduce jobs.

- One to join, the other to eliminate duplicates.

Number of rounds needed equals the diameter.

- More parallelizable than classical methods (or equivalent to breadth-first search) when D is small.
If you have a graph with large diameter \( D \), you do not want to run the Seminaive TC algorithm for \( D \) rounds.

- **Why?** Successive MapReduce jobs are inherently serial.

Better approach: *recursive doubling* = compute 
\[
\text{Path}(U,V) + \text{Path}(U,W) \circ \text{Path}(W,V)
\]
for \( \log_2(D) \) number of rounds.

- After \( r \) rounds, you have all paths of length \( \leq 2^r \).
- Seminaive works for nonlinear as well as linear.
Nonlinear Seminaive Details

Path = ∅;
NewPath = Arc;
while (NewPath != ∅) {
    Path += NewPath;
    NewPath(U,V) =
        Path(U,W) δ NewPath(W,V);
    NewPath -= Path;
}

Note: in general, seminaive evaluation requires
the “new” tuples to be available for each use of
a relation, so we would need the union with another
term NewPath(U,W) δ Path(W,V). However, in this case
it can be proved that this one term is enough.
Each Path fact is in NewPath only once.
There can be $N^2$ Path facts.
When $(a,b)$ is in NewPath, it can be joined with $N$ other Path facts.
  - Those of the form Path($x,a$).
Thus, total computation is $O(N^3)$.
  - Looks worse than the $O(MN)$ we derived for linear TC.
Good news: You generate the same Path facts as for linear TC, but in fewer rounds, often a lot fewer.

Bad news: you generate the same fact in many different ways, compared with linear.

Neither method can avoid the fact that if there are many different paths from u to v, you will discover each of those paths, even though one would be enough.

But nonlinear discovers the same exact path many times.
Example: Linear TC Arc + Path = Path
Nonlinear TC Constructs Path + Path = Path in Many Ways
(Valduriez-Boral, Ioannides) Construct a path from two paths:

1. The first has a length that is a power of 2.
2. The second is no longer than the first.
Example: Smart TC
The trick is to keep two path relations, P and Q.

After the i-th round:

- P(U,V) contains all those pairs (u,v) such that the shortest path from u to v has length less than $2^i$.
- Q(U,V) contains all those pairs (u,v) such that the shortest path from u to v has length exactly $2^i$.

For the next round:

- Compute $P += Q \circ P$.
  - Paths of length less than $2^{i+1}$.
- Compute $Q = (Q \circ Q) – P$.
  - P here is the new value of P; gives you shortest paths of length exactly $2^{i+1}$.
### Summary of TC Options

<table>
<thead>
<tr>
<th>Method</th>
<th>Total (Serial) Computation</th>
<th>Parallel Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warshall</td>
<td>$O(N^3)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>Depth-First Search</td>
<td>$O(NM)$</td>
<td>$O(M)$</td>
</tr>
<tr>
<td>Breadth-First Search</td>
<td>$O(NM)$</td>
<td>$O(D)$</td>
</tr>
<tr>
<td>Linear + Seminaive</td>
<td>$O(NM)$</td>
<td>$O(D)$</td>
</tr>
<tr>
<td>Nonlinear + Seminaive</td>
<td>$O(N^3)$</td>
<td>$O(\log D)$</td>
</tr>
<tr>
<td>Smart</td>
<td>$O(N^3)$</td>
<td>$O(\log D)$</td>
</tr>
</tbody>
</table>

Seems odd. But in the worst case, almost all shortest paths can have a length that is a power of 2, so there is no guarantee of improvement for Smart.
Graphs With Large Cycles

- In a sense, acyclic graphs are the hardest TC cases.
- If there are large *strongly connected components* (SCC’s) = sets of nodes with a path from any member of the set to any other, you can simplify TC.
- **Example:** The Web has a large SCC and other acyclic structures (see Sect. 5.1.3).
  - The big SCC and other SCC’s made it much easier to discover the structure of the Web.
Pick a node $u$ at random.

Do a breadth-first search to find all nodes reachable from $u$.

- Parallelizable in at most $D$ rounds.

Imagine the arcs reversed and do another breadth-first search in the reverse graph.

The intersection of these two sets is the SCC containing $u$.

- With luck, that will be a big set.

Collapse the SCC to a single node and repeat.
Instead of just asking whether a path from node u to node v exists, we can attach values to arcs and extend those values to paths.

**Example:** value is the “length” of an arc or path.

- Concatenate paths by taking the sum.
  - $\text{Path}(u,v, x+y) = \text{Arc}(u,w, x) \circ \text{Path}(w,v, y)$.
- Combine two paths from u to v by taking the minimum.

**Similar example:** value is cost of transportation.