Chapter 27

Big data and scalability

Networks get big. Really big. In this chapter, we discuss some of the challenges you face and solutions you can try when scaling up to massive networks. These range from implementation details to new algorithms and strategies to reduce the burden of such big data.

27.1 Do you *really* have big data?

Before talking about how to scale the network analysis, the very first thing to remember is that chances are *your network is probably not that big* to necessitate the following practices and that you may really want to *avoid using them* unless it is absolutely necessary or the big data infrastructure is already set up for you. Most methods and practices that we will discuss here come with substantial overhead and extra cost, and you want to avoid prematurely jumping into a fancy, complex technology.

The first questions you need to ask are: what kinds of data do we need to address our question? Is our data really that *big*? Can we reduce the data enough so that we don't need "big data" technology? For instance, even if your network has many millions of nodes, you may be able to answer your particular question by carefully *sampling* a small fraction of nodes or by reducing the network with the methods explained in Ch. 10. You may even be able to address your question simply by using a more efficient library or more resource-aware programming language.

Remember: premature optimization is the root of all evil. Try to tackle your problems in the simplest ways possible. Only if that fails should you turn to some of the technologies and methods described here.

27.2 When networks become large

Computers are so powerful nowadays, with so much storage. We shouldn't have to worry about our data becoming too big to work with, right? Yes and no. Although many networks are large, they are often still manageable with powerful computers and large

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

disks. They are not so massive that we need to turn to specialized storage systems and algorithms. On the other hand, there are areas where networks are absolutely massive, forcing us to worry about more efficient storage systems, ways to interact with the network efficiently, and using specialized algorithms for analysis.

In some sense, network data are simpler than other forms of data. Representing a network (Ch. 8) will not be as data-intensive as, say, storing all the videos on YouTube, or all the pictures shared on Instagram. However, if a network becomes dense, the set of links will become quadratic, meaning $M = O(N^2)$, as any node can in principle be linked to any other node. Fortunately, we almost never encounter real, large networks close to that level of density—sparsity saves us. But working with large networks requires data structures that may encounter such density, even if only locally (a fully connected subgraph for example). The storage of one YouTube video will never depend on the contents of other videos.[1]

Examples of large networks. Three problem domains in particular stand out as focusing on massive networks:

The web graph The network of web pages connected by directed hyperlinks. This evolving network is the focus of web search engines from major companies like Google that spend considerable resources to index and re-index, or *crawl* the web. PageRank (Secs. 12.9 and 25.4) was introduced to tame the chaos of the web by giving a centrality measure to rank important web pages. A 2014 analysis of a Common Crawl[2] dataset found the web page graph contains 3.5 billion nodes and 128 billion links [306]. The public web graph is highly dynamic, however, with many pages coming and going, and estimating its size will depend on when and for how long the crawl occurred. A recent data release of the Common Crawl project, for instance, which crawled only from 26 January to 9 February 2023, contained an estimated 3.15 billion web pages and 1.3 billion new URLs, not seen in previous crawls.

Online social networks Major online platforms deal with huge amounts of social connections. For example, researchers at Facebook released a 2015 paper describing a graph storage system capable of handling up to 1 trillion edges [103]. Of course, such a graph will contain far more than a social network, since businesses and other organizations exist on such platforms. Further, nodes in the graph represent individual pieces of content such as posts, comments, and likes. Twitter, a less popular social platform, probably still comprises a graph structure of 10^8 nodes and 10^9 or more edges.

Knowledge graphs Some of the largest sources of network data are knowledge graphs. These networks, or *ontologies*, contain huge numbers of semantic *triples* meant to represent factual statements as links. For example, "Leonardo da Vinci is a painter" is the triple "Leonardo da Vinci" (subject), "IsA" (predicate), "painter"

[1] That said, there are lots of networks related to features on platforms like YouTube, such as the user subscriber lists and comment threads.

[2] https://commoncrawl.org

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

(object). The nodes are subjects and objects while the links can be categorized or arranged into layers based on the predicates. Such representations are meant to enable computational reasoning by enabling network algorithms to work with facts and statements. These can perhaps one day imbue AI systems with commonsense reasoning. As you can imagine, the set of all possible triples is large and very large knowledge graphs have been created. For example, DBpedia,[3] an open knowledge graph extracted from Wikimedia projects such as Wikipedia, contains over 21 billion triples [215]. Meanwhile, proprietary knowledge graphs, such as those created by Facebook or Google, are expected to be at least as large; a 2020 post from Google [450] described their knowledge graph as having "amassed over 500 billion facts about five billion entities" gathered from hundreds of online sources. Working with knowledge graphs needs significant computing infrastructure.

Problems and strategies

Big data can bring us back to the drawing board. We need scalable methods (see below). Often the solutions we have used so far take details for granted that are no longer applicable.

For example, the network can become so large it no longer fits into memory. If we have relied on, say, a Python dictionary to store a network as an adjacency set (Ch. 8), we have to rethink our data store. We can move to a database that works on disk, but this will make working with the network far slower. And we may need to be more efficient with how we represent the network, even to the level of choosing node identifiers more appropriately. We need methods to compress the graph.

Even if we can be space-efficient storing the graph, in principle it can get so big it won't fit on a single computer. We then need a *distributed system* where parts of the graph can be stored on different computers and we can efficiently access those different parts. Given how the network structure pulls together disparate regions of the network, this can get tricky. If many nodes stored on one machine are connected to many nodes on another machine, we are likely to require more data transmission between the machines as we work with the network, leading to wasted bandwidth and even reliability issues when synchronizing changes to the data.

Sometimes a network is so big relative to our compute resources that we have no chance of storing and reanalyzing the data. Or perhaps we could store the network but we don't have time to reanalyze the data—we need to finish the calculation in real-time for a customer-facing feature. (A practical example of a (nearly) real-time calculation is verifying a credit card transaction as non-fraudulent using a large knowledge graph.) Imagine a *stream of data* flying past, where we can only read it once. And maybe we can't even read it completely, only a subset of it. How can we implement algorithms on a graph stream? Even generating a random sample of nodes can be challenging.

Lastly, many network algorithms have high complexity. If we want to compute the diameter of a network, say, we need to run breadth-first search between every pair of nodes. That's $O(N^2)$, which is no longer feasible when we have a billion or more

[3] https://www.dbpedia.org

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

nodes. This challenge demands approximations. Can we estimate the diameter from a sample, lowering the cost but introducing a (hopefully acceptable) degree of error into our estimate? Even simpler calculations than the diameter are challenging: how can we efficiently estimate how many *distinct* nodes are present in a stream of dynamic graph events so large we cannot store the stream in memory? Approximations and local algorithms rule the day when it comes to massive networks.

27.3 What do we mean by "scalability"?

What does it mean for a network analysis algorithm or technique to be *scalable*? Scalability usually means it can handle large networks or "scale up" to that large scale. But whether a network is large-scale or not depends on many factors; there is not a clear dividing line between small-scale and large-scale. One of the major factors is the computer hardware being used. A network of a million nodes is probably considered as medium size today, but 20 or 30 years ago at the dawn of modern network science it was large. Earlier in time, it would be considered absolutely massive.

To skip past the details of hardware means, just like with any algorithmic analysis, we consider the computational complexity of the algorithm either in time (number of calculations) or space (amount of memory or storage). For our purposes, a scalable method must be cheaper than quadratic. An $O(N^2)$ or $O(M^2)$ algorithm will be inaccessible to us. Of course, this too depends on various other factors. An algorithm may have good average complexity but poor worse-case complexity. Likewise, one algorithm may have better scaling than another but worse constant factors, which may make a difference in practice.

In other words, roughly speaking, a network analysis tool is scalable when its complexity is sub-quadratic in the size of the network.[4]

Note that network size includes both nodes and links. An algorithm with complexity $O(N)$ or $O(N \log N)$ is usually scalable; algorithms with complexity $O(N^2)$ or $O(M^2)$ or $O(NM)$ are not.

Sometimes, an algorithm is scalable for some networks and not others. The most common cause is density. An algorithm can be very efficient when the network is sparse but slows down as it becomes dense. We can see this in the complexity because a dense network is defined as one where most edges exist; there are $N(N-1)/2 \sim N^2$ possible edges, so if most exist then $M = O(N^2)$. An algorithm that is scalable, for instance with complexity $O(N+M)$, will now be too costly: $O(N+M) \rightarrow O(N+N^2) = O(N^2)$. Such a method will not scale to a large, dense network.

27.4 Compressing, distributing, and streaming graphs

Here we discuss problems and strategies for operating with very large amounts of graph data. Keep in mind our earlier admonishment that hardware continues to advance and some of these problems can be solved not by complex data engineering or involved algorithms but just by brute-force hardware. A modern multi-core, shared-memory

[4] Of course, anything worse than polynomial is completely out the window when it comes to scalability.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

machine can store graphs with billions of edges in memory. Compress, distribute, stream when you need to, not before: premature optimization is the root of all evil.

27.4.1 Compressing graphs

We have already seen in Ch. 8 how data structures such as adjacency sets can save space. We can store the network as an edgelist $(i, j_1), (i, j_2), \ldots$ but we are repeating the node i for each of i's edges. Instead, store i's neighbors altogether: $(i, j_1, j_2, \ldots), \ldots$

We can take this further. If each neighbor j_s is represented with an identifier, we want to use as few bits as possible to store those identifiers. Compression algorithms on text data, for instance, work by assigning smaller identifiers (fewer bits) to more common words or letters. We can do something similar for networks, giving high-degree nodes shorter identifiers. But networks give us more opportunities: we can exploit the connectivity patterns of the network to save more space.

Consider this subset of a directed adjacency set, written as a table (node 17 has degree 0) [62]:

⋮	⋮
15:	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16:	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17:	
18:	13, 15, 16, 17, 50
⋮	⋮

Notice how often we have a consecutive run $n, n + 1, \ldots$ of IDs? We can encode this information more efficiently by using *gaps* (also called *deltas*):

⋮	⋮
15:	3, 1, 0, 0, 0, 0, 3, 0, 178, 111, 718
16:	1, 0, 0, 4, 0, 0, 290, 0, 0, 2723
17:	
18:	9, 1, 0, 0, 32
⋮	⋮

Here an entry of zero in a neighbor list indicates that the node is one more than the preceding node. If $S(x) = (s_1, \ldots s_k)$ are the neighbors (successors) of node x, the gap representation stores[5] them as $(s_1 - x, s_2 - s_1 - 1, s_3 - s_2 - 1, \ldots, s_k - s_{k-1} - 1)$. In practice, of course, the benefit of gaps will depend on how nodes are identified, and often a reordering of the nodes needs to be computed.

Gaps exploits consecutivity. We can also exploit similarity. Two nodes i and j may have nearly identical adjacency lists, if their neighborhoods strongly overlap. We can

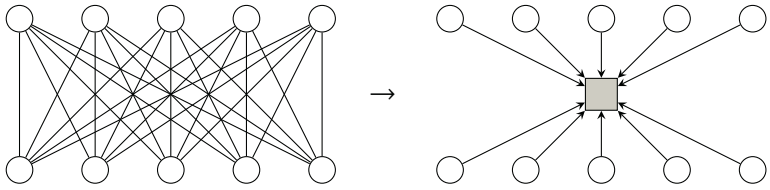
[5] All these gaps will be positive except possibly the first one $(s_1 - x)$. If we are dealing with enough data that we care about this, we probably do not want to allocate space for storing the sign of an ID so we want to avoid negative entries. The solution is to encode the first element $y = s_1 - x$ using the map $\nu(y) = 2y$ if $y \geq 0, 2|y| - 1$ otherwise.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

exploit this by storing $S(i)$ and then, instead of $S(j)$, store a reference from j to i along with a list of differences between $S(j)$ and $S(i)$. It will require computation to find the differences, but the space savings can be substantial.

Many algorithmic questions still need to be addressed to combine these strategies together into a workable, large scale system. See our remarks for references with more details. For brevity, we discuss one more common compression strategy: virtual nodes.

Some network motifs (Sec. 12.6) are highly dense and regular. If they occur frequently, it is not worth storing every instance. Instead, we can remove all the links within the motif and replace it with a node that represents the entire motif, then link the nodes within the motif to this new virtual node. The savings can be substantial for completely dense subgraphs:



For a clique of n nodes, we go from n nodes and $n(n-1)/2$ edges to $n+1$ nodes and n edges. An efficient algorithm called virtual node mining was introduced by Buehrer and Chellapilla [81] to find those nodes.

Virtual nodes are one example of a grouping compression (and summarization) technique [278]. In general, these compression strategies, like all data compression algorithms, have a tradeoff: more space can be saved but at the expense of more computation. For very large networks, it can become costly to find all the redundancy possible, so we often settle for "good enough" compression.

Random data generally compress poorly and the same holds for graphs. Some graphs are inherently more compressible than others. The web graph in particular compresses surprisingly well. In part, this is because of redundancy in how links are set up. Many links on a web page repeat, for example, a navigation bar on every page within a site will lead to a highly redundant set of links. Social networks also show compressibility but generally less so [64].

27.4.2 Distributing and streaming graphs

For exceptionally large networks, it becomes untenable to store them in memory (even on disk) on a single machine. The network needs to be distributed to multiple machines. Computations on the network also need to be distributed. While we have techniques and standards for distributed computation, such as the MapReduce framework [124], they may not be suitable for graph-structured data.

For example, suppose we wish to compute shortest paths on a weighted graph. We can run Dijkstra's method but, because not all of the graph is on a single machine or single disk, the algorithm must be distributed across all the data machines. If we have distributed the graph poorly, such that many edges exist between nodes on different machines, we may find poor performance: the distributed computation will need to pass

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

many messages between those different machines as the shortest paths are computed, and inter-machine communication is far slower than intra-machine communication.

Minimizing cross-talk between data machines is another motivation for graph partitioning (Sec. 25.5). We can use (scalable) methods to distribute subgraphs between our data machines such that few edges exist between machines. Many graph databases have algorithms for this *sharding* or *horizontal distribution* problem, but it remains a difficult feat of data engineering. Often, careful choices of schema for a graph database are needed to ensure queries of the data can run efficiently. One technique for improving efficiency is by incorporating overlap: allow the "boundary" of groups to exist on *multiple* machines. If chosen with care, this can lower significantly the need for inter-machine messaging, at the expense of more storage. Consistency is also an issue here (as it is always a concern with distributed data): if the data shards are changing after they have been distributed, we may run into scenarios where the boundaries duplicated across machines are no longer isomorphic. Errors may creep in.

Distributed computation engines A problem closely associated with distributing the graph is distributing computations on the graph. Frameworks such as MapReduce [124] and its successor implementations (Apache Spark, Beam, etc.) have served this purpose well for traditional relational data stores. But graphs bring distinct challenges as their connectivity patterns demand more from parallel calculations. (It is difficult to apply a divide-and-conquer parallelism strategy to a small-world network, for instance.) Graph-specific distributed computational models have been proposed, including Pregel [289, 103], GraphLab [280, 185, 281], and GraphX [496].

Beyond distributing graph data to multiple machines, we can consider the problem of ***streaming graphs***. A data stream is a read-once sequence of data and we generally have no control over the order we see items from the sequence. The idea being that the data come by in such a high volume or velocity that computationally we are not able to store the data, or at best we only have memory capacity that is sublinear in the length of the data stream. Often we do not even know how long the stream will be.

We can imagine a graph data stream where each item in the stream is an edge in the graph, or perhaps a subgraph of some kind. What calculations can we perform when we are forced to interact with the graph in such a piecemeal manner [300]?

27.4.3 Sampling large streams

Reservoir sampling [476] is an interesting algorithm for sampling uniformly from a large stream of data, so large that you cannot store it in memory and you don't even know how big the stream is. Without knowing the number of observations n it becomes difficult to define the uniform sample probability $1/n$.

We wish to sample k items uniformly at random from a stream of items we observe one at a time. The length of the stream is unknown. We want to guarantee at any given sampling step, if the stream ends, the probability for each item to be sampled is the same. The reservoir sampling algorithm is quite simple:

1. Store the first k elements of the stream in a buffer (the reservoir).

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

2. Every time we look at an element $i > k$, we decide to keep it with probability k/i. If kept, select an item from within the buffer uniformly at random and replace it with the ith item.

Does this really work? We need to show that this sampling strategy leads to a uniform sample of the stream. Specifically, because we sampled k items, we need the probability that element i is sampled to be k/n for all i. We study two cases, $i \leq k$ and $i > k$:

- For $i \leq k$: i has already been selected. What is the probability it is not replaced? The probability that element $j > k$ does not replace i is 1 minus the probability j is selected and i is chosen as the replacement location in the reservoir, or

$$\Pr(j > k \text{ does not replace } i) = 1 - \frac{k}{j}\frac{1}{k} = \frac{j-1}{j}. \tag{27.1}$$

  From this, the probability that *every* $k < j \leq n$ does not replace i is given by

$$\frac{k}{k+1}\frac{k+1}{k+2}\cdots\frac{n-1}{n} = \frac{k}{n}. \tag{27.2}$$

- For $i > k$: now we ask, what is the probability i is selected and then not subsequently replaced? Element i is selected with probability k/i. The probability it is not replaced later is, like before, $\frac{i}{i+1}\frac{i+1}{i+2}\cdots\frac{n-1}{n} = i/n$. Put together, the probability that i is selected and not replaced is k/n.

Both cases show the correct probability, and therefore we know reservoir sampling generates a uniform sample of the data stream. All while not knowing the value of n. Amazing!

One immediate application for our purposes is computing statistics like the assortativity of a graph stream. Use reservoir sampling to generate a sample of k edges, then compute the correlation between the attributes or degrees (assuming we can efficiently compute or estimate the degrees) of their constituent nodes. Note, of course, that sampling edges uniformly at random is not the same as sampling nodes uniformly at random (Chs. 12 and 21). And with further effort, reservoir sampling-like streaming algorithms can be introduced for sampling triangles and other motifs [231, 301]. Indeed, many algorithms have been proposed for graph streams beyond subgraph enumeration, including finding minimum spanning trees, checking for bipartiteness, thinning and sparse cuts, and more [300].

27.5 Approximations and local methods

Graph streaming brings us more broadly to problems and solutions that work either locally or by making approximations. Approximations trade off computational complexity for accuracy, allowing breathtakingly efficient solutions if we are willing for answers to have some (known) degree of error. We first discuss probabilistic counting, a general technique that we will then use for network-specific data structures.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

27.5.1 Probabilistic counting

Suppose we are faced with a huge data stream, far too large for our memory, and we want to count the unique items in the stream. For example, how many *unique* edges are present in a large graph stream of edges. This is called *cardinality estimation* and it is often encountered in big data applications. Of course, we can store all the edges we've seen before in a data structure, but this will consume considerable space and even time. Can we estimate cardinality without using a large in-memory hashing data structure such as a Python dict or resorting to slow out-of-memory storage like a disk? Yes, if we are willing to have some errors in the count.

We discuss HyperLogLog (HLL) [161], one of the best solutions to large-scale cardinality estimation. HyperLogLog (and its ancestors [160, 137]) begin from the following intuition. Suppose each item x in our stream can be mapped to a string of bits using a *hash function* $h(x)$ such that (approximately) all bit strings are equally likely; each bit in the string is equal to zero with probability 1/2. If this is the case and you observe in your stream a bit string that begins with $\rho - 1$ zeros before the first 1, then it stands to reason the cardinality n of the stream is at least 2^ρ. As we encounter items in our stream, we can simply track the largest value of ρ we observe, using very little memory.

Unfortunately, as you can imagine, this solution alone will be extremely error-prone (it also turns out to be biased). We may get unlucky in a small stream and by chance encounter a large ρ, wildly affecting our estimate. To solve this, HLL takes two courses of action. One, it splits the stream into substreams that are averaged over, helping with noise. Two, it uses a harmonic mean which is less susceptible to outliers. The authors of HLL also identify biases and corrective factors in their analysis of the algorithm.

Algorithm 27.1 describes the HyperLogLog algorithm. Each hashed item is placed into a substream based on its first few bits, while cardinality is estimated using the remaining bits. The storage requirements for HLL are extremely modest: when estimating cardinalities $\leq N$, only a collection of "registers," each of which using $O\left(\log_2 \log_2 N\right)$ bits, are needed.

How does this method work? If n is the unknown cardinality, each substream should have cardinality approximately n/m, where m is the number of registers. Then the maximum ρ stored in the register for that substream should be approximately $\log_2(n/m)$ and the harmonic mean (times m) of 2^M should be of the order n/m. Therefore $m^2 Z$ should be of the order n and we have our cardinality estimate. The term α_m corrects a multiplicative bias in $m^2 Z$ that was shown from the analysis by Flajolet et al. [161]. The authors also prove that the relative standard error between the estimated cardinality E and n is $\leq 1.04/\sqrt{m}$.

To illustrate the power of HLL, the authors describe that "using $m = 2048$, hashing on 32 bits, and short bytes of 5 bit length each: cardinalities till values over $N = 10^9$ can be estimated with a typical accuracy of 2% using 1.5 kB (kilobyte) of storage" [161].

With a tool like HLL,[6] we can solve otherwise intractable enumeration questions. Because the counters are so small, we can have very many of them, and we can now, for example, enumerate motifs or shortest paths in a very large network.

[6] Other probabilistic data structures can be used for similar problems, including Bloom filters [58] and MinHashes [76].

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Algorithm 27.1 HyperLogLog cardinality estimation [161].

Input: Data stream (multiset) S, hash function h, $m = 2^b$ with $b \in \mathbb{Z}^+$
Initialize m registers $M[1], \ldots, M[m] \leftarrow -\infty$
for $v \in S$ **do**
    $x \leftarrow h(v)$
    $j \leftarrow 1 + \langle x_1 x_2 \cdots x_b \rangle_2$ ▷ *the binary address determined by the first b bits of x*
    $r \leftarrow \rho(x_{b+1} x_{b+2} \cdots)$ ▷ *$\rho(s)$ is the position of the leftmost 1 in s*
    $M[j] \leftarrow \max(M[j], r)$
$Z \leftarrow \left(\sum_{j=1}^{m} 2^{-M[j]} \right)^{-1}$ ▷ *Harmonic mean*
Return: $E \leftarrow \alpha_m m^2 Z$, where

$$\alpha_m := \left(m \int_0^\infty \left(\log_2 \left(\frac{2+u}{1+u} \right) \right)^m du \right)^{-1} \tag{27.3}$$

27.5.2 Approximate Neighborhood Functions

The *neighborhood function* (NF) counts how many pairs of nodes i, j in a graph are reachable within t hops (i.e., the shortest path distance $\ell_{ij} \leq t$) as a function of t. This function contains a wealth of information about the graph topology (see also: network portraits, Secs. 13.3 and 14.2.2; graph distances and connectedness, Sec. 12.10). From the NF, we can compute quantities like the typical distance between nodes, and the diameter. But computing the NF requires global breadth-first search (usually) calculations, which are expensive and memory-intensive.

A highly space-efficient algorithm [355, 63] to approximate the NF can be constructed by recognizing that the NF can be determined by taking set unions and that probabilistic counting registers (Sec. 27.5.1) can estimate the cardinality of a set union. Specifically, let $B(i, r)$ be the set of all nodes within distance r from node i. This "ball" satisfies $B(i, 0) = \{i\}$ and

$$B(i, r) = \bigcup_{j \in N_i} B(j, r-1). \tag{27.4}$$

We can compute the NF (and generalizations) from these sets. Given two sets A and B, the cardinality of $A \cup B$ can be estimated from HyperLogLog counters for A and B by maximizing register-by register (assuming the counters have the same bit length), i.e., $M_{A \cup B}[i] = \max(M_A[i], M_B[i])$.[7]

These two ingredients combine to make the *approximate* NF (ANF) method. The sets $B(i, r)$ are never stored as they will become massive for large enough networks. Probabilistic counters are used instead to track the cardinalities of their unions, saving

[7] In principle, we could also use this to estimate the set intersection from $|A \cap B| = |A| + |B| - |A \cup B|$ as we have counters for all three. In practice, the errors scale poorly for this strategy and more sophisticated methods are employed [460].

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

considerable space. The ANFs are built by scanning over the adjacency set of the network, applying HLL counters to the unions (Eq. (27.4)). When finished, meaning when the HLL counters have converged (and they must be run to convergence for the estimates to be accurate), we have the information necessary to approximate the NF. The basic HyperANF algorithm is given in Alg. 27.2.

Algorithm 27.2 HyperANF without optimizations [63].

Input: Network G as adjacency sets $\{i : N_i\}$

procedure UNION(M, N)
    for each $i < m$ **do**
        $M[i] \leftarrow \max(M[i], N[i])$

Initialize $c[-]$, an array of N HLL counters
for $v \in G$ **do**
    Add v to $c[v]$
$t \leftarrow 0$
while $c[-]$ has not converged **do**
    $NF(t) \leftarrow \sum_v \text{size}(c[v])$ ▷ *The neighborhood function*
    for $v \in G$ **do**
        $m_v \leftarrow c[v]$
        for $w \in N_v$ **do**
            $m_v \leftarrow$ UNION($c[w], m_v$) ▷ *Estimator from Eq.* (27.4)
        Store (v, m_v)
    Recall $\{(v, m_v) \mid v \in G\}$ and update array $c[-]$
    $t \leftarrow t + 1$

Several further optimizations make the HyperANF method highly efficient (and these are what really distinguish it from the earlier ANF method [355]). For one, the outer loop over G can be parallelized. Second, the HLL union, which is performed many times, can be parallelized using clever bitwise operations [63]. Third, we can track which c's have not changed and avoid maximizing them. This last improvement can be taken even further by allowing nodes to "signal" backwards to predecessors when their counter changes, and we can later skip over nodes whose successors have unmodified counters.

With ANFs we can derive a variety of useful network statistics [355, 63], the most prominent being the "effective" diameter. Computing the cumulative distance distribution $H_G(t) := NF(t)/\max_t NF(t)$, the effective diameter is the distance t such that $H_G(t) < \alpha$ (usually, $\alpha = 0.9$). By modifying Alg. 27.2, we can also devise other statistics related to cut numbers (Sec. 25.5) and similar quantities by tracking subsets of the NF, such as $NF_i(t)$ for individual nodes i or even sets of nodes [355]. Thus we have a way to measure how big the network is, and other properties, even when interacting with it is challenging due to its scale.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

27.5.3 Application: The small world of Facebook

As an application of these technologies, we briefly describe a 2012 study by Backstrom et al. [20] to estimate social distance (small world; Sec. 22.5) in the Facebook social network. They used both graph compression (Sec. 27.4.1) and approximate neighbor functions (Sec. 27.5.2) with HyperLogLog counters (Sec. 27.5.1) and other optimization to make the calculation feasible. With this infrastructure in place, they analyze social networks for individual countries and the entire platform, at the time consisting of over 700 million active users and nearly 70 billion friendship edges. They found the average distance (number of hops) between individuals was 4.74 or 3.74 "degrees of separation," lower than the famous six degrees of separation and lower than the 4.4–5.7 degrees famously found by Milgram [309]. Regional networks showed consistently smaller distances still, which are more directly comparable to Milgram's work which was itself restricted to a single country. Perhaps with richer data to analyze, or perhaps with more venues for communication, our small world is smaller still.

27.5.4 Community detection at scale

Fast community detection with label propagation

Community detection (Secs. 12.7, 23.2, and 25.6) on large networks is challenging. Here we discuss an efficient algorithm using label propagation (LP), proposed by Raghavan et al. [384]. Nearly all the previous community methods we've studied, from modularity maximization to inference of the stochastic block model, are too costly for very large networks. The LP method works by transforming the global community detection problem into a local updating rule, akin to an eigenvector centrality update but without the need for matrix operations.

The LP algorithm is beautifully simple. Suppose each node is given a label denoting which community it belongs to. The labels propagate by a majority rule: each node i looks at every neighbor $j \in N_i$ and adopts the most common label among the neighbors. Initially, every node has a unique label, corresponding to N communities all of size 1. As labels propagate, dense groups will reach consensus on a label, which crowds out uncommon labels and spreads until colliding with the labels of other dense groups, until propagation stops. Updating stops naturally at consensus, when every node has the same label as the majority of its neighbors.

This spreading process can be implemented with synchronous updating steps, where all nodes update their labels at the same time, or with asynchronous updating, where one node updates its label, then another, etc. and later updates can be affected by earlier updates. Raghavan et al. [384] show that asynchronous updating has an advantage because, with synchronous updates, certain network structures (bipartite subgraphs) can cause "blinking" where labels oscillate back and forth instead of converging. Asynchronous updating avoids this, and cycling through nodes in a random order for each update helps speed up mixing.

The LP algorithm is as follows. Let $C_i(t)$ be the label of node i at step t.

1. Initialize the labels, giving each node a unique label: $C_i(0) = i$.
2. Set $t = 1$.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

3. Arrange the nodes in a random order L.
4. For each $i \in L$, chosen in the order of L, let

$$C_i(t) = f\left(C_{i_1}(t), \ldots, C_{i_u}(t), C_{i_{u+1}}(t-1), \ldots, C_{i_k}(t-1)\right), \tag{27.5}$$

   where $i_1, \ldots, i_u \in N_i$ are neighbors of i who have already updated (preceded i's position in L) and $i_{u+1}, \ldots, i_k \in N_i$ are neighbors who have not, and $f(\cdot)$ returns its most frequent argument, breaking ties at random.
5. If every node has the label that is most common among its neighbors, terminate the algorithm and return the $\{C_i\}$. Otherwise, set $t = t + 1$ and continue from 3.

There is one wrinkle after the algorithm converges. While usually rare, it is possible for disjoint groups to independently converge on the same label. This occurs when two or more neighbors of a node adopt its label and then pass that label in different directions, which eventually leads to separated communities adopting that same label. The fix to distinguish different groups with the same label is to find the connected components of the subgraph induced by each label, which can be found by one breadth-first search per component in $O(N + M)$ time.

During each iteration, finding $C_i(t)$ for each node requires looking at the k neighbors of i, at a cost of $O(k_i)$. For all the nodes, this gives a total cost of $\sum_i k_i = O(M)$—each iteration is linear in the number of edges. The algorithm tends to converge very quickly, even when there is no community structure (Raghavan et al. [384] report that typically 95% of nodes reach consensus after five iterations), although the number of iterations needed is not known theoretically. This fast convergence in practice famously gives the algorithm a near-linear complexity.

Local community methods

Another problem of interest in big data is finding a local community only, and not worrying about partitioning the full network. Local methods can do this, finding a single community belonging to a starting node by spreading outward from that node. We discussed local community methods in Sec. 12.7.

27.6 Updating schemes for network statistics

Dynamic updating refers to tracking the evolution of a network summary statistic s as the data are changing [452]. Suppose, for instance, we wish to know the degree assortativity of our very large network. Stopping to compute this will be intractable, it is a global calculation. Instead, we can design our data system to track the necessary components of the statistic, updating them efficiently on any changes to the network (node insertion, edge insertion, node deletion, edge deletion). Then, when needed, we can perform the final calculation to derive s. While there can be a space tradeoff depending on s, often it is minor compared to the time complexity of the global calculation.

We now develop dynamic updating schemes for several important network statistics. Some are trivially straightforward, such as updating the degree. Others require more bookkeeping. We focus on node degree, clustering coefficient, assortativity, and modularity.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Recall the clustering coefficient (Eq. (12.8)):

$$C_i = \begin{cases} \frac{2\triangle_i}{k_i(k_i-1)} & \text{if } k_i \geq 2, \\ 0 & \text{otherwise,} \end{cases} \tag{27.6}$$

where $\triangle_i$ is the number of triangles that contain i. Then the average clustering coefficient[8] of the whole network is simply the average of all C_i's:

$$C = \frac{1}{N}\sum_i C_i. \tag{27.7}$$

Meanwhile, we now write the assortativity coefficient (Eq. (12.10)) as,

$$\begin{aligned} r &:= \frac{8M\sum_{(i,j)\in E} k_i k_j - \left[\sum_{(i,j)\in E}\left(k_i + k_j\right)\right]^2}{4M\sum_{(i,j)\in E}\left(k_i^2 + k_j^2\right) - \left[\sum_{(i,j)\in E}\left(k_i + k_j\right)\right]^2} \\ &= \frac{8Mu - v^2}{4Mw - v^2}, \end{aligned} \tag{27.8}$$

where

$$\begin{aligned} u &:= \sum_{(i,j)\in E} k_i k_j, \\ v &:= \sum_{(i,j)\in E}\left(k_i + k_j\right), \\ w &:= \sum_{(i,j)\in E}\left(k_i^2 + k_j^2\right), \end{aligned} \tag{27.9}$$

and modularity (Eqs. (12.15) and (12.16)) as

$$Q = \frac{1}{2M}\sum_{i,j}\left(A_{ij} - \frac{k_i k_j}{2M}\right)\delta(g_i, g_j) = \frac{1}{2M}\left[S_A - \frac{1}{2M}S_P\right], \tag{27.10}$$

where $\delta(g_i, g_j) = 1$ if nodes i and j are in the same group and zero otherwise, and

$$S_A := \sum_{i,j} A_{ij}\delta(g_i, g_j), \quad S_P := \sum_{i,j} k_i k_j \delta(g_i, g_j). \tag{27.11}$$

27.6.1 Updating schemes

Connecting a new node

First, consider connecting a new node. We can decompose this into two successive operations, adding a new isolated node to the network, then adding an edge from that new node to an existing node. (Multiple connections can then be handled using the updating scheme for connecting edges between existing nodes; see below.) We use $\widetilde{\cdot}$ to denote updated statistics.

[8] Remember that transitivity is a better global measure; Ch. 12 and Ex. 27.5.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

With a new node and no new edges, we have the following update relations:

$$\widetilde{N} = N + 1, \quad \widetilde{M} = M, \quad \widetilde{E} = E, \tag{27.12}$$

and[9]

$$\widetilde{A}_{ij} = \begin{cases} A_{ij} & \text{if } i \neq N+1 \text{ and } j \neq N+1, \\ 0 & \text{otherwise.} \end{cases} \tag{27.13}$$

The other statistics follow likewise:

$$\widetilde{k}_i = k_i, \; i \neq N+1; \quad \widetilde{k}_{N+1} = 0; \tag{27.14}$$

and

$$\widetilde{C}_i = C_i, \; i \neq N+1; \quad \widetilde{C}_{N+1} = 0. \tag{27.15}$$

Similarly, $\widetilde{r} = r$ since $\widetilde{u} = u$, $\widetilde{v} = v$, and $\widetilde{w} = w$; and $\widetilde{Q} = Q$ since $\widetilde{S}_A = S_A$, and $\widetilde{S}_P = S_P$.

Deleting a degree-zero node follows likewise.

Adding a new edge between existing nodes

Suppose nodes p and q are not connected ($A_{pq} = 0$). Here we derive how adding edge p, q changes our statistics. First, since we inserted an edge we have

$$\widetilde{E} = E \cup \{(p,q),(q,p)\}, \tag{27.16}$$

$$\widetilde{M} = M + \Delta^+ M = M + 1, \tag{27.17}$$

and

$$\widetilde{A}_{ij} = A_{ij} + \Delta^+ A_{ij} = A_{ij} + \delta_{ip}\delta_{jq} + \delta_{iq}\delta_{jp}, \tag{27.18}$$

where we use the "update delta" Δ^+ to denote the change in statistic after adding an edge to the existing network. Using these expressions, we can now derive efficient updating schemes for network statistics.

Degree The degree update is simple, as only p and q are affected:

$$\widetilde{k}_i = k_i + \Delta^+ k_i = k_i + \delta_{ip} + \delta_{iq}. \tag{27.19}$$

If we wish to study the network's degree distribution, we can use this update to track how many nodes currently have a given degree by, after an update $k \to k+1$, decrementing the number of nodes with degree k and incrementing the number with $k+1$.

[9] We use **A** for presenting calculations, but it will serve as a placeholder for how the network data system is actually implemented, which is unlikely to be with an adjacency matrix, even if the network is very sparse.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Clustering coefficient To compute the new clustering coefficient of each node, and thus the whole network, we need the updated number of triangles at node i:

$$\widetilde{\triangle}_i = \begin{cases} \triangle_i & \text{if } i \notin \{p,q\} \cup N_{pq}, \\ \triangle_i + 1 & \text{if } i \in N_{pq}, \\ \triangle_i + \left|N_{pq}\right| & \text{if } i \in \{p,q\}, \end{cases} \tag{27.20}$$

where $N_{ij} := N_i \cap N_j$ is the shared neighborhood of nodes i and j. Combining this with Eq. (27.19) and $\triangle_i = \frac{1}{2} C_i k_i (k_i - 1)$, from Eq. (27.6), we have

$$\widetilde{C}_i = \begin{cases} C_i & \text{if } i \notin \{p,q\} \cup N_{pq}, \\ C_i + \frac{2}{k_i(k_i-1)} & \text{if } i \in N_{pq}, \\ \frac{k_i-1}{k_i+1} C_i + \frac{2\left|N_{pq}\right|}{k_i(k_i+1)} & \text{if } i \in \{p,q\}. \end{cases} \tag{27.21}$$

(Whenever the denominator of a fraction is zero, we define the fraction to be zero, in Eq. (27.21) and throughout. This ensures $C_i = 0$ if $k_i < 2$.) Finally, the average clustering coefficient C becomes

$$\widetilde{C} = C + \Delta^+ C = C + \frac{2}{N} \left[\sum_{i \in N_{pq}} \frac{1}{k_i(k_i-1)} + \sum_{i \in \{p,q\}} \left(\frac{\left|N_{pq}\right|}{k_i(k_i+1)} - \frac{C_i}{k_i+1} \right) \right], \tag{27.22}$$

where

$$\Delta^+ C = \frac{2}{N} \left[\sum_{i \in N_{pq}} \frac{1}{k_i(k_i-1)} + \sum_{i \in \{p,q\}} \left(\frac{\left|N_{pq}\right|}{k_i(k_i+1)} - \frac{C_i}{k_i+1} \right) \right]. \tag{27.23}$$

Notice that updating the average clustering coefficient requires keeping C_i for each i, at $O(N)$ space complexity.

Degree assortativity To compute $\widetilde{r}$, we need $\widetilde{u}$, $\widetilde{v}$, and $\widetilde{w}$. The update for u is

$$\begin{aligned} \widetilde{u} &= \sum_{(i,j) \in \widetilde{E}} \widetilde{k}_i \widetilde{k}_j = \sum_{(i,j) \in E} \widetilde{k}_i \widetilde{k}_j + 2(k_p+1)(k_q+1) \\ &= \sum_{(i,j) \in \widehat{E}} k_i k_j + 2 \sum_{i \in N_p} k_i (k_p+1) + 2 \sum_{i \in N_q} k_i (k_q+1) \\ &\quad + 2(k_p+1)(k_q+1) \\ &= u + 2 \left(\sum_{i \in N_p} k_i + \sum_{i \in N_q} k_i \right) + 2(k_p+1)(k_q+1) \\ &= u + \Delta^+ u. \end{aligned} \tag{27.24}$$

Here $\widehat{E} = E \setminus \{(p,q),(q,p)\}$ is the edge set that contains all edges in E but (p,q) and (q,p) and

$$\Delta^+ u = 2 \left(\sum_{i \in N_p} k_i + \sum_{i \in N_q} k_i \right) + 2(k_p+1)(k_q+1). \tag{27.25}$$

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Similarly, the update formula for v and w are

$$\widetilde{v} = \sum_{(i,j)\in\widetilde{E}} (\widetilde{k}_i + \widetilde{k}_j) = v + \Delta^+ v = v + 4(k_p + k_q + 1), \tag{27.26}$$

and

$$\widetilde{w} = \sum_{(i,j)\in\widetilde{E}} (\widetilde{k}_i^2 + \widetilde{k}_j^2) = w + \Delta^+ w, \tag{27.27}$$

where

$$\Delta^+ w = 6\left[k_p(k_p+1) + k_q(k_q+1)\right] + 4. \tag{27.28}$$

Finally, the new assortativity coefficient can be updated using

$$\widetilde{r} = r + \Delta^+ r = \frac{8\widetilde{M}\widetilde{u} - \widetilde{v}^2}{4\widetilde{M}\widetilde{w} - \widetilde{v}^2} = \frac{8\,(M+1)\,(u+\Delta^+ u) - (v+\Delta^+ v)^2}{4\,(M+1)\,(w+\Delta^+ w) - (v+\Delta^+ v)^2}. \tag{27.29}$$

Modularity For modularity, we assume that after connecting the nodes p and q, the partitions g_i do not change for any node i.[10] Then the new modularity measure will be

$$\widetilde{Q} = \frac{1}{2\widetilde{M}}\left[\widetilde{S}_A - \frac{1}{2\widetilde{M}}\widetilde{S}_P\right]. \tag{27.30}$$

We already have $\widetilde{M} = M + 1$, we now derive updating formulas for S_A and S_P. By Eq. (27.11), we have

$$\begin{aligned}\widetilde{S}_A = S_A + \Delta^+ S_A &= \sum_{i,j} \widetilde{a}_{ij}\delta(g_i, g_j) \\ &= \sum_{i,j} \left(a_{ij} + \delta_{ip}\delta_{jq} + \delta_{iq}\delta_{jp}\right)\delta(g_i, g_j) \\ &= S_A + 2\delta(g_p, g_q)\end{aligned} \tag{27.31}$$

and

$$\begin{aligned}\widetilde{S_P} = S_P + \Delta^+ S_P &= \sum_{i,j} \widetilde{k}_i\widetilde{k}_j\delta(g_i, g_j) \\ &= \sum_{i,j} \left(k_i + \delta_{ip} + \delta_{iq}\right)\left(k_j + \delta_{jp} + \delta_{jq}\right)\delta(g_i, g_j) \\ &= S_P + 2\sum_i k_i\left[\delta(g_i, g_p) + \delta(g_i, g_q)\right] + 2\left[\delta(g_p, g_q) + 1\right].\end{aligned} \tag{27.32}$$

However, computing the sum in Eq. (27.32) after every update is expensive. To avoid this, we introduce the following auxiliary statistics:

$$K_g := \sum_i k_i\delta(g_i, g) \tag{27.33}$$

[10] We actually already encountered the updating scheme for a partition change in Sec. 25.6.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

with updating scheme

$$\widetilde{K}_g = K_g + \Delta^+ K_g = K_g + \delta(g_p, g) + \delta(g_q, g), \tag{27.34}$$

giving

$$\widetilde{S}_P = S_P + \Delta^+ S_P = S_P + 2\left(K_{g_p} + K_{g_q}\right) + 2\big[\delta(g_p, g_q) + 1\big], \tag{27.35}$$

where $\Delta^+ S_P = 2\left(K_{g_p} + K_{g_q}\right) + 2\big[\delta(g_p, g_q) + 1\big]$. Finally, combining Eq. (27.31) and Eq. (27.35) with Eq. (27.30) gives the updating scheme for Q:

$$\begin{aligned}\widetilde{Q} = Q + \Delta^+ Q = \frac{1}{2(M+1)}\Big[S_A + 2\delta(g_p, g_q) \\ - \frac{1}{2(M+1)}\left(S_p + 2[K_{g_p} + K_{g_q}] + 2[\delta(g_p, g_q) + 1]\right)\Big].\end{aligned} \tag{27.36}$$

From Eq. (27.36) we can predict whether Q increases or decreases given the existing partition and the edge to be added. For example, if there is a preexisting partition of the network into two groups, then if a new edge is added in between the groups, $\Delta^+ Q < 0$. On the other hand, if a new edge connects nodes in the same group, then the modularity is sure to increase *only if* the edge is added to the group with smaller total degree. Perhaps surprisingly, adding an edge within the group does not necessarily increase Q if the edge is added into the group with larger total degree.

Removing an existing edge

Now we focus on updates when an existing edge is deleted. These updates can also be used for the removal of a node, since removing a node requires deleting its edges then deleting the now-disconnected node.

Suppose $p \neq q$ and p, q are connected, and we delete this edge, $(p, q) \cup (q, p)$, from our edge set E. We'll use $\widehat{\cdot}$ to represent the updated statistics. First, we immediately have

$$\widehat{E} = E \setminus \{(p, q), (q, p)\}, \tag{27.37}$$

$$\widehat{M} = M - 1, \tag{27.38}$$

$$\widehat{A}_{ij} = A_{ij} + \Delta^- A_{ij} = A_{ij} - \delta_{ip}\delta_{jq} - \delta_{iq}\delta_{jp}, \tag{27.39}$$

where Δ^- will be used to denote change for statistics upon deleting an existing edge.

Degree The change in degree for node i is simply

$$\widehat{k}_i = k_i + \Delta^- k_i = k_i - \delta_{ip} - \delta_{iq}. \tag{27.40}$$

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Clustering coefficient For the new clustering coefficient, as before, we start with the updating scheme for the number of triangles containing node i:

$$\widehat{\triangle}_i = \begin{cases} \triangle_i & \text{if } i \notin \{p,q\} \cup N_{pq}, \\ \triangle_i - 1 & \text{if } i \in N_{pq}, \\ \triangle_i - |N_{pq}| & \text{if } i \in \{p,q\}. \end{cases} \tag{27.41}$$

Then we obtain the scheme for updating C_i:

$$\widehat{C}_i = \begin{cases} C_i & \text{if } i \notin \{p,q\} \cup N_{pq}, \\ C_i - \frac{2}{k_i(k_i-1)} & \text{if } i \in N_{pq}, \\ \frac{k_i}{k_i-2} C_i - \frac{2|N_{pq}|}{(k_i-1)(k_i-2)} & \text{if } i \in \{p,q\}. \end{cases} \tag{27.42}$$

The average clustering coefficient C is updated by

$$\begin{aligned} \widehat{C} &= C + \Delta^- C = \frac{1}{N} \sum_i \widehat{C}_i \\ &= C - \frac{2}{N} \left[\sum_{i \in N_{pq}} \frac{1}{k_i(k_i-1)} + \sum_{i \in \{p,q\}} \left(\frac{|N_{pq}|}{(k_i-1)(k_i-2)} - \frac{C_i}{k_i-2} \right) \right]. \end{aligned} \tag{27.43}$$

Degree assortativity The updating formulas for u, v, w are

$$\begin{aligned} \widehat{u} = u + \Delta^- u &= u - 2 \left(\sum_{i \in N(p)} k_i + \sum_{i \in N(q)} k_i \right) - 2(k_p - 1)(k_q - 1), \\ \widehat{v} = v + \Delta^- v &= v - 4(k_p + k_q - 1), \\ \widehat{w} = w + \Delta^- w &= w - 6 \left[k_p(k_p - 1) + k_q(k_q - 1) \right] - 4, \end{aligned} \tag{27.44}$$

and the new assortativity coefficient $\widehat{r}$ is

$$\widehat{r} = \frac{8\widehat{M}\widehat{u} - \widehat{v}^2}{4\widehat{M}\widehat{w} - \widehat{v}^2} = \frac{8(M-1)(u + \Delta^- u) - (v + \Delta^- v)^2}{4(M-1)(w + \Delta^- w) - (v + \Delta^- v)^2}. \tag{27.45}$$

Modularity Once more, we assume that the community partitions g_i are unchanged after disconnecting the edge between p and q. It follows that

$$\widehat{S}_A = S_A + \Delta^- S_A = S_A - 2\delta(g_p, g_q), \tag{27.46}$$

$$\widehat{S}_P = S_P + \Delta^- S_P = S_P - 2 \left(K_{g_p} + K_{g_q} \right) + 2 \left(\delta(g_p, g_q), +1 \right), \tag{27.47}$$

where K_g is now updated using

$$\widehat{K}_g = K_g + \Delta^- K_g = K_g - \delta(g_p, g) - \delta(g_q, g). \tag{27.48}$$

These now define the updating scheme for $\widehat{Q} = \left(\widehat{S}_A - \widehat{S}_P / 2\widehat{M} \right) / 2\widehat{M}$.

Lastly, in Table 27.1 we compare the computational complexity of these updating schemes to solutions using an adjacency matrix and an edgelist [452]. In all cases, the updating schemes have better (lower) complexity.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Table 27.1 Complexity of updating schemes [452].

Statistic	Adjacency matrix	Edge list	Updating scheme
Degree (one node)	$O(N)$	$O(\langle k\rangle)$	$O(1)$
Degree (network)	$O\left(N^2\right)$	$O(\langle k\rangle N)$	$O(1)$
Clustering coefficient (one node)	$O(\langle k\rangle N)$	$O\left(\langle k\rangle^3\right)$	$O(1)$ /$O(\langle k\rangle)$
Clustering coefficient (network)	$O\left(\langle k\rangle N^2\right)$	$O\left(\langle k\rangle^3 N\right)$	$O(\langle k\rangle)$
Assortativity	$O\left(N^2\right)$	$O(\langle k\rangle N)$	$O(\langle k\rangle)$
Modularity	$O\left(N^2\right)$	$O(\langle k\rangle N)$	$O(1)$

27.7 Making graphs

Most graph models are implemented in ways that don't account for scalability. For example, suppose we wish to generate a very large Erdős–Rényi graph $G(N,p)$ on $N \gg 1$ nodes where each edge exists with probability p. The naive, or unscalable way is to take each pair of nodes (i,j), of which there are $\binom{N}{2} = O\left(N^2\right)$, generate a random number r uniformly distributed on $[0,1]$, and insert edge i,j if $r < p$. Because we are looping through the set of nodes twice, we have a quadratic algorithm; fine for small networks, prohibitive for large ones.

This naive double-loop is also wasteful. We individually test every pair of nodes, and (depending on p) many of those tests will fail and no edge will be created. A more efficient approach (see also remarks) is to change how edges are inserted to skip over the node pairs where no edge will be inserted.[11] We describe this now.

Suppose nodes are numbered $0,1,\ldots,N-1$. We begin at node $u=0$. Then for each node $v=1,\ldots,N-1$, the naive algorithm would generate r and insert edge u,v if $r<p$. The process would continue with $u \geq 1,2,\ldots$ and $v = u+1$. Let's be more efficient. Let $v_1 = u+1+\delta$ be the first neighbor with which u forms an edge. Here δ is the number of intervening pairs u,v_i that did not form an edge. Similarly, $v_2 = v_1+1+\delta$ where now δ is the new number of intervening pairs between v_1 and v_2. The steps δ are distributed geometrically: $\Pr(\delta) = (1-p)^\delta p$. Therefore, instead of considering every v after u, we generate δ and skip ahead that many nodes. Specifically, generate r uniformly on $(0,1)$ and set $\delta = \lfloor \log(r)/\log(1-p) \rfloor$, where $\lfloor\cdot\rfloor$ is the floor function.[12] (We take $\delta = 0$ if $p = 1$.) Then we can insert edge $u, v+1+\delta$, and continue. We give the full algorithm in Alg. 27.3.

While the naive algorithm has an $O\left(N^2\right)$ complexity, Alg. 27.3 has a complexity of only $O(N+M)$. To see this, consider the average number of times the inner loop

[11] We can also just generate edges as integer pairs $i,j \sim U[0,N-1]$, but we focus on the "skipping" method because it more easily generalizes to the other graph models we will discuss.

[12] This expression comes from using *inverse transform sampling* [130]. We compute the CDF of $\Pr(\delta)$, set it equal to $r \sim U[0,1]$, then solve for δ. When the CDF $F(x)$ can be inverted, this is a clever way to transform $U[0,1]$-distributed random numbers to $x \sim f(x)$. In our case, we have $r = \sum_{d=\delta}^{\infty} p(1-p)^d = (1-p)^\delta \Rightarrow \delta = \log r/\log(1-p)$.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Algorithm 27.3 Efficient algorithm for generating Erdős–Rényi graphs. Runs in $O(N+M)$.

```
Input: Number of nodes N, edge probability 0 < p < 1
E ← ∅
for u = 0 to N − 2 do
    v ← u + 1
    while v < N do
        draw r ~ U[0, 1]
        v ← v + ⌊log(r)/log(1−p)⌋
        if v < N then
            E ← E ∪ {(u, v)}
            v ← v + 1
Return: instance of G(N, p) with nodes 0, ..., N − 1
```

executes: $1 + N/\langle\delta\rangle = 1 + O(Np)$, the 1 being the number of unsuccessful times. The outer loop executes N times, giving a total complexity of $N(1 + O(Np)) = O(N + M)$ since $M = \binom{N}{2}p$. This complexity is just about optimal in that we'd need to examine every edge during generation regardless.

Now, we generalize this method from Erdős–Rényi networks to *Chung–Lu* networks [107], networks with a given expected degree sequence. These networks introduce heterogeneity into the degree distribution by using node weights w_u and setting the probability for an edge u, v to be $p_{uv} = w_u w_v / N\bar{w}$, where $\bar{w} = \sum_u w_u / N$.

The obvious, naive $O(N^2)$ algorithm consists of again performing a double loop over all nodes and checking for each edge by generating $r \sim U[0, 1]$ and inserting edge u, v if $r < w_u w_v / N\bar{w}$. We again wish to be efficient by skipping unnecessary node pairs, but this will be more difficult because the probabilities for possible edges will not be fixed. To derive a more efficient algorithm, we first present an alternative to the naive algorithm that can be modified to be efficient.

First, assume the list W of weights for the network is available and sorted in descending order. As we consider every $v = u + 1, \ldots, N - 1$, p_{uv} decreases monotonically, so we avoid recalculating p for each v by setting $p = p_{u,u+1} = w_u w_{u+1} / N\bar{w}$ and skipping each v with probability $1 - p$. Arriving at the first node v_1 we do not skip, we then need to check if v_1 is actually neighbors with u, since $p_{uv_1} \leq p$. We calculate $q = p_{uv_1}$ and assign the edge with probability q/p. We then set $p = q$ and continue to discard nodes with probability $1 - p$ until all v have been considered. Then we increment u and repeat the process. The probability that an edge is inserted, $pp_{uv_1}/p = p_{uv_1}$ is as we expect and, like the naive algorithm, this algorithm is $O(N^2)$.

Why organize the naive algorithm in this way? Doing so ensures that p is fixed at each step until a *potential neighbor* is identified, allowing us to identify how to skip over those intervening nodes. Starting with $u = 0$, and setting $p = p_{u,u+1}$, draw $r \sim U[0, 1]$ and find the first potential neighbor $v_1 = u + 1 + \delta$ by generating a step $\delta = \lfloor \log r / \log(1-p) \rfloor$ (taking $\delta = 0$ if $p = 1$). Once v_1 is selected, insert edge u, v_1 with probability p_{uv_1}/p. Then set $p = p_{u,v1}$ and continue to the second potential neighbor v_2 by generating another value of δ. Since p decreases monotonically for a given u, the

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

expected value of δ increases monotonically. We give the full algorithm for Chung–Lu graphs in Alg. 27.4.

Algorithm 27.4 Efficient algorithm for generating Chung–Lu [107] graphs. Runs in $O(N + M)$.

Input: sorted list of N weights $w_0 \geq w_1 \geq \ldots \geq w_{N-1}$
$E \leftarrow \emptyset$
$S \leftarrow \sum_u w_u$
for $u = 0$ to $N - 2$ **do**
    $v \leftarrow u + 1$
    $p \leftarrow \min(w_u w_v / S, 1)$
    while $v < N$ and $p > 0$ **do**
        if $p \neq 1$ **then**
            draw $r \sim U[0, 1]$
            $v \leftarrow v + \left\lfloor \frac{\log(r)}{\log(1-p)} \right\rfloor$
        if $v < N$ **then**
            $q \leftarrow \min(w_u w_v / S, 1)$
            draw $r \sim U[0, 1]$
            if $r < q/p$ **then**
                $E \leftarrow E \cup \{(u, v)\}$
            $p \leftarrow q$
            $v \leftarrow v + 1$
Return: instance of Chung–Lu graph with nodes $0, \ldots, N - 1$

Algorithm 27.4 has complexity $O(N + M)$. We omit the proof, which is more involved compared to that of Alg. 27.3 due to the dynamic rejection sampling. We refer readers to Miller and Hagberg [310] for details.

27.8 Summary

What should we do when networks get big? Really big? A variety of tools, such as graph databases, probabilistic data structures, and local algorithms, are at our disposal, especially if we are able to accept sampling effects and uncertainty. But remember our admonishment: there is an opportunity cost to these solutions and it is very often the case that they are not needed. Try simpler methods first, then adapt.

Areas where such tools are needed include web crawls, online social networks, and knowledge graphs, and major graph database systems have been proposed for such domains. Systems biology is yet another area where networks will continue to grow: as sequencing methods continue to advance, more networks and larger, denser networks will need to be analyzed. Perhaps not to the scale of trillions of edges, but perhaps so. Some day, big data solutions may be necessary. Either way, infrastructure is in place if and when that scale needs to be confronted.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

Bibliographic remarks

For more on "big data," its general challenges and opportunities, see Madden [286], Marx [296], Fan et al. [155], Chen et al. [102], and references therein. For a broad introduction to graph databases see Robinson et al. [399]. An early and very influential graph storage project was the Connectivity Server [54]. Many of its ideas became standard practice for large graph storage and compression, including in the WebGraph Framework [62]. For a recent survey on the related problem of graph summarization, see Liu et al. [278]

Many problems related to graph streams have been studied, see McGregor [300] for a survey. Detecting anomalies in a graph stream is an important, practical problem; see Aggarwal et al. [4], Manzoor et al. [290], and Eswaran et al. [151] for methods and Ranshous et al. [389] for a review. Enumerating subgraphs is another streaming problem, and McGregor et al. [301] discuss algorithms for it. More generally, Al Hasan and Dave [8] give a survey of triangle counting methods that covers both graph streams and systems where random access to the graph is possible.

Space-efficient approximate or probabilistic counting dates back to Morris [321]. The HyperLogLog method for cardinality estimation we describe is the culmination of a long line of research [160, 137, 161]. Heule et al. [212] devised the HyperLogLog++ algorithm to help improve practical application of the method.

Approximate Neighborhood Functions were introduced by Palmer et al. [355]. ANF used Flajolet–Martin counters [160]. Boldi et al. [63] introduced HyperANF by extending ANF with more advanced HyperLogLog counters (which were introduced after ANF) and other programming advancements.

Fast community detection with label propagation (LP) was introduced in a highly influential paper by Raghavan et al. [384]. An interesting variant that aligns LP with modularity by using constraints was introduced by Barber and Clark [38]. A multi-resolution variant of LP was also employed by Boldi et al. [64] for graph compression.

The dynamic updating schemes we presented were introduced by Sun et al. [452].

The algorithms we presented for efficient Erdős–Rényi and Chung–Lu graphs were introduced by Miller and Hagberg [310]. Batagelj and Brandes [45] and Hagberg and Lemons [198] also address this problem for other graph models. Ramani et al. [385] provide a readable summary of efficient graph-making strategies across a variety of models.

Exercises

27.1 HyperANF (Alg. 27.2) assumes you can sequentially access the neighbors of a node (the inner loop). What if you have a graph stream where edges are given one at a time? What problems may this pose with the basic algorithm and with optimizing it?

27.2 Suppose you use HyperANF to compute the neighborhood function for a network. Then that network changes, perhaps some edges have been rewired. How much does this affect your previous computation? Will you need to rerun HyperANF from scratch?

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press

27.3 Modify HyperANF to compute the graph conductance $\text{cut}(S)/\text{vol}(S)$ (Sec. 25.5) of a set of nodes S (S is an input to the new algorithm and assume it can be stored in memory). What additional data must be tracked during the outer loop?

27.4 Can label propagation be used for local community detection, i.e., to find the community containing a starting node i without needing to examine the (entire) rest of the network? Why or why not?

27.5 Derive the updating schemes for transitivity (Eq. (12.9)) instead of the average clustering coefficient.

27.6 Modify the Miller–Hagberg algorithm (Alg. 27.3) for Erdős–Rényi graphs to generate *bipartite* Erdős–Rényi graphs.

https://doi.org/10.1017/9781009212601.032 Published online by Cambridge University Press