CS167: Reading in Algorithms
Counting Triangles*

Tim Roughgarden†

March 31, 2014

1 Social Networks and Their Properties

In these notes we discuss the earlier sections of a paper of Suri and Vassilvitskii, with the great title "Counting Triangles and the Curse of the Last Reducer" [2]. It is a good example of an applied paper with interesting algorithmic content.

Consider an unweighted, undirected graph $G = (V, E)$. G could be anything, but the authors are motivated by graphs derived from social networks. For example, G could be the Facebook graph, where vertices represent people, and the edges represent (bilateral) friendship links. Or G could be derived from a naturally directed social network, such at the Twitter graph (with an arc (u, v) if u follows v), by ignoring the directions of the links.

Having an application domain in mind, like social network analysis, can suggest extra properties of the data which guide the decision of what algorithm to use for a computational problem. This point often differentiates "pure" algorithms papers and "applied" algorithms papers. In the former, there is usually a premium on "general purpose" algorithms that work pretty well on all inputs (your undergrad algorithms class likely had this flavor). In the latter, one often wants to tailor the algorithm somewhat to the application domain of interest, to obtain results superior to that of the general-purpose algorithm. We'll see many more examples of this later in the course.

The relevant properties of social networks for today's discussion are:

1. They are big. Big enough that one needs an algorithm that run in close to linear time. Possibly so big that it doesn't even fit into the main memory of a single commodity computer, in which case parallel algorithms are also important.

2. They are sparse. Meaning the number of edges m is linear in the number of vertices n. Equivalently, the average degree of a vertex is constant. For example, in the Facebook graph, the average degree is in the low hundreds, while n is over 1 billion.

†Department of Computer Science, Stanford University, 462 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

3. The degree distribution is skewed. That is, while the average degree of a vertex is constant, there are a significant number of vertices — the "heavy tail" — that have a very large degree. As a first cut, you might want to keep in mind a graph that has $\approx \sqrt{n}$ nodes with degree $\approx \sqrt{n}$ (this is consistent with $m = O(n)$).

2 Clustering Coefficients and Counting Triangles

The very first thing one should understand about a research paper is:

What is the problem studied?

The paper under discussion tackles two closely related problems.

1. Given a graph $G = (V, E)$, how many triangles does it have? Here a "triangle" is a set of three vertices that are mutually adjacent in G.

2. Given a graph $G = (V, E)$, for every $v \in V$, how many triangles in G include vertex v?

The algorithms we'll discuss effectively solve both of these problems at the same time. Note that a solution to the second problem immediately yields a solution to the first problem: just add up the n counts and divide by 3 (since each triangle is counted exactly once for each of the 3 vertices it contains).

Another thing one should try to understand about a research paper is:

Why should anyone care about this problem?

There are many possible answers to this question. Sometimes the interest comes directly from an application, sometimes the interest is theoretical (e.g., to clarify the power of a specific algorithmic technique). Both applied and theoretical motivations can be important. Some research papers fail to answer this question convincingly. Sometimes you will disagree with the authors about their motivation — that's OK, it's still good to know why other people regard something as interesting.

The paper under discussion [2] reviews the convincing argument that triangle counting is important in social network analysis. Specifically, define the *clustering coefficient* of a vertex v as the fraction of its pairs of neighbors that are themselves connected by an edge. That is, as

$$\frac{\#\text{ of neighbors } u, w \text{ of } v \text{ with } (u, w) \in E}{\binom{\deg(v)}{2}}, \tag{1}$$

where $\deg(v)$ denotes the degree of a vertex (i.e., the number of incident edges). For example, in Figure 1, the clustering coefficient of the vertex v is 3/10.

Observe that the problem of computing clustering coefficients reduces easily to computing all of the numerators in (1), since the denominators are trivial to compute. Note that this is precisely the second triangle-counting problem above. That is, if computing clustering coefficients is a well-motivated problem, then so is triangle counting.

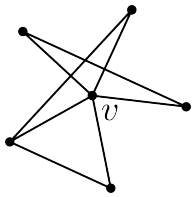

Figure 1: Vertex with clustering coefficient 3/10.

The paper [2] reviews two reasons why social network analysts care about clustering coefficients; more discussion can be found in the book [1]. The first reason is that high clustering coefficients signal "tightly knit communities" — a group of people such that most pairs with a mutual friend are themselves friends. Such communities are expected to have interesting properties, like unusually high degrees of "trust" — there is a strong disincentive to doing anything wrong to someone you are connected to, since lots of your other friends are likely to hear about it. The second reason is that a low clustering coefficient can signal a "structural hole" — a vertex that is well connected to different communities that are not otherwise connected to each other. Such a vertex is in a potentially advantageous situation — for example, to combine two different skill sets to produce innovations, or at least to transfer ideas from one community to another.

3 Triangle Counting Algorithms

3.1 The Obvious Algorithm: Enumerating over Vertex Triples

Whenever you're faced with a new problem, a useful exercise is to ask: (i) what is the "obvious" solution?; (ii) is there a need to improve over this obvious solution? For triangle counting, the obvious algorithm is brute-force search: enumerate over all $\Theta(n^3)$ triples of distinct vertices, and keep track of how many of these triples are triangles. Assuming a graph representation that can answer "edge queries" in constant time — i.e., given a pair u, v of vertices, is (u, v) an edge? — this algorithm runs in $\Theta(n^3)$ time, where n is the number of vertices of the graph. Either of the two triangle counting algorithms above can be solved in this way. This obvious algorithm would be fine for graphs with only a few hundred vertices; for bigger graphs, we need a better algorithm.

Can we expect to do better than a running time of $\Theta(n^3)$? The answer depends on the graph. Some graphs, like a clique (a.k.a. a complete graph), have $\Theta(n^3)$ triangles. Any algorithm that counts triangles one-by-one — like all the algorithms discussed today — is doomed to run in $\Omega(n^3)$ time on such a graph. The brute-force algorithm above, however, runs in $\Theta(n^3)$ time on *every* graph, even those with no triangles at all. So we can definitely hope to do better than the obvious algorithm on many graphs.

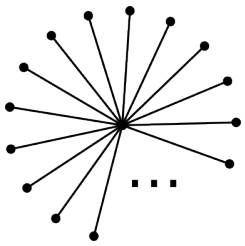

Figure 2: Star graph on n vertices. A naive algorithm will waste $\Theta(n^2)$ time at the center vertex.

3.2 A Better Algorithm: Enumerating over Neighbor Pairs

A smarter approach is to enumerate over all two-hop paths, or "wedges," instead of over vertex triples. That is, we only bother to check vertex triples in which we already know two edges are present. Precisely, the proposal is to solve the second triangle-counting problem as follows:

1. For each vertex $v \in V$:
    (a) For each pair $u, w \in N(v)$ of distinct neighbors of v:
        i. If u, v, w form a triangle, increment a running count of triangles that include vertex v.

With a graph data structure that supports constant-time edge queries, the amount of work done at vertex v is $\Theta(\deg(v)^2)$. Thus, the total amount of work is

$$\Theta\left(\sum_{v \in V} \deg(v)^2\right). \tag{2}$$

As promised, this bound (2) can be $\Theta(n^3)$, for example if every vertex has degree linear in n. In a graph in which every vertex has constant degree, however, the algorithm has linear $O(n)$ running time, which is great.

In social networks, we would like a still better triangle-counting algorithm. Recall that such networks have a skewed degree distribution — the average degree might be constant, but there is a "heavy tail" of vertices with very high degree. For an extreme example, consider a star graph (Figure 2). There are only $n-1$ edges, and 0 triangles, yet the running time of the algorithm above is $\Theta(n^2)$ because of all the work performed at the high-degree center vertex.[1]

[1] To understand the paper's title, "The Curse of the Last Reducer," imagine the obvious parallel implementation of the algorithm above, with a different machine (or "reducer") responsible for the neighbor-pair enumeration of a different vertex v. If one vertex has much higher degree than the rest, then the corresponding machine will finish its processing long after everybody else.

3.3 Better Still: Delegating Low-Degree Vertices

To motivate the next optimization, suppose we're interested in the first triangle-counting algorithm — counting the overall number of triangles in the graph. The algorithm in Section 3.2 counts each triangle exactly three times — once per vertex in the triangle. An obvious optimization is to only count each triangle once. It might seem like this would only save us a factor of 3, but we get a much bigger savings if we're clever about the choice of the vertex responsible for counting a given triangle. The key idea is that only the *lowest-degree* vertex of a triangle is responsible for counting it. Precisely, the new algorithm is:

1. For each vertex $v \in V$:

   (a) For each pair $u, w \in N(v)$ of distinct neighbors of v that both have higher degree than v:[2]

      i. If u, v, w form a triangle, increment a running count of the triangles of the graph.[3]

The star graph suggests the motivation for delegating the counting of a triangle to the lowest-degree vertex. Note that our new algorithm does essentially no work on the star graph — no enumeration is done at the center node (there are no neighbors with a higher degree) or at the spokes (there are no neighbor pairs).

What should be clear is that this algorithm is at least as good as the previous one — it enumerates over only fewer neighbor pairs.[4] The next theorem gives a sense in which the new algorithm's running time is qualitatively better than that of the algorithm in Section 3.2.

Theorem 3.1 *Assuming constant-time edge queries, the worst-case case running time of the algorithm above is $O(m^{3/2})$, where m is the number of edges.*

First, a sanity check: we argued in Section 3.1 that every algorithm based on enumeration, like the algorithm above, has running time $\Omega(n^3)$ on graphs with $\Omega(n^3)$ triangles, like a clique on n vertices. A clique has $m = \Theta(n^2)$ edges and $\Theta(n^3) = \Theta(m^{3/2})$ triangles, so this is consistent with Theorem 3.1. Second, to compare with the algorithm in Section 3.2, consider the star graph (Figure 2). The algorithm from Section 3.2 requires $\Theta(n^2) = \Theta(m^2)$ time on this graph, so it does not meet the guarantee of Theorem 3.1.

Proof of Theorem 3.1: Call a vertex v of the graph G *big* if its degree $\deg(v)$ is at least $\sqrt{m}$, and *small* otherwise. (Recall m is the number of edges.) Observe that there are at most $2\sqrt{m}$ big vertices: otherwise, the sum of the degrees of the graph would be more than $\sqrt{m} \cdot 2\sqrt{m} = 2m$, which cannot be true. Thus, all but a relatively small number of vertices have relatively small degrees.

[2] Ties between vertices with equal degree are broken in some consistent way, for example alphabetically by the "name" of a vertex.

[3] To solve the second triangle-counting problem, v would increment running counters for u, v, and w.

[4] The degrees of all vertices can be computed in $O(m + n)$ time in a preprocessing step.

The first motivation for defining this small vs. big dichotomy is that we already know a good bound for the amount of work done by small vertices. Recall that even in the unoptimized algorithm of Section 3.2, the work done at vertex v is $O(\deg(v)^2)$; this is still valid for the only better algorithm of this section. Thus, if S denotes the set of small vertices, the total work done by the small vertices is

$$O\left(\sum_{v \in S} \deg(v)^2\right). \tag{3}$$

How big could the expression in (3) be? We have two things going for us: first, $\deg(v) \leq \sqrt{m}$ for every (small) $v \in S$; second, since the sum of all degrees is exactly $2m$, $\sum_{v \in S} \deg(v) \leq 2m$. The way to maximize the expression in (3) subject to these two constraints is to make the individual terms as big as possible — this involves having $2\sqrt{m}$ vertices with degree $\sqrt{m}$ each. We leave the formal proof as an exercise; it follows from the convexity of the function $f(x) = x^2$. With the worst-case choice of the $\deg(v)$'s, the expression in (3) is $\Theta(m^{3/2})$.

It remains to bound the work done by the big vertices. All work done by a big vertex v corresponds to neighbor pairs u, w where u, w have even larger degree (and hence are also big). Thus, the total work done by all big vertices is at most a constant times the number of triples u, v, w of big vertices. Recalling that there are at most $2\sqrt{m}$ big vertices, there are at most $8m^{3/2} = O(m^{3/2})$ such triples.

Combining the bounds for the work done by small and by big vertices, the total work done by the algorithm is $O(m^{3/2})$, as claimed. ■

References

[1] David Easley and Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[2] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 607–614, 2011.