Graph Twiddling in a MapReduce World

The easily distributed sorting primitives that constitute MapReduce jobs have shown great value in processing large data volumes. If useful graph operations can be decomposed into MapReduce steps, the power of the cloud might be brought to large graph problems as well.

Relationship collections are well expressed as graphs, which provide natural human interpretation and simple mechanical analysis. As collections for study have increased in size, the volume of associated graphs has stressed practical analysis means and driven development of methods that scale well.

Although scaling solutions usually involve improvements in algorithmic complexity, at some point accommodating the graph in memory becomes impractical and prohibitively expensive. Reasonable responses to such a challenge include distributed computing, stream computing, or processing on a single computer with the graph resident on disk.

Distributed processing on a "cloud"—a large collection of commodity computers, each with its own disk, connected through a network—has enjoyed much recent attention. Atop the hardware is an infrastructure that supports data and task distribution and robust component-failure handling. Google's use of cloud computing, which employs the company's proprietary infrastructure,[1] and the subsequent open source Hadoop cloud computing infrastructure (http://hadoop.apache.org) have largely generated cloud computing's attention. Both of these environments provide data-processing capability by hosting so-called MapReduce jobs, which do their work by sequencing through data stored on disk. The technique increases scale by having a large number of independent (but

loosely synchronized) computers running their own instantiations of the MapReduce job components on their data partition.

MapReduce processes are interesting beyond the cloud as well. Having factored a problem in terms of MapReduce primitives, those primitives are also useful for computing in a streaming environment or on a single computer equipped with a large disk. However, despite the strong interest in MapReduce, few researchers have published work (even in informal Web settings) on MapReduce algorithms. In particular, almost no descriptions of graph algorithms appear in the literature, with the exception of a simplified PageRank calculation and a naive implementation of finding distances from a specified node.[2]

This article begins an investigation into the feasibility of decomposing useful graph operations into a series of MapReduce processes. Such a decomposition could enable implementing the graph algorithms on a cloud, in a streaming environment, or on a single computer.

For the most part, my MapReduce approaches don't implement existing graph algorithms in MapReduce; rather, they discard the usual algorithms and find procedures that produce the same

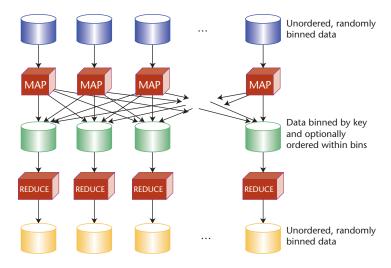JONATHAN COHEN
US National Security Agency

Figure 1. A MapReduce job. The input file consists of records distributed across bins, usually in what may be regarded as an arbitrary fashion. Each mapper reads from a bin and emits records that are binned according to their keys to form an intermediate file. The system then supplies each intermediate file bin to a reduce process that produces records whose storage again can be regarded as random. This distributed output file can then serve as the input to a subsequent job.

outcome. Like me, others might find that the process of factoring a solution into a sequence of sorting operations (rather than graph traversals, say) is something of an impediment at the outset, but becomes instructive and even freeing. An added benefit is that algorithms based on sorting easily admit hypergraph extensions.

Before proceeding, those seeking a reference for graph definitions and results might wish to consult a compendium such as Jonathan Gross and Jay Yellen's work.[3]

The MapReduce Construct

The terms "map" and "reduce" come from Lisp's operations of the same names. In practice, the reduce here can differ significantly from Lisp's in that the output needn't be smaller than the input and might, in fact, be larger.

The Process

The map and reduce operations are general and have a simple interface. Each receives a sequence of records, and each usually produces records in response. A record consists of a key and a value.

Input records presented to the mapper by its caller have no guaranteed order or relationship to one another. The mapper's job is to create some number of records (perhaps none) in response to each input record. Records presented to the re-

ducer by its caller are binned by key, such that all records with a given key are presented as a package to the reducer. The reducer then examines the packages sequentially through an iterator. The programmer also has the option of specifying an order for the packaged records.

Hadoop doesn't employ mappers and reducers independently; they're part of MapReduce jobs in which each job specifies one mapper and one reducer. Figure 1 outlines a MapReduce job's function. A job operates on an input file (or more than one) distributed across bins and produces an output file also distributed across bins. (Distribution of the file contents across bins for both the input and output files can be taken as arbitrary in most cases.) The system feeds input-file bins to the job's mapper instances and partitions the mapper instances' outputs globally by key into bins, producing an intermediate file. It then feeds each of these intermediate bins to a reducer instance; the reducers then produce the job's output file. A complete algorithm might cascade such jobs to implement a more complex process.

From this description, it's apparent that the map operation does what's expected: it transforms each record, although it might produce any number of transformed records from one input. In particular, the map seeks to key its outputs so that the system places in the same bin the records that should come together in the reduce phase. To perform some of the reduction early, thereby lessening the need to store and transport records, the programmer can also specify a combiner that operates on the mapper's output while the mapper is running.

The reduce process operates on a bin's contents, and gets its name from the idea that it creates a small output, such as a count, that characterizes its relatively larger input. Although Google papers suggest that a reduce operation's result is usually small (such as a count), the graph MapReduce processes that I describe in this article usually produce results whose size is on par with the input's size, belying the name "reduce."

As Figure 1 suggests, the programmer can specify that the system sort the bins' contents according to any provided criterion. This customization occasionally can be useful. The programmer can also specify partitioning methods to more coarsely or more finely bin the map results.

Although every job must specify both a mapper and a reducer, either of them can be the identity, or an empty stub that creates output records equal to the input records. In particular, an algorithm frequently calls for input categorization or ordering for many successive stages, suggesting a re-

ducer sequence without the need for intermediate mapping; identity mappers fill in to satisfy architectural needs.

Environmental Assumptions

Although a user can employ MapReduce on a single machine, MapReduce frameworks are designed to support operation on a cloud of computers. The infrastructure then implements a single MapReduce job as parallel mapper and reducer instances running concurrently on different computers. A simplification that MapReduce brings to parallel computing is that the only synchronization takes place when creating and accessing files. The downside is that shared state, beyond the files, is limited.

Hadoop, implemented in Java, runs each mapper or reducer instance in an independent virtual machine on each computer. Consequently, the "static" fields defined in each Java class aren't static across instances. Hadoop supports limited distributed (read-only) tables and common scratches, but the algorithms I describe here don't use them.

What the algorithms do use is Hadoop's facility for passing parameters to the MapReduce job environment and for each mapper or reducer instance to contribute to counts accumulated across all mapper and reducer instances. More specifically, when mapper and reducer instances are created, they're initialized with a copy of the job input parameters.

The assumptions include that the data is too voluminous to be held in memory, and, conversely, that the contents of a single bin can fit in memory if necessary, provided that bins aren't permitted to grow without bound. In particular, the assumption is that a bin describing edges adjacent to a single vertex can fit into memory.

Graph Algorithms

Let's begin with a simple example that illustrates and serves as a component of other algorithms.

An Example: Augmenting Edges with Degrees

Figure 2 summarizes the desired end result for this example. An input file comprises records that each hold an edge. This process will augment the records with vertex-degree information, creating a new file. Creating the desired output requires three logical passes through the data: one map and two reduces. Because of the need to pair maps with reduces, the process requires two MapReduce jobs: two reduce steps and two map steps, one of which is the identity map.

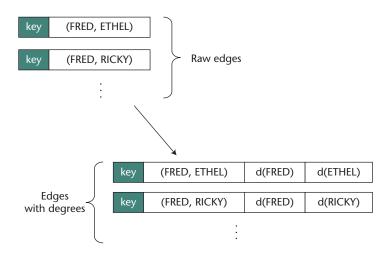Figure 3a summarizes the first MapReduce job. For each edge record the mapper reads, the map

Figure 2. Records for representing vertex degrees. The input file comprises records whose values are edges. The output file has edges augmented with degree information. The keys for these records are unimportant at this point.

emits two records, one keyed under each of the vertices that form the edge. This process will create bins corresponding to vertices such that each bin will hold records for every edge adjacent to its associated vertex. The reduce phase works on each such bin in turn. Having read the bin's contents, the reducer knows the vertex degree, which is equal to the number of records in the bin. The reducer can now emit output records (one for each record read) that are augmented with the degree of the bin's vertex. This is only half of the degree information; another reducer, or another call to the same reducer, produces the other half. The reducer keys output records by the edge so that the two halves of each edge's vertex information can come together in the next phase.

Figure 3b summarizes the second MapReduce job, which employs an identity mapper so the system bins each record by its key—namely, the edge it represents. Each bin then collects the partial-degree information, one record for each vertex in the edge. The reducer combines these records to produce a single record for the edge containing the desired degree information.

Simplifying the Graph

For many algorithms, the starting point is simplifying the graph—that is, removing loops and removing edges that duplicate other edges. Reducing multiple edges between a pair of vertices to a single representative edge can include weighting the representative edge for downstream algorithms that might regard multiple edges as an

Map 1

Reduce 1

(a)

Map 2 (Identity)
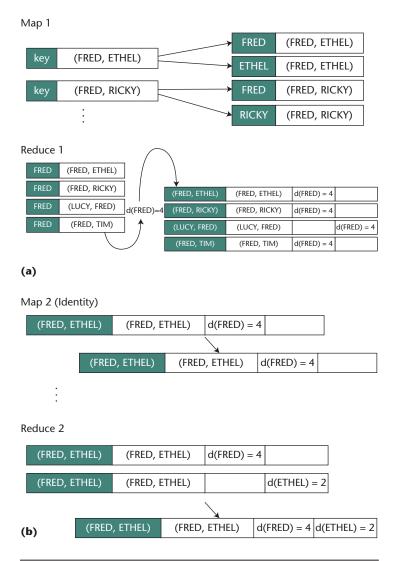

Reduce 2

(b)

Figure 3. Augmenting edges with degrees. (a) In the first MapReduce job, for each input record, the map creates two output records, one keyed under each vertex in the edge. The reduce takes all edges mapped to a single vertex ("Fred" here), counts them to obtain the degree, and emits a record for each input record, each keyed under the edge it represents. (b) In the second MapReduce job, the identity mapper preserves the records unchanged, so the records are binned by the edges they represent. The reducer combines the partial-degree information to produce a complete record, which it exports.

indicator of connection strength.

One can simplify the graph as a directed graph or as an undirected graph. If the graph is undirected, the edge (A, B) duplicates (B, A), and a single representative edge will replace the pair. If the graph is directed, the process should preserve each of these edges.

Simplifying the graph is achievable in a single MapReduce job. The mapper removes loops and bins the edges by membership. The map process-ing on each edge involves several steps. First, to remove loops, the mapper drops all but the first occurrence of a vertex in the (ordered) member list. If the resulting membership list has fewer than two members, the mapper ignores the edge and emits no records. Otherwise, if the mapper removes members, it creates a new representative edge to stand in for the original. Representative edges act as other edges in subsequent processing but are derived from original source edges either by combining edges or by editing them (that is, removing edge members). Representative edges also contain references to their source edges for mapping results back to the source graph.

Next, the mapper generates a hash to represent the edge membership. Used for binning, the hash guarantees that the system places any two edges with the same membership in the same bin. The hash might bin edges with distinct membership together, too, which I'll discuss in a moment. For treating the edge as undirected, the hash operates on membership that's first ordered in some consistent fashion. For treating the edge as directed, the hash takes members in their specified order.

Finally, the reducer records unique edges and drops duplicates. The reducer takes the first edge in the bin, and, using a hash table, recognizes and removes duplicates (but totals the weights), making a single representative edge for the starting edge and those that match it. Having completed this bin portion, the reducer repeats the process to address the rest of the bin's contents, in case edges of different contents happened to hash to the same bin.

Enumerating Triangles

Triangles, or 3-cycles, can be the basis for analyzing graphs and subgraphs. In particular, enumerating triangles constitutes most of the work in identifying the dense subgraphs called *trusses* (see the "Finding Trusses" section for more information). MapReduce offers a good framework for locating triangles.

Enumerating triangles is essentially a two-step approach: enumerate open triads (pairs of edges of the form {(A, B), (B, C)}) and recognize when an edge closes those triads to form triangles. This sounds like a tautology, but it's useful. Note that it isn't necessary to enumerate all open triads to locate all triangles; only one per triangle is needed.

Suppose I have an ordering of vertices (a lexicographic ordering of the names would work, for example). Further suppose I record every edge under its low-order member. Then I'm guaranteed that every triangle will have exactly one vertex that

receives two of its edges. This vertex is the apex of an open triad comprising the two edges that I mapped to it. So, to find triangles, I can choose a vertex ordering, bin all edges under their minimum vertex, and test each pair of edges recorded in each bin to see if that pair (forming an open triad) is closed by a third edge.

The one possible problem with this approach is the quadratic explosion that could result by exhausting the pairs of edges recorded in a bin. To avoid this problem, I can make a judicious choice of vertex ordering: degree. I choose to record each edge under its low-degree member. With this choice, a high-degree vertex will have few edges in its bin, so none of the bins will become large. Thus, the quadratic scaling won't be an issue in most graphs. Certainly, it's possible to construct graphs for which this won't work, but most naturally occurring graphs won't present a problem.

This approach to enumerating triangles applied to the example in Figure 4 starts with simplifying the graph, as I described in the previous section, and augmenting the edge records with vertex valence. After preparation, two MapReduce jobs, shown in Figures 5 and 6, identify the triangles. The process begins by mapping each edge to its low-degree vertex, as Figure 5a illustrates. If the two vertices have the same valence, the mapper breaks the tie by employing a consistent method such as using the order of the vertex names.

Figure 5b shows the first reduce. Each bin it operates on is labeled with a vertex and contains edges adjacent to that vertex. The reducer's job is to emit a record for each pair of edges in a bin: one for every open triad whose apex is at that vertex. The reducer keys output records under the pair's outside vertices so that the subsequent mapper will bin the records with edges closing the triads, should they exist. Note that the mapper only recorded edges under the incident vertex with the lowest valence to minimize bin size, resulting in fewer pairs.

Although I've done it here for illustration, map 1 isn't required to emit a record when a vertex's degree is only 1. Such a vertex will never have a pair produced in the following reduce phase, nor will it appear in a triangle.

Figure 6a shows the second map, which has two input files: the output file from reduce 1 and the degree-augmented edge file. Its job is to combine those records and change the edge records' keys so they're keyed by the vertices that the edges join. The vertices' ordering in the keys is consistent (lexicographic ordering, say) with their ordering in the triad records.
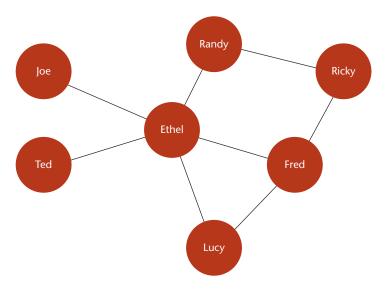

Figure 4. Example graph for finding triangles. Triangles are extracted from this graph by using the MapReduce operations shown in Figures 5 and 6, after simplification and degree augmentation.

Figure 6b shows the second (and final) reduce. Each bin corresponds with a vertex pair. Each bin might contain direct edges joining those vertices and triads joining those vertices. The existence of both will produce a triangle. In particular, a bin will contain at most one edge record and any number of triad records. If a bin contains an edge record and n triad records, then the reducer will recognize n triangles. In the case in Figure 6b, the reducer found only one triangle: the one the (ETHEL, FRED) bin emitted.

Enumerating Rectangles

The job of enumerating rectangles (4-cycles) is similar to that of enumerating triangles. Here, the approach is to find two open triads connecting the same pair of vertices; their combination is a rectangle.

Suppose the vertices have an ordering. In the case of a triangle, there was only one way to order the vertices, up to reflection and rotation, and this guaranteed detection of a triangle by recording edges under the adjacent vertex of minimum order. Rectangles are more interesting. Consider the possibilities for four vertices labeled 1, 2, 3, and 4, occupying the corners of a rectangle in the graph. The labels can be permuted 4! = 24 ways, but the symmetry group on four elements has $|S_4| = 8$. This yields 24/8 = 3 distinct cases, as Figure 7 illustrates.

Slightly different methods detect the three relative orderings, but each involves finding a pair of open triads that join the same pair of vertices. In

Map 1

key	(FRED, ETHEL)	d(FRED) = 3	d(ETHEL) = 5	→	FRED	(FRED, ETHEL)
key	(ETHEL, LUCY)	d(ETHEL) = 5	d(LUCY) = 2	→	LUCY	(ETHEL, LUCY)
key	(LUCY, FRED)	d(LUCY) = 2	d(FRED) = 3	→	LUCY	(LUCY, FRED)
key	(JOE, ETHEL)	d(JOE) = 1	d(ETHEL) = 5	→	JOE	(JOE, ETHEL)
key	(TED, ETHEL)	d(TED) = 1	d(ETHEL) = 5	→	TED	(TED, ETHEL)
key	(RICKY, FRED)	d(RICKY) = 2	d(FRED) = 3	→	RICKY	(RICKY, FRED)
key	(ETHEL, RANDY)	d(ETHEL) = 5	d(RANDY) = 2	→	RANDY	(ETHEL, RANDY)
key	(RANDY, RICKY)	d(RANDY) = 2	d(RICKY) = 2	→	RANDY	(RANDY, RICKY)

(a)

Reduce 1

Joe

| JOE | (JOE, ETHEL) | → ∅ |

Ricky

| RICKY | (RICKY, FRED) | → ∅ |

Ted

| TED | (TED, ETHEL) | → ∅ |

Fred

| FRED | (FRED, ETHEL) | → ∅ |

Lucy

| LUCY | (ETHEL, LUCY) |
| LUCY | (LUCY, FRED) |

→ | (ETHEL, RICKY) | (ETHEL, RANDY), (RANDY, RICKY) |

Randy

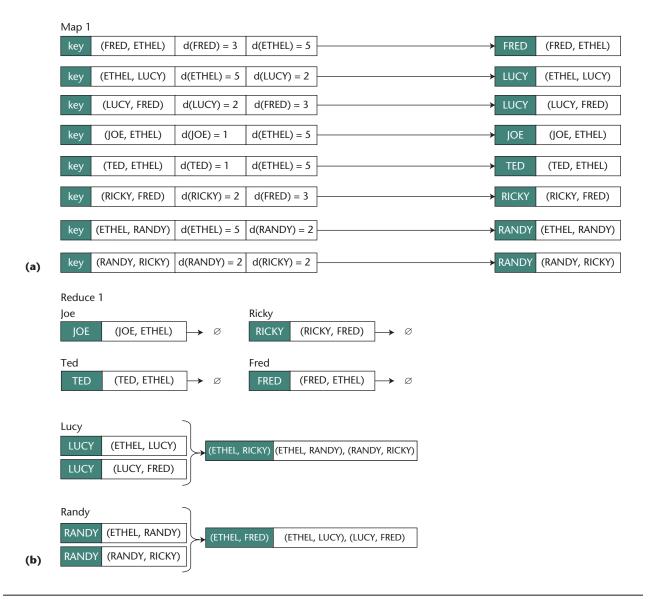| RANDY | (ETHEL, RANDY) |
| RANDY | (RANDY, RICKY) |

→ | (ETHEL, FRED) | (ETHEL, LUCY), (LUCY, FRED) |

(b)

Figure 5. First map and reduce for enumerating triangles. (a) In the first map operation for enumerating triangles, the mapper records each edge under the vertex with the lowest degree. The incoming records' key doesn't matter. (b) In the first reduce, each bin is labeled with a vertex and holds edges that connect that vertex to another vertex of higher (or equal) degree. The reducer emits all pairs of entries in that bin. In this case, only two bins have an output, each emitting one record.

Figure 7, color indicates the decomposition. In Figure 7a, the method locates two triads (each shown with a blue apex) by finding a vertex of lower order than its neighbors, as with detecting the triad in a triangle, although two are required here. To find such a triad, a mapper records each edge under its incident vertex of lowest order and a reducer looks for pairs recorded to the same vertex. This type of triad is a "low" triad.

In Figure 7b, one triad (shown in blue) again has an apex whose order is lower than its neighbors. But the other triad has an apex (green) whose order is between its neighbors. To find such a triad, a mapper records every edge under both of the incident vertices and a reducer looks for a pair of edges binned to that vertex: one low and one high. This type of triad is a "mixed" triad.

Finally, Figure 7c comprises two mixed triads, again shown in green. This case is also decomposable into low and high triads in which the apex has an order higher than its neighbors. Because this method will use degree for ordering, which I'll discuss in a moment, high-degree vertices could make processing explode quadratically, so high-apex triads are not used.
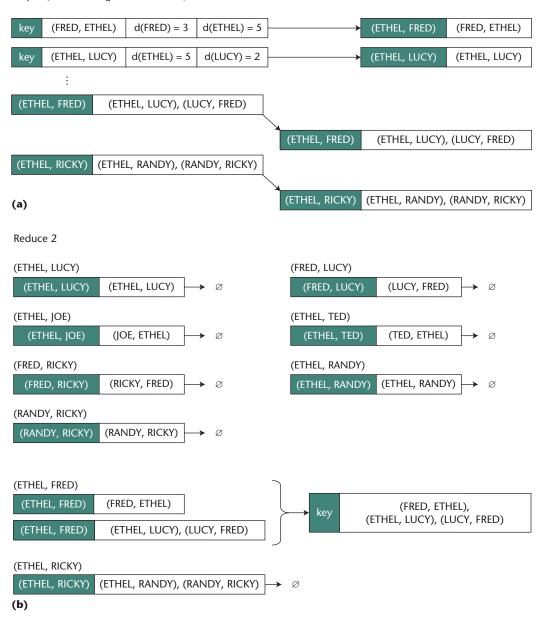
As in the case with triangle detection, the map-

Map 2 (combine edge and triad files)

| key | (FRED, ETHEL) | d(FRED) = 3 | d(ETHEL) = 5 | ⟶ | (ETHEL, FRED) | (FRED, ETHEL) |

| key | (ETHEL, LUCY) | d(ETHEL) = 5 | d(LUCY) = 2 | ⟶ | (ETHEL, LUCY) | (ETHEL, LUCY) |

⋮

| (ETHEL, FRED) | (ETHEL, LUCY), (LUCY, FRED) |

| (ETHEL, FRED) | (ETHEL, LUCY), (LUCY, FRED) |

| (ETHEL, RICKY) | (ETHEL, RANDY), (RANDY, RICKY) |

| (ETHEL, RICKY) | (ETHEL, RANDY), (RANDY, RICKY) |

(a)

Reduce 2

(ETHEL, LUCY)
| (ETHEL, LUCY) | (ETHEL, LUCY) | ⟶ ∅

(FRED, LUCY)
| (FRED, LUCY) | (LUCY, FRED) | ⟶ ∅

(ETHEL, JOE)
| (ETHEL, JOE) | (JOE, ETHEL) | ⟶ ∅

(ETHEL, TED)
| (ETHEL, TED) | (TED, ETHEL) | ⟶ ∅

(FRED, RICKY)
| (FRED, RICKY) | (RICKY, FRED) | ⟶ ∅

(ETHEL, RANDY)
| (ETHEL, RANDY) | (ETHEL, RANDY) | ⟶ ∅

(RANDY, RICKY)
| (RANDY, RICKY) | (RANDY, RICKY) | ⟶ ∅

(ETHEL, FRED)
| (ETHEL, FRED) | (FRED, ETHEL) |
| (ETHEL, FRED) | (ETHEL, LUCY), (LUCY, FRED) |

⟶ | key | (FRED, ETHEL), (ETHEL, LUCY), (LUCY, FRED) |

(ETHEL, RICKY)
| (ETHEL, RICKY) | (ETHEL, RANDY), (RANDY, RICKY) | ⟶ ∅

(b)

Figure 6. Second map and reduce for enumerating triangles. (a) The second map for enumerating triangles brings together the edge and open triad records. In the process, it rekeys the edge records so that both record types are binned under the vertices they connect. (b) In the second reduce, each bin contains at most one edge record and some number of triad records (perhaps none). For every combination of edge record and triad record in a bin, the reduce emits a triangle record. The output key isn't significant.

per orders the vertices by degree, deciding ties consistently using a method such as label comparison. This process makes enumerating low triads inexpensive and enumerating mixed triads practical. In particular, let the degree of vertex v be decomposed as $d(v) = d_L(v) + d_H(v)$, where $d_L(v)$ and $d_H(v)$ are the number of edges incident to v in which v is the low- and high-degree vertex,

respectively. Then v is the apex for $O(d_L^2(v))$ low triads and $O(d_L(v) d_H(v))$ mixed triads.

So, a MapReduce procedure for finding all rectangles in a graph is as follows:

1. Bin every edge by both its high and low vertex, marking each output record as high or low, producing a binned edge file.
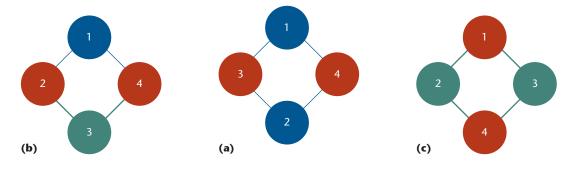
Figure 7. Three rectangle orderings. Up to rotation and reflection, the only ways in which the relative orderings of vertices in a rectangle can occur are as shown. The color of the triad's apex and legs indicates the type: (a) and (b) a low-order triad has a blue apex, (c) a mixed-order triad has a green apex.
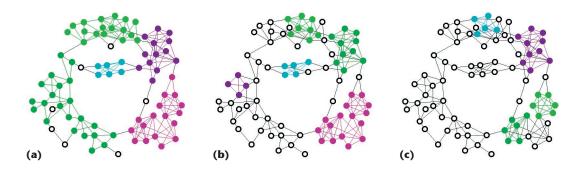

Figure 8. Trusses of a graph. Each truss has a randomly assigned color: (a) 3-trusses, (b) 4-trusses, and (c) 5-trusses. Vertices and edges not in trusses are black; such vertices are also hollow.

2. Go through each bin in the binned edge file, exporting every pair of distinct low records in the bin and every pair of a low record with a high record in the bin to produce a triad file. Each record in the triad file is binned by the pair of vertices the triad connects, that pair ordered lexicographically.
3. Go through each bin in the triad file bin, exporting a rectangle for every triad pair in the bin.

Steps 1 and 2 constitute a single MapReduce job. Step 3 is a second MapReduce job with an identity map. Of course, before these steps, one might want to simplify the graph and must have the vertex valences, perhaps by augmenting the edge records with degree information.

Finding Trusses

Trusses are subgraphs of high connectivity, suitable for recognizing clusters of tight interaction in social networks. They're a relaxation of cliques and capture cliques' intent without their many shortcomings. Cliques are computationally in-tractable. They're unlikely to be found in naturally occurring graphs, particularly when only sampled information is available, and intersect in ways that make their interpretation difficult. I discuss trusses, cliques, and more classical social network constructs in prior work.[4]

Specifically, a k-truss is a relaxation of a k-member clique and is a nontrivial, single-component maximal subgraph, such that every edge is contained in at least $k - 2$ triangles in the subgraph. The use of "nontrivial" here is meant to exclude a subgraph that consists only of a single vertex. A clique of k vertices is an example of a k-truss. Figure 8 shows the 3-, 4-, and 5-trusses of a graph.

It shouldn't come as a surprise that the algorithm to locate trusses is based on the method for finding triangles. Complicating the operation is the requirement that the triangles that support the truss edges must themselves be in the truss. Removing edges with insufficient support might cause other edges to lose their support, and so on.

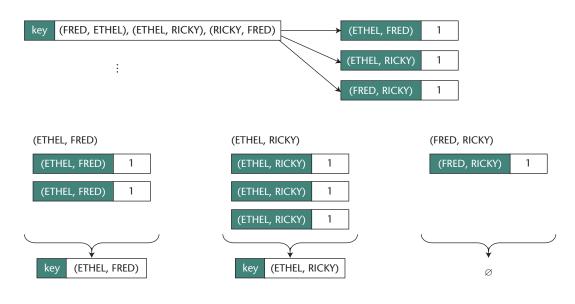The steps, after simplifying the graph are as follows:

Figure 9. Passing edges with sufficient support for trusses. Reading a triangle file, the mapper emits a record for each edge involved in each triangle. The reduce passes edges that occur in a sufficient number of triangles. The records shown here are only a portion of the map and reduce data. (Records in this illustration are unrelated to the graph in Figure 4.)

1. Augment the edges with vertex valences.
2. Enumerate triangles.
3. For each edge, record the number of triangles containing that edge.
4. Keep only the edges with sufficient support.
5. If step 4 dropped any edges, return to step 1.
6. Find the remaining graph's components; each is a truss.

Steps 3 and 4 are combined into a single MapReduce job, as Figure 9 shows. The figure illustrates the case of finding 4-trusses, in which each edge must be in at least two triangles. In the figure's example, the map takes a triangle record and emits one record for each of the edges in the triangle. In the reduce stage, each bin corresponds to an edge. The reduce emits a record only if the number of records in the bin reaches the threshold of support (two, in this case). In the example, one of the edges is in two triangles; one is in three; and another is in only one and fails to pass.

Barycentric Clustering

Researchers often partition the graph into clusters to transform the problem of studying a graph into the problem of studying each cluster subgraph separately. Such a transformation is useful if the resulting partition keeps corelevant information together and divides information that isn't.

Barycentric clustering[5] scales well and identifies tightly connected subgraphs. In barycentric
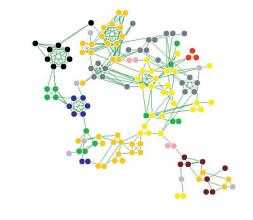

Figure 10. Barycentric clusters. Randomly assigned colors indicate the clusters. Some colors were reused. The layout was not influenced by barycentric clustering.

clustering, vertices are given random initial positions that are then updated by multiplying by a matrix a given number of times. The matrix effectively replaces each vertex position with a weighted sum of its old position and the positions of its neighbors. A "run" is then completed by recording each edge's length, given by the vertex position difference. After conducting multiple runs (a random start followed by iterative updates by matrix multiplication and edge measurement), one cuts the edges with relatively long average

length, resulting in components that are taken as clusters.

More specifically, let the (scalar) positions of the n vertices be $x = (x_1, x_2, \ldots, x_n)^T$. Initially, x comprises standard normal samples, translated and scaled to zero mean and unit L^2 length. Given edge weights $\{w_{ij}\}$, which are zero where no edge exists, and weighted degree $d_i = \sum w_{ij}$, each iteration consists of replacing x with x', so that each vertex position is replaced by a weighted average of its old location and the locations of its neighbors. Specifically, $x' = Mx$, with

$$
M = \begin{pmatrix} 1/(d_1+1) & w_{12}/(d_1+1) & \cdots & w_{1n}/(d_1+1) \\ w_{21}/(d_2+1) & 1/(d_2+1) & \cdots & w_{2n}/(d_2+1) \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1}/(d_n+1) & w_{n2}/(d_n+1) & \cdots & 1/(d_n+1) \end{pmatrix}.
$$

In practice, five iterations of applying M are adequate.[5] Note that this matrix is sparse (provided that the graph is); the amount of work per multiply is proportional to the number of edges in the graph. Multiple random starts are necessary, but they all can be done at the same time. To implement r random starts, the algorithm uses an r-long vector of positions for each vertex. (For the motivation and implementation details of barycentric clustering, see my earlier work.)[5]

The initializing process takes two MapReduce stages and is similar to augmenting the edges with vertex degrees, but it augments them instead with position vectors, one for each vertex:

1. *Map 1.* Bin each edge by its vertices (two records out for each record in).
2. *Reduce 1.* For each bin (vertex), create a position vector of length r; initialize it with standard normal samples, and emit a record for each edge record in the bin, augmenting it with the vertex position vector for that vertex. Key by edge.
3. *Map 2.* (identity).
4. *Reduce 2.* For each bin (edge), combine the edge records augmented with a single vertex's position vector; output one edge record with both vertex position vectors.

Once initialized, the update iterations begin, with five iterations in all. Each iteration consists of two MapReduce jobs:

5. *Map 1.* Bin each (augmented) edge by its vertices (two records out for each record in).
6. *Reduce 1.* This is the essence of the position

update. The bin represents one vertex and holds the records for every edge incident on the vertex, so it has the degree and the (optional) weights of the edges (a row in the M matrix) and the positions of all neighboring vertices. From this, a new position vector is calculated for the vertex based on the updates described above. For each input edge, the reducer writes an output edge record with the new position vector for the vertex and keys it by the edge.

7. *Map 2.* (identity).
8. *Reduce 2.* For each bin (edge), combine the edge records augmented with a single vertex's position vector; output one edge record with all the vertex position vectors.

Note that the second MapReduce job for iteration is the same as the second MapReduce job for initialization.

Having calculated, in parallel, the results of many random starts, it's time to determine which edges are abnormally long relative to their neighbors. Let a_{ij} denote the average length of edge (i,j) taken over all r of the starts. The edge (i,j) is evaluated by comparing a_{ij} to $\bar{a}_{ij}$, a weighted average of a_{ij} over a neighborhood of the graph centered on (i,j). The (spatial) average is intended to measure what's normal for that graph region. Those edges whose value of a_{ij} is large relative to $\bar{a}_{ij}$ are deemed longer than average and can be cut. In general, one could remove edges above a preferred threshold, but I found it sufficient to drop those edges with $a_{ij} > \bar{a}_{ij}$.

I chose the simple 1-out neighborhood as a spatial average, using

$$
\bar{a}_{ij} = \frac{\left(\sum_{k \in N_i} a_{jk}\right) + \left(\sum_{k \in N_j} a_{jk}\right) - a_{ij}}{|N_i| + |N_j| - 1},
$$

where N_i is the set of vertices neighboring vertex i. In other words, $\bar{a}_{ij}$ is a spatial average of the trials (runs) average of edge (i,j) and the edges incident with vertices i and j. The subtraction in the formula avoids double counting of the edge under examination.

Calculating $\{\bar{a}_{ij}\}$ requires two MapReduce jobs, as the method for expressing the average above suggests. The second reduce also provides the filter to drop edges that are too long on average:

9. *Map 1.* Each input record is an edge with the positions of its vertices resulting from the r random starts. Get the edge's length averaged

over the trials (a_{ij} for an edge between i and j) and emit a record for each vertex containing that average, binned by that vertex. The output records are of the form $i \mid ((i,j), a_{ij})$, where the symbol before the divider is the key.

10. *Reduce 1.* A bin contains all the records for one vertex, such as i. Sum the lengths of the records in the bin, obtaining

$$\sum_{k \in N_i} a_{ik}.$$

Emit a record for each edge in the bin of the form

$$(i, j) \left| \left(a_{ij}, |N_i|, \sum_{k \in N_i} a_{ik} \right) \right..$$

11. *Map 2.* (identity).
12. *Reduce 2.* For edge (i, j), its bin holds one record for each of the two vertices in the edge, providing the information to compute $\bar{a}_{ij}$. Compare the length a_{ij} to the neighborhood average $\bar{a}_{ij}$ and emit the edge if $a_{ij} \le \bar{a}_{ij}$.

The edges that the last reducer emits make up a filtered graph, whose components are clusters. The components are found as I'll discuss in the next section.

Figure 10 shows the example of applying barycentric clustering through MapReduce to a graph. The tightly knit subgraphs are clearly identified. I created the graph layout in the figure independently of the clustering algorithm.

The iteration process, being essentially a matrix–vector multiply, is similar to Google's PageRank, whose MapReduce implementation is described in the literature.[2]

Finding Components
Useful in its own right, component finding is also beneficial as a stage in many other operations, such as barycentric clustering and truss finding. Some might think the problem of finding components, or connected subgraphs, is trivial, because it usually involves a single traversal on a graph. But in the MapReduce world, traversals aren't trivial, and component finding is complicated.

The obvious solution. Google's MapReduce lecture series describes a simple MapReduce approach to component finding.[2] This method does the obvious: it starts from a specified seed vertex s, uses a MapReduce job to find those vertices adjacent to s, compiles the updated vertex information in another MapReduce job, then repeats the process, each time using two MapReduce jobs to advance the frontier another hop. If the graph consists of a single component, this approach will take $2\varepsilon(s)$ MapReduce jobs, where $\varepsilon(v)$ is the eccentricity of v. (The eccentricity of a vertex is the maximum distance from that vertex to another vertex in the graph.)

Of course, those lectures deliberately present only simple algorithms. One can do much better than this naive approach, which can take an unnecessarily large number of passes to complete, even when the graph has only one component. If multiple components are present, one must run it with a single seed, see if the vertices are exhausted, and then repeat the process with a new seed if vertices remain.

A collective alternative. Here, I sketch an alternative to the single-seed scheme, in which I build many local solutions that merge into a global one. The idea is to form "zones," each of which comprises a set of vertices known to be in the same component as its seed vertex. Initially, the algorithm constructs one zone for each vertex, having that vertex as its seed and as its sole member. As the algorithm proceeds, the zones merge to form larger zones. When no further expansion is possible, each zone is known to be a component.

Each zone's seed vertex serves as its identification. Throughout, I'll use two files: an *edge file* that describes the graph, each of whose records is an edge, and a *zone file*, whose records, one for each vertex, are of the form (v, z), indicating that vertex v is currently assigned to zone z. The edge file is immutable; the zone file will evolve. Initially, the zone file comprises records whose form is (v, v), because each vertex is in its own seed at the outset.

When compared, zones get their ordering according to their seeds, which themselves have an order. The order might be arbitrary, such as by name. When two zones meet, the lower-order zone absorbs the higher-order one.

The basic idea employs two steps: Step 1 is to use the zone file to transform the vertex-to-vertex adjacencies in the edge file to zone-to-zone adjacencies, dropping resulting records that connect a zone to itself. Step 2 is to use the zone-to-zone adjacencies to find the lowest-order zone adjacent to each current zone and use this information to update the zone file: for each zone with a lower-order neighbor, replace that zone with its lowest-order neighboring zone. This second step assigns the vertices to new and likely fewer zones.

Steps 1 and 2 alternate until the first step produces no records, at which time the result is in the zone file. Step 1 can be implemented using two MapReduce jobs: binning by vertex and then by edge, as Table 1 illustrates. Map 1 merges records from the edge

Table 1. Using zones to find components.

MAP 1*	Input: edge file; key: arbitrary; payload: edge e
	Input: zone file; key: v; payload: pair (v, z)
	If record payload is edge, for each vertex $v \in e$, emit a record with key v and payload e; else emit a record with key v and payload pair (v, z)
REDUCE 1	Input key: vertex t; input payloads consist of one zone pair (v, z), and all edges E incident to vertex v
	Iterate through the bin, extracting the edges present E and the zone z. For each $e \in E$, emit a record with key e and payload z
	Output: EdgeWithOneZone file
*MapReduce 1 implements the first part of step 1: binning by vertex. Each output record connects an edge to a zone.	
MAP 2**	Input: EdgeWithOneZone file; key: e; payload: zone z
	Identity
REDUCE 2	Input key: edge e; each input payload is a zone z
	Iterate through the bin, extracting the set of distinct zones Z. If $|Z| = 1$, skip this bin. Increment an InterzoneEdges counter. Find the minimum zone $z_m \in Z$ such that $z_m \leq z, \forall z \in Z$. For each $z \in Z - \{z_m\}$, emit a record with key z and payload z_m
	Output: InterzoneEdges file
**MapReduce 2 implements the second part of step 1: binning by edge. Output records each connect one zone to a better one. The global InterzoneEdges counter can be examined to see if any vertices changed zones.	
MAP 3†	Input: InterzoneEdges file; key: zone z; payload: new zone z_m. Input: zone file; key: v; payload: pair (v, z)
	If record payload is vertex-zone pair (v, z), emit a record with key z and payload v; else, emit a record with key z and payload z_m
REDUCE 3	Input key: old vertex zone z; input payloads consists of some number of vertices V, and some number of "better" zones Z
	Iterate through the bin, extracting the suggested new zones Z and the vertices V with the old zone z. Find the best new zone for this vertex: $z_b = \min\{t \mid t \in Z \cup z\}$. For each $v \in V$, emit a record with key v and payload pair (v, z_b)
	Output: zone file
†MapReduce 3 implements step 2: binning by old zone. This step takes in the old zone file and produces a new zone file.	

and zone files. The idea is to translate the vertex-to-vertex information in each edge to zone-to-zone information. MapReduce 1 associates edges with zones by producing one record for each vertex in an edge, identifying the zone of that vertex with the edge. MapReduce 2 brings these together by edge, because reduce 1 binned MapReduce 1's records by edge. If distinct zones are joined by the edge, then the "best" one (the one with the lowest value) captures the others. This is indicated by output records that are binned by the zone to be changed, with a payload of the new zone. Note that MapReduce 2 increments a counter, accumulated across all reducers, that tallies the number of interzone edges remaining. The tally is checked after running the job. If it's zero, the procedure terminates.

Step 2 can be implemented in a single MapReduce job, binning by zone. Table 1 shows MapReduce 3, which updates the zone file—that is, it records new zones for the vertices. The job does this by merging the interzone-edges file, which offers better zones for some of the old zones, and the preceding zone file, which held the earlier zone assignments. Each reduce bin holds all of the old zone's vertices and some number (possibly zero) of better zones for the old zone. The reduce finds the best of the new zones and emits records for each of the vertices, assigning them to the new zone.

This simple approach has room for improvement. Rather than iterating these steps to the bitter end, the algorithm can iterate them until the number of interzone edges becomes small enough to complete the calculation through conventional means on a single compute node in memory. A final MapReduce step then translates the results of this zone-to-zone calculation to update the zone file.

As the computation progresses, map 3 might create uneven loading because the increasingly large zones each map to single bins. Monitoring the size of zones and distributing a large zone across L bins, where L is chosen to achieve sufficient balance, can alleviate this potential problem with little expense. Monitoring is done through global counters, which are fed to the next iteration. A zone that gets large

causes each zone-file record to be sent to one of the L bins and each interzone-edges file record to be sent to all of L bins. This improvement makes modifications only to MapReduce 3.

Clearly, MapReduce implementations of useful graph operations exist and can be rather simple. Moreover, although I don't describe them here, one can extend each of the algorithms to hypergraphs with no additional MapReduce steps. This is an important extension because relationships dealing with more than two vertices are of great interest and often not handled well in conventional graph-processing environments.

The MapReduce implementations aren't obvious analogs of standard algorithms; indeed, for the most part, they require a complete rethinking of the problem. I found the return to fundamentals useful, resulting in an improved method for enumerating triangles and an efficient way of enumerating squares that I'll carry to implementations outside of MapReduce.

A common pattern in the algorithms in the "Graph Algorithms" section is as follows:

1. *A map operation*. Go through all the edges, changing some piece of vertex information; key the resulting records by vertex.
2. *A reduce operation*. For each vertex bin, read the edge records and determine the updated state of the vertex; emit this information in partially updated edge records; bin the result by edge.
3. *A reduce operation*. For each edge bin, combine the updates from each of its member vertices to get an edge record with complete updated vertex information.

It's unfortunate, then, that Hadoop forces each reduce to be proceeded by a map, thus wasting a map in implementing the pattern. To be fair, many functions that I could have implemented as map operations, I incorporated as part of the preceding reduce operation, leaving an identity map. Such a factoring is worth reconsidering.

I was able to devise MapReduce approaches to most operations that I sought, though not all— for example, I didn't find an efficient implementation of finding bridges. (A *bridge* is an edge whose removal would cause the graph to divide into separate components.) Although a complete characterization of what's practical and what's not doesn't exist, here's a beginning: we can efficiently implement operations that we can characterize by independent local communication (message passing, percolation, matrix-vector multiplication, and so on); algorithms that require depth-first traversals likely have poor MapReduce analogs.

Having established a small set of graph operations that perform well in a MapReduce framework, and having concluded that other graph operations are probably not appropriate, future work should make a concerted stab at refining the line between these two classes.

Beyond the existence of graph operations' MapReduce implementations, the practicality of cloud-based MapReduce computations on graphs depends most heavily on a concern I didn't address in this article: demands on interprocessor bandwidth. Each MapReduce job has the potential to move every graph record from one processor and its disk to another. The prospect of the entire graph traversing the cloud fabric for each MapReduce job is disturbing. Serious testing of these algorithms on the target hardware is needed before researchers can declare them practical. If modifications exist that can reduce bandwidth requirements, they warrant investigation.

With the existence of MapReduce methods for graph processing now established, we must next address the practical issues of hardware mapping. $\overset{\text{C}}{\underset{\text{SE}}{}}$

References

1. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, vol. 51, no. 1, 2008, pp. 107–112.
2. GoogleDevelopers, "Lecture 5: Parallel Graph Algorithms with MapReduce," 28 Aug. 2007; http://youtube.com/watch?v=BT-piFBP4fE.
3. J.L. Gross and J. Yellen, *Handbook of Graph Theory*, CRC Press, 2004.
4. J.D. Cohen, "Trusses: Cohesive Subgraphs for Social Network Analysis," 2008; http://www2.computer.org/cms/Computer.org/dl/mags/cs/2009/04/extras/msp2009040029s1.pdf.
5. J.D. Cohen, "Barycentric Graph Clustering," 2008; http://www2.computer.org/cms/Computer.org/dl/mags/cs/2009/04/extras/msp2009040029s2.pdf.

Jonathan Cohen *is a research engineer at the US National Security Agency. His research interests include signal processing, visualization, graph algorithms, and information retrieval. Cohen has a PhD in electrical engineering from the University of Maryland. Contact him at jdcohenee@gmail.com.*