Chapter 10
Fundamental Network Algorithms

M. E. J. Newman

May 6, 2015
Table of Contents

1 Algorithms for Degrees and Degree Distributions
   - Degree-Degree Correlation

2 Clustering Coefficient

3 Shortest Paths
   - Breadth-First Search
   - Variants of BFS
   - Finding Shortest Paths

4 Betweenness Centrality

5 Shortest Paths in Networks with Varying Edge Lengths
Degree

- Degree of a vertex is one of the most fundamental and important of network quantities.
- If we use an adjacency list, we normally maintain an array containing the degree of each vertex.
  - Computational complexity to find a degree of a vertex: $O(1)$.
- If we use an adjacency matrix, the calculation takes time $O(N)$.  
  - If one needed to find the degrees of vertices frequently during a large calculation using an adjacency matrix, it would be better to calculate the degree of each vertex once and for all and store the results for later easy retrieval in a separate array.
Degree Distributions

Degree distributions are of considerable interest in the study of networks for the effect they have on network structure and processes on networks.

Calculating the degree distribution, $p_k$ is straightforward:

- Once we have the degrees of all vertices, we make a histogram of them by creating an array to store the number of vertices of each degree up to the network maximum.
- Let $\text{ndegree}[i]$ be the $i$th element of the array.
- Initially set $\text{ndegree}[i]=0$ for all $i$.
- Running through the vertices in turn, if the degree of $i$th node is $k_i$ then increase $\text{ndegree}[k_i]$ by one.
- Normalize each element, $\text{ndegree}[i]$, by dividing $N$.
- Computational complexity: $O(N)$.

Cumulative degree distribution: $P_k$

\[
P_k = \sum_{k'}^\infty p_{k'} = -p_{k-1} + \sum_{k'=k-1}^\infty p_{k'} = P_{k-1} - p_{k-1}.
\]  

(1)
Correlation Coefficient

Definition

\[ r = \frac{\sum_{ij} (A_{ij} - k_i k_j / 2m) k_i k_j}{\sum_{ij} (k_i \delta_{ij} - k_i k_j / 2m) k_i k_j} \]  

(2)

Rewrite the equation as

\[ r = \frac{S_1 S_e - S_2^2}{S_1 S_3 - S_2^2}, \]  

(3)

where

\[ S_e = \sum_{ij} A_{ij} k_i k_j = 2 \sum \text{edges} (i,j) k_i k_j, \]  

(4)

and

\[ S_1 = \sum_i k_i, \quad S_2 = \sum_i k_i^2, \quad S_3 = \sum_i k_i^3 \]  

(5)

Use Eq. (3) for calculating correlation coefficient with complexity \( O(m + N) \).
Local Clustering Coefficient

\[ C_i = \frac{\text{\# of pairs of neighbors of } i \text{ that are connected}}{\text{\# of pairs of neighbors of } i} \]  

- For each pairs of neighbor of \( i \), \((j, l) \) \((j < l)\), we determine whether an edge exists between them.
- The details of algorithm depends on the stored network data format.
- Then divide by \( k_i(k_i - 1)/2 \).
Clustering Coefficients

Global Clustering Coefficient

\[ C_i = \frac{[\text{# of triangles}] \times 3}{[\text{# of connected triples}]} \]  

- **connected triples**: three vertices \(uvw\) with edges \((u, v)\) and \((v, w)\). (The edge \((u, v)\) can be present or not).
- For every vertex each pair of neighbors \((j, l)\) with \(j < l\) and find whether they are connected by an edge.
- Add up the total number of such edges.
- Divide by the number of connected triples, which is \(\sum_i k_i (k_i - 1)/2\).
- Unfortunately, the complexity is \(O \left( n \langle k^2 \rangle \left[ \frac{\langle k^2 \rangle}{\langle k \rangle} - 1 \right] \right)\).
- For highly skewed \(p_k\), \(\langle k^2 \rangle\) becomes large or diverges.
- Matrix/list hybrid structure can slightly improve the performance.
Breadth-First Search (BFS)

- Known as the “burning algorithm” in physics.
- A single run of the BFS algorithm finds the shortest (geodesic) distance from a single source vertex $s$ to every other vertex in the same component of the network as $s$.
- Even when we want to know the shortest distance between a single pair of vertices $s$ and $t$, use the BFS algorithm to calculate shortest distance from $s$ to every other vertex and then throwing away all of the results except for the one we want.
- The BFS can be also applied to the directed networks.
Description of the Algorithm

- BFS algorithm finds the shortest distance from a given starting vertex $s$ to every other vertex in the same component as $s$.
- Initially, we set the distance to $s$ be 0.
- Find all the neighbors of $s$, and set the distance to be 1.
- Then find all the neighbors of those vertices.
- If those neighbors are not already visited, set the distance to be 2.
- Repeat the procedure.
A Naive Implementation: Complexity $O(m + N \log N)$

- Create an array of $N$ elements to store the distance of each vertex from the source vertex $s$.
- Initially set the distance to $s$ from itself to be zero.
- All other vertices have unknown distance, for example, $-1$, or some similar value that could not occur in reality.
- We also introduce a distance variable $d$ to keep track of where we are in the BFS process.
- Set $d = 0$ as its initial value.

1. Find all vertices that are distance $d$ from $s$, by going through the distance array, element by element.
2. Find all the neighbors of those vertices and check each one to see if its distance from $s$ is unknown.
3. If the number of neighbors with unknown distance is zero, the algorithm is over. Otherwise, if the number of neighbors with unknown distance is non-zero, set the distance of each of those neighbors to $d + 1$.
4. Increase $d$ by 1.
5. Repeat from step 1.
A Better Implementation

- The time-consuming part of the "naive implementation" is step 1.
  - We go through the list of distances of size $N$ to find vertices that are distance $d$ from the starting vertex $s$.
  - Only the small fraction of the list will be at distance $d \Rightarrow$ waste a lot of time.

- Strategy:
  - Make a list of vertices at distance $d$.
  - Then we wouldn’t have to search through the whole network for vertices at distance $d + 1$.

- Use the fist-in/first-out buffer or queue.
  - Nothing more than an array of $N$ elements.
  - In the array, a list of labels of vertices are stored.
  - On each sweep of the algorithm, we read the vertices with distance $d$ from the list.
  - Use these to find the vertices with distance $d + 1$.
  - Add those vertices with distance $d + 1$ to the list.
  - Repeat the procedure.

- Introduce the read and write pointers, which are the simple integer variables.
A Better Implementation

Algorithm

0. Use two arrays of $N$ elements.
   - one for the queue.
   - one for the distance from $s$ to each vertex.

1. Place the label of the source vertex $s$ in the first element of the queue.
   - Set the read pointer to point the first element.
   - Set the write pointer to point the second element, which is the first empty one.
   - In the distance array, record the distance of vertex $s$ from itself as being zero.
   - Set the distances of all other vertices to be “unknown” (e.g. $-1$).
A Better Implementation

Algorithm

2. If the read and the write pointers are pointing to the same element of the queue then the algorithm is finished. Otherwise, read the vertex label from the element pointed to by the read pointer and increases that pointer by one.

3. Find the distance $d$ for that vertex by looking in the distance array.

4. Go through each neighboring vertex in turn and look up its distance in the distance array.
   - If it has a known distance, leave it alone.
   - If it has an unknown distance, assign it distance $d + 1$, store its label in the queue array in the element pointed to by the write pointer, and increase the write pointer by one.
A Better Implementation

Algorithm

5. Repeat from step 2.
   - Complexity: $O(m + N)$ or on sparse networks, $O(N)$. 
Variants of BFS

The shortest distance between a specific pair of nodes
- Apply BFS from a source vertex \( s \), and stop the algorithm if a target \( t \) is found.

The shortest distance between every pair of nodes
- Apply BFS starting at each vertex in the network in turn.
- Complexity: \( O(N(m + N)) \) or \( O(N^2) \) on a sparse network.

Identify the component
- Apply BFS from a source \( s \).
- Size of the component is the number of vertices with known distance.

Closeness centrality
- Apply BFS from a source vertex \( s \).
- Add the distances of all nodes with known distance and divide by the size of the component and take the inverse.
Finding Shortest Paths

- BFS algorithm so far does not tell us the particular path or paths by which that shortest distance is achieved.
- With only relatively small modification of the algorithm gives the paths as well.
- Introduce another directed network on top of the original one.
- The directed network represents the shortest paths ⇒ shortest path tree.
- In general it is a directed acyclic graph, not a tree.
Finding Shortest Paths

Idea

- At start of BFS algorithm, create an extra network, which will become the shortest path tree.
  - The extra network has the same number $N$ of vertices as the original network.
  - The extra network has the same vertex labels, but with no edges at all.
- Start BFS algorithm from the specified source vertex $s$.
- At each time, if the neighbor $j$ of some vertex $i$ has a distance “unknown”, we assign $j$ a distance, store it in the queue, and add a directed edge to the shortest path tree from $j$ to $i$.
- This directed edge tells us that we found $j$ by following a path from its neighbor $i$.
- By following succession of these directed edge, we eventually get all the way back to $s$, and so we can reconstruct entire shortest path between $(j, s)$. 
Problem

- This algorithm finds only one shortest path to each vertex from $s$.
- In general, a pair of vertices can have more than one shortest paths between them.
- This occurs if there is a vertex $j$ somewhere along that path, say at distance $d + 1$ from $s$, that has more than one neighbor at distance $d$.
- We can resolve the problem by adding more than one directed edge from $j$ to each of the relevant neighbors.
Finding Shortest Paths

Modified Algorithm

- We perform the BFS starting from $s$.
- Add directed edges from newly found vertices to their neighbor.
- If in the process of examining the neighbor of a vertex $i$ that has distance $d$ from $s$, we discover a neighbor $j$ that already has a “known” distance $d + 1$, then add an extra directed edge to the shortest path tree from $j$ to $i$.
  - This makes the shortest path tree no longer a tree, but it’s usually called as a tree.
The betweenness centrality (BC) of vertex $v$ is the number of geodesic paths between pairs of vertices $s, t$ that pass through $v$.

**The simplest way:**
- Implement the definition of the measure directly:
  - Use BFS to find the shortest path between $s$ and $t$.
  - Along that shortest paths, check the vertices it passes to see if the vertex $v$ lies among them.
  - Repeat the process for every distinct pair $s, t$, then count the total number of paths that pass through $v$.
- Complexity: $O(N^2(m + N))$ or $O(N^3)$ for sparse network.
- Should be improved.
Betweenness Centrality

Improved Algorithm

- Make use of some of the results about BFS.
- For each $s$ we use BFS to find shortest paths between $s$ and all other nodes.
  - Construct a shortest path tree.
  - Use that tree to trace the paths from each node back to $s$.
  - Count the number of paths that go through $v$.
- Repeat this calculation for all $s$.
- Complexity: $O(N^2 \log N)$.
Further Improvement

- First consider the case when there is only a single shortest path.
  - Find the “leaves” of the tree.
  - Assign a score 1 to each of these leaves.
    - This means that the only path to \( s \) that pass through these vertices is the one that starts there.
  - Starting at the bottom of the tree we go upward, assigning to each vertex a score that is 1 plus the sum of the scores on the neighboring nodes immediately below it.
    - This means that the number of paths through a node \( v \) is 1 for the path that starts at \( v \) plus the count of all paths that starts below \( v \) in the tree and hence have to pass through it.
Betweenness Centrality

Further Improvement—cont’d

- When we have worked all the way up the tree in this manner and reaches \( s \), the scores at each vertex are equal to the betweenness counts for path that end at \( s \).
- Repeating the process for all \( s \) and summing the scores.
- The resulting score is the full betweenness score for all paths.
- In practice, this modified algorithm is accomplished by running through the vertices in order of decreasing distance from \( s \).
  - Use the queue.
  - The algorithm involves running backwards through the list of vertices in the queue and calculating the number of paths through each nodes until the beginning of the queue is reached.
  - Complexity: \( O(N(m + N)) \).
More General Case

- When there is loops, it is convenient to think about the flow BC introduced by Freeman et al.
- Basic idea is that the multiple shortest paths between the same pair of vertices are given equal weights summing to 1.
  - Ex]. for a vertex pair connected by three shortest paths, we give each path weight 1/3.
- To calculate the weights of the paths flowing through each vertex in a network, we have first to calculate the total number of shortest paths from each vertex to $s$.
  - This is quite straightforward to do:
  - The shortest paths from $s$ to $i$ must pass through one or more neighbor of $i$
  - The total number of shortest paths to $i$ is simply the sum of the numbers of shortest paths to each of those neighbors.
Betweenness Centrality

Flow BC: Algorithm – assigning weight

- Use the modified BFS.
- Suppose we are starting at $s$.
  1. Assign vertex $s$ distance zero, to indicate that it is zero steps from itself and set $d = 0$. Also assign $s$ a weight $w_s = 1$.
  2. For each vertex $i$ whose assigned distance is $d$, follow each attached edge to the vertex $j$ at its other end and do one of the following three things:
     a) If $j$ has not yet been assigned a distance, assign it distance $d + 1$ and weight $w_j = w_i$.
     b) If $j$ has already assigned a distance and that distance is equal to $d + 1$, then the vertex’s weight is increased by $w_i$, that is $w_j \leftarrow w_j + w_i$.
     c) If $j$ has already been assigned a distance less than $d + 1$, do nothing.
  3. Increase $d$ by 1.
  4. Repeat from step 2 until there are no vertices that have distance $d$.

- See the numbers at the left of each vertex (weight).
Betweenness Centrality

Flow BC: Algorithm – determining BC

- $w_i$ is the number of geodesic paths between $s$ and $i$.
- If $i$ and $j$ are connected by a directed edge in the shortest path tree pointing from $j$ to $i$, then the fraction of the paths to $s$ that pass through (or starting at) $j$ and that also pass through $i$ is given by $w_i/w_j$. 

![Diagram showing geodesic paths and weights between nodes]

- $w_i$ is the number of geodesic paths between $s$ and $i$.
- If $i$ and $j$ are connected by a directed edge in the shortest path tree pointing from $j$ to $i$, then the fraction of the paths to $s$ that pass through (or starting at) $j$ and that also pass through $i$ is given by $w_i/w_j$. 

![Diagram showing geodesic paths and weights between nodes]
Betweenness Centrality

Flow BC: Algorithm – determining BC

1. Find every “leaf” vertex $t$ and assign it a score of $x_t = 1$.
2. Starting at the bottom of the tree, work up towards $s$ and assign to each vertex $i$ a score $x_i = 1 + \sum_j x_j w_i / w_j$, where the sum is over the neighbors $j$ immediately below vertex $i$.
3. Repeat from step 2 until $s$ is reached.

See the numbers at the right of each vertex.

Complexity: $O(N(N + m))$ or $O(N^2)$ for a sparse network.
Shortest Paths in Networks with Varying Edge Lengths

- Weighted Networks:
  - the traffic capacity of connections on the Internet
  - frequencies of contacts between acquaintances in a social network
- In some cases the values on edges can be interpreted as lengths for edges.
- The length can be:
  - real length such as the distances along roads in a road network
  - or length like measures.
- Finding shortest path between two nodes taking the length of edges into account.
  - finding the shortest driving route from $A$ to $B$ via a road network
  - finding the route across the Internet to send a data packet to a specified destination in the shortest time.
Problem:

- The shortest path across a network with edge length may not be the same as the shortest path in terms of number of edges.

To solve the problem, introduce *Dijkstra’s algorithm* (DA).

- DA works by keeping a record of the shortest distance it has found so far to each vertex and updating that record whenever a shorter one is found.
Dijkstra’s Algorithm

Dijkstra’s Algorithm – Preliminary Procedures

- Start by creating an array of $N$ elements to hold our current estimates of the distances from $s$ to every vertex.
  - At all time during the running of the algorithm these estimates are upper bounds on the true shortest distances.
  - Initially, set the estimates of the distance from $s$ to $s$ to be zero.
  - Set the distance from $s$ to every other node to be $\infty$.
- Create another array of $N$ elements.
  - This is used to save the information whether we are certain that the distance we have to a given vertex is the smallest possible distance.
  - Ex.) use 1s to indicate the distances we are sure about and 0s for the distances that are just our best current estimate.
  - Initially, set all element of this array to be 0.
Dijkstra’s Algorithm

Dijkstra’s Algorithm – Finding the Shortest Distances

1. Find the vertex \( v \) in the network that has the smallest estimated distance from \( s \), i.e., the smallest distance about which we are not yet certain.

2. Mark this distance as being certain.

3. Calculate the distance from \( s \) via \( v \) to each of the neighbors of \( v \) by adding to \( v \)’s distance the length of the edges connecting \( v \) to each neighbor.
   - If any of the resulting distances is smaller than the current estimated distance to the same neighbor, the new distance replaces the older one.

4. Repeat from step 1 until the distance to all vertices are flagged as being certain

   - Complexity: \( O(m + N^2) \) or \( O(m + N) \) with binary heap.
   - Store the estimated distance in a binary heap.
Dijkstra’s Algorithm

Dijkstra’s Algorithm – Why it works correctly?

- The crucial step is step 2.
  - We declare the current smallest estimated distance to be certain.
  - Consider a vertex \( v \).
  - Consider a hypothetical path from \( s \) is shorter than the current estimated distance recorded for \( v \).
  - Since the hypothetical path is shorter than the estimated distance to \( v \), the distance along the path to each vertex in the path must be shorter than the estimated distance to \( v \).
  - Furthermore, there must exist somewhere along the path a pair of adjacent vertices \( x, y \) such that \( x \)’s distance is known for certain and \( y \)’s is not.
  - Vertex \( x \) need not necessarily be distinct from \( s \).
Dijkstra’s Algorithm

Dijkstra’s Algorithm – Why it works correctly?

- But $y$ must be distinct from $v$.
  - If $y$ and $v$ were the same vertex, so that $v$ was a neighbor of $x$, then we would already have found the shorter path to $v$ when we explored the neighbor of $x$ in step 3.
  - We accordingly have revised our estimate of $v$’s distance downward.
  - Since this hasn’t happened, $y$ and $v$ must be distinct nodes.

- $y$’s current estimated distance will be at most equal to its distance from $s$ along the path because that distance is calculated in step 3.

- Since all distance along the path are necessarily less than the current estimated distance to $v$, $y$’s estimated distance must be less than $v$’s.

- Contradict the hypothesis that $v$ is the vertex with the shortest estimated distance.

- Hence there is no path to vertex $v$ with length less than $v$’s current estimated distance, so we can safely mark that distance as being certain.