MODELING NATURAL GRAPHS QUICKLY WITH STOCHASTIC KRONECKER GRAPHS

NIKHIL DESAI

Why are data scientists so obsessed with graphs? It's because graphs are the best tools we have for modeling the real world. By analyzing the graph representation of a real-world structure, we can glean a variety of insights about it. Graphs that model real-world phenomena are called "natural" graphs, and a great deal of data science focuses on them. However, obtaining natural graphs is hard; it would be nice if we had a way to generate similar-looking graphs without the data-gathering work. Enter the **stochastic Kronecker graph model**: an easy way to generate almost-natural synthetic graphs. This post will give an overview of natural graphs and describe the stochastic Kronecker model of generating graphs.

1. Introduction to natural graphs

1.1. **Background and motivation.** At Lab41, we see graphs everywhere. Much of our work revolves around analyzing and generating natural graphs that have structural properties similar to those found in real-world settings. Such graphs could represent an arrangement of computers in a network, animals in a food chain, or neurons in your brain. Unlike randomly-generated graphs, natural graphs have *meaning*. For example, characteristics of a system modeled by a graph can be deduced by calculating mathematical metrics such as its nodes' *degree* (the number of edges connected to a node in a graph) or the number of triangles formed by its edges.

Working with natural graphs involves a number of challenges:

- **Obtaining natural graphs is hard.** One must painstakingly collect a large dataset of real-world observations and connections, find a suitable way to interpret it as a graph, and then actually convert it into a graph - a process that can be tedious and time-consuming.
- **Datasets for natural graphs are scarce.** There are only a small number of existing datasets representing natural graphs. In fact, at the recent GraphLab workshop, one speaker noted that he was getting tired of every presenter using the same dataset (articles and links between them on Wikipedia) for their analyses!
- **Synthetic graphs miss the mark.** Graphs randomly generated according to standard models (as my colleague Charlie did in his previous post, and others have done using the Erdos-Renyi graph model) tend to look *unnatural*, no matter what parameters we use. We can't just create natural graphs by taking a random number generator and going crazy. Instead, we need to find out what properties make a

graph "natural," and then find a way to effectively and efficiently generate graphs with those properties.

1.2. **Properties of natural graphs.** So, what makes a graph "natural"? While there is no hard-and-fast definition, nearly all natural graphs exhibit two simple properties:

- **Power-law degree distributions.** A very small number of nodes have a very large number of connections (high degree), while a large number of nodes have a very small number of connections (low degree). Mathematically speaking, this means the degree of any vertex in the graph can be interpreted as a random variable that follows a power-law probability distribution.
- **Self-similarity.** In natural graphs, the large-scale connections between parts of the graph reflect the small-scale connections within these different parts. Such a property also appears within fractals, such as the Mandelbrot or Julia sets.

An accurate mechanism for natural graph generation must preserve these properties. As it turns out, the **stochastic Kronecker graph model** does this. It has a few other advantages as well:

- **Parallelism.** The model allows large graphs to be generated at scale via parallel computation.
- **Structural summarization.** The model provides a very succinct, yet accurate, way to "summarize" the structural properties of natural graphs. Two Kronecker graphs generated with the same parameters will produce graphs with matching values for common structural metrics, such as degree distribution, diameter, hop number, scree value, and network value.

The remainder of this blog post will describe the basic Kronecker generation algorithm and how it can be modified to efficiently generate very large graphs via parallel computation, on top of MapReduce and Hadoop.

2. Mathematical formulation

2.1. **Kronecker products.** The core of the Kronecker generation model is a simple matrix operation called the *Kronecker product*. Let $\mathbf{A}$ be a matrix with dimension $m \times n$ and a matrix $\mathbf{B}$ of dimension $m' \times n'$. Let the (i, j)th element of $\mathbf{A}$ be a_{ij}. The Kronecker product of $\mathbf{A}$ and $\mathbf{B}$, which we'll call $\mathbf{A} \otimes \mathbf{B}$, is a $(m \cdot m') \times (n \cdot n')$ block matrix, given by

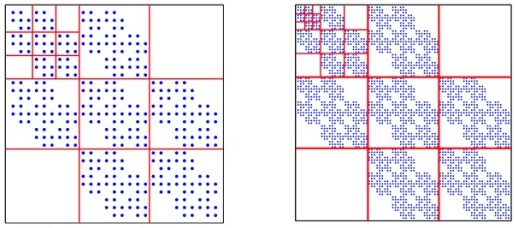
$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1n}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2n}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mn}\mathbf{B} \end{bmatrix}.$$

When we take the Kronecker product of two matrices, we find that it seems to "nest" many copies of the second within the first. While this is interesting on its own, it doesn't seem to have any obvious connection with our problem of generating graphs, so why do we care about it?

As it turns out, graphs and matrices have a far deeper connection than you might think. In fact, *every* graph G can be represented by an *adjacency matrix* $A(G)$ - a matrix whose rows and columns correspond to the nodes of the graph, each cell of which contains a zero if an edge does not exist between the two nodes it corresponds to, and a one if it does. This means that any operation on two matrices can be thought of as an operation on two graphs as well.

With this in mind, we'll define the Kronecker product of two *graphs* to be the graph whose adjacency matrix is the Kronecker product of their adjacency matrices. Mathematically speaking, if G and H are two graphs, then we define $G \otimes H$ so that $A(G \otimes H) = A(G) \otimes A(H)$.

FIGURE 1. Fractal patterns visible in the adjacency matrix of a Kronecker graph. Taken from [1].



(a) K_3 adjacency matrix (27×27) (b) K_4 adjacency matrix (81×81)

2.2. **Kronecker graphs.** As we mentioned above, the Kronecker product nests copies of one matrix within the other. It turns out that the same nesting occurs when we take the Kronecker product of two graphs - the graph $G \otimes H$ looks like the graph G, in which each node has been expanded into a copy of H.

Naturally, we may wonder what occurs when we take the Kronecker product of a graph with itself. We find that each node of the graph will "expand" into a small copy of the graph, and the connections between these copies will mirror the connections between the original nodes of the graph. This is why the Kronecker graph product is so interesting - it allows us to build self-similar graphs!

We can extend this idea further and consider "Kronecker powers" of a graph G - that is, the result of taking the Kronecker product of G with itself a number of times.

3. Algorithms for generating Kronecker graphs

3.1. **Naive algorithm.** The simplest algorithm for generating Kronecker graphs is to use Kronecker powers to generate a stochastic adjacency matrix, and then step through each cell of the matrix, flipping a coin biased by the probability present in that matrix. In more detail, the algorithm is as follows:

(1) We start with an n by n initiator matrix, θ, and the number of iterations k for which we wish to run the algorithm. We compute the k-th Kronecker power of the

matrix θ, giving us a large matrix of probabilities, which we call P. Each cell in this matrix corresponds to an edge between two nodes in the graph; the formula for the value at the (u, v)th cell of P is:

$$\prod_{i=0}^{k-1} \theta \left[\left\lfloor \frac{u}{n^i} \right\rfloor \bmod n, \left\lfloor \frac{v}{n^i} \right\rfloor \bmod n \right].$$

(For convenience, we have assumed the matrix is zero-indexed, as is common in computer science.)

(2) To generate the actual graph, we 1) step through each cell in the matrix, 2) take the probability in the cell, 3) flip a coin biased by that probability, and if the coin "comes up heads," we 4) place the corresponding edge in the graph.

If the initiator matrix is an $n \times n$ square matrix, and we perform k iterations of the Kronecker power operation, the generated matrix will have dimension $N = n^k$. We will need to take a product of k values to obtain each cell of the final matrix, and there will be N^2 cells, so the runtime of this algorithm will be $O(kN^2)$.

This means that if we want to generate a graph with approximately one billion nodes (a reasonable size for a large natural graph) from an initiator matrix of size 2, our runtime expression tells us we should expect to perform approximately $(30)(10^9)^2 = 3.0 \times 10^{19}$ operations. That's 30 *quintillion* operations. This leads us to wonder whether we could do this with fewer operations. Spoiler alert: it's possible.

3.2. **Fast algorithm.** If we switch from a node-oriented approach to an edge-oriented approach, there does exist a faster algorithm for generating a Kronecker graph. Most natural graphs are sparse - $E = O(N)$. Thus, if we can find a way to place each *edge*, one at a time, in the graph, rather than figuring out if a pair of nodes has an edge between them, we can vastly reduce the on-average running time. To do this, we need to figure out how many edges are in the graph, and we need to figure out which nodes are associated with each edge.

It turns out that the expected number of edges in a stochastically generated Kronecker graph is encoded within the initiator matrix itself - it's given by:

$$E = \left(\sum_{i,j} \theta[i, j] \right)^k.$$

In general, this works out to being on the order of the number of nodes.

Next, we need to find a procedure that starts from nothing, and in k iterations picks a new edge in the graph to add. Thankfully, this operation is already staring us in the face - in the formula presented in the previous section. Here it is again:

$$\prod_{i=0}^{k-1} \theta \left[\left\lfloor \frac{u}{n^i} \right\rfloor \bmod n, \left\lfloor \frac{v}{n^i} \right\rfloor \bmod n \right].$$

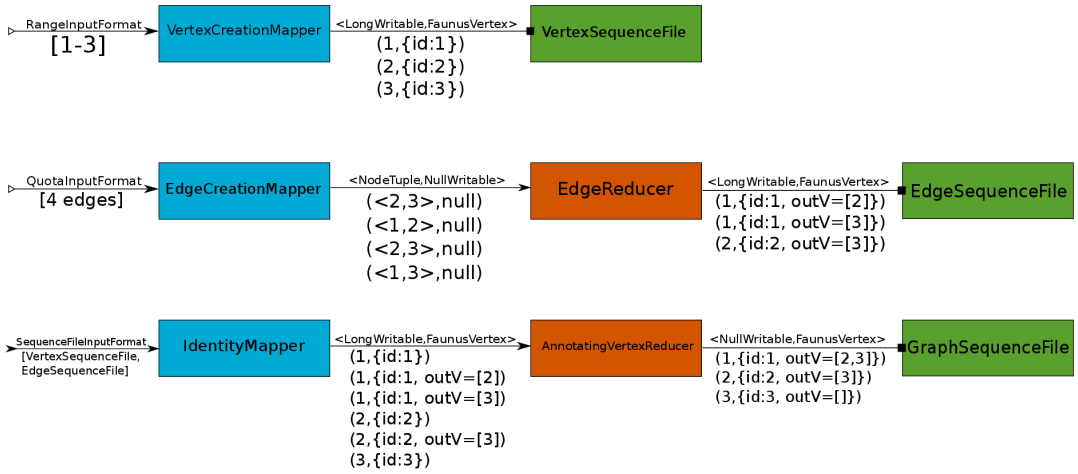
This formula can be understood in a different way - as a "recursive descent" into the adjacency matrix of the graph, picking smaller and smaller blocks of the matrix until we have finally narrowed our choice to a single cell, which we then "color in" to represent that an edge should be placed there.

Thus, to generate a stochastic Kronecker graph, all we need to do is set up a loop which runs E times, generating a new edge in the graph on each pass-through. (If we generate the same edge twice, we ignore it and repeat the pass-through as if nothing happened.) This runs in $O(kE)$ time, which means that for sparse, real-world graphs, it runs in $O(kN)$.

3.3. **Parallel algorithm.** This algorithm allows us to generate every edge in the graph independently of every other edge, allowing us to parallelize the graph's generation. This means we can leverage the power of Hadoop to generate very large graphs.

The only twist is that this method allows for the creation of duplicate edges, and most of the graphs we're interested in don't contain such duplicates. Thus, we need to figure out how to identify and eliminate them. This is hard when generating the graph across multiple machines, because it's very likely the duplicate edges will be generated on separate machines. Fortunately, with a bit of cleverness, we can leverage the nature of MapReduce to do our duplicate checking. Instead of one MapReduce job, we'll have three - one to generate edges and eliminate duplicates, one to generate vertices, and one to combine the two together to form a single graph. This gives us the workflow below.

FIGURE 2. The workflow for Kronecker graph generation. Datatypes appear above the line, sample data below. For convenience, FaunusVertex objects have been represented in JSON and NodeTuple objects by pairs of values between angle brackets.



The pipeline consists of three stages:

(1) The first stage of our pipeline is vertex generation. This is the simplest stage - it is a map-only job, utilizing a custom input format representing a range of vertices to be generated. We use as the key a unique `Long` identifying the vertex, and a `FaunusVertex` object as the value, giving us a ⟨`Long,FaunusVertex`⟩ output sequence file.
(2) The second stage of our pipeline is edge generation. As with vertex generation, it uses a custom input format representing a quota of edges to place into the graph. For each edge in this quota, we run the fast stochastic Kronecker placement algorithm, yielding a tuple of vertex IDs that represents a directed edge in the graph. This tuple is stored as a custom intermediate key type (called a `NodeTuple`), with the value as a `NullWritable`; this allows the shuffling and sorting logic of MapReduce to place identical tuples together, and consequently allows us to easily eliminate duplicate copies of the directed edges before the reduce step. Finally, in our reduce step, we emit a ⟨`Long,FaunusVertex`⟩ tuple. The `FaunusVertex` represents the edge's source vertex and contains a `FaunusEdge` indicating its destination vertex. The `Long` key is the source vertex's ID.
(3) The third and final stage of our pipeline reads in the vertex objects generated by both the edge and vertex creators and combines them, creating a final list of `FaunusVertexes` that represents the graph.

A few details on the pipeline:

- **Faunus.** This pipeline uses the same data types as the Faunus engine for graph analytics. Faunus provides objects representing edges (`FaunusEdge`s) and vertices (`FaunusVertex`es) that can be serialized and utilized by MapReduce jobs but can also serve as a final representation of a graph. Conveniently, `FaunusVertex`es can store the edges coming off them as well, so we do not need to store edges separately from vertices in the final graph - we need only store the list of vertices with edges added to them.
- `SequenceFiles`**.** This pipeline produces `SequenceFiles` (a native MapReduce serialization format) consisting of `FaunusVertex`es to serve as intermediate representations of the graph as we construct it.
- **Annotations.** In the final stage, we annotate the vertices with several property values (a mixture of floating-points and strings) in order to mimic the data we are interested in.

References

[1] Leskovec, Jure, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: an approach to modeling networks. *ArXiv*, arXiv:0812.4905v2