Università
degli Studi
di Padova

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

INFORMATION ENGINEERING DEPARTMENT

COMPUTER ENGINEERING - AI & ROBOTICS

Learning form networks 2023/2024

Francesco Crisci 2076739

Basic notions

Graph

We define a Graph as G=(V,E), where V is the set of vertices, or nodes, and E is the collection of edges. A graph can be:

- directed if every edge $(u, v) \in E$ is an ordered pair $(u \to v)$, sometimes called arc;
- undirected if every edge $(u, v) \in E$ is an undordered pair (u-v).

E is a collection and not a set since there could be multiple edges u,v. We can have a self loop,i.e, an edge (u,u). We call a graph simple if there are no multiple edges and self loops. A weighted graph is a graph with edger and/or vertices with weights.
Let $e = (u, v) \in E$ an edge of a graph G, then e is incident to u and to v; u and v are adjacent.

Degree

The degree d of u in an inderected graph is the number of adjacent vertices or number of incident edges. In the case of a directed graph we define the outdegree of a node as the number of leaving arcs, the indegree as the number of entering arcs and the degree as the sum of the outdegree and the indegree.

Paths

A path is a sequence $u_1, u_2, ..., u_k$ of vertices with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.
Single path: is a sequence $u_1, ..., u_k$ of vertices all distinct with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.

Cycles

A cycle is a sequence $u_1, ..., u_k = u_1$ of vertics with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.
Simple cycle: is a sequence $u_1, .., u_k = u_1$ of vertices all distinct with $(u_i, u_{i+1}) \in E$ for all $1 \leq i < k$.

Subgraph

$G' = (V', E')$ is a subgraph of G=(V,E) if $V' \subseteq V, E' \subseteq E$ and $\forall (u, v) \in E' : u, v \in V'$

Connected and disconnected graphs

G=(V,E) is connected if $\forall u, v \in V$ there exists a path in G starting in u and ending in v.
G=(V,E) is disconnected if there exists $u, v \in V$ such that there is no path in G starting in u and ending in v.

Connected components

The connected components of G(V,E) are a partition of G in subgraphs $\{G_1, ..., G_k\}$, with $G_i = (V_i, E_i)$ such that

- $G_i = (V_i, E_i)$ is connected for all $1 \leq i \leq k$
- $V = V_1 \cup V_2 \cup ... \cup V_k$ (partition of V)
- $E = E_1 \cup E_2 \cup ... \cup E_k$ (partition of E)
- $\forall i \neq j$ there are no edges in E between V_i and V_j

$\{G_i : 1 \leq i \leq k\}$ are maximal connected subgraphs; if G is connected then k=1 in the definition above. Th partition of G in connected components is unique.

Trees

A tree is a graph G=(V,E) connected and without simple cycles. A rooted tree is a graph G=(V,E) such that:

- there exists a root $r \in V$
- $\forall u \in V$, with $u \neq r$, there exists a unique vertex $p(u) \in V$ parent of u
- $E = \{(u, p(u)) : u \in V, u \neq r\}$
- $\forall u \in V$ walking from parent to parent we reach r.

A rooted tree and a free tree are equivalent.

Spanning tree

It is a connected spanning subgraph of G without cycles. If G is not connected then we cannot have a spanning tree.
We can define a spanning forest as a spanning subgraph without cycles.

Graph properties

Let G=(V,E) be a simple and undirected graph with $|V| = n$ and $|E| = m$. Then the following properties holds:

- $$\sum_{v \in V} \text{degree}(v) = 2m$$

  We can prove it by showing that every edge is counted twice in the sum.
- $m \leq \binom{n}{2}$ ($\Rightarrow m \in O(n^2)$). We can prove it by showing that G is simple: E is a subset of the $\binom{n}{2}$ possible pairs of vertices.

- if G is a tree then m=n-1. To prove it we have to consider G as a rooted tree. Then E contains the relations parent-child, that are n-1.
- If G is connected then $m \geq n-1$.
  Proof: Consider the following loop:
  While($\exists$ cycle C) do remove an edge of C from G. Then:
  - at the end of every iteration: G is connected
  - at the end of the loop: G is connected and without cycles, that is G is a tree with $m' = n-1 \leq m$ edges.
- If G is a forest, then $m \leq n-1$
  Proof: Consider the following loop:
  while(G is not connected) do:
  add an edge between two connected components G_1, G_2 of G.
  Then:
  - every added edge does not create a cycle in G;
  - at the end of the loop: G is connected and without cycles, that is G is a tree with $m' = n-1 \geq m$ arcs.

Graph representations

Let G=(V,E) be a graph with $|V| = n$ and $|E| = m$. The simplest representation of G uses the following basic data structure:

- Vertex list L_V : every node of the list contains all relevant information for a distinct $v \in V$. If $V = \{1, 2, ..., n\}, L_V$ could be an array
- Edge list L_E : every node of the list contains all the relevant information for a distinct $e = (u, v) \in E$, including pointers to u and v.

Adjacency list representation

For every vertex $v \in V$ there is a list l(v) of pointers to the edges incident to v. This representation is commonly used for the following reasons:

- it allows to represent G with space linear in the size of G: $\Theta(n+m)$
- it allows sequential access to the neighbors of a vertex v, in time linear in the degree of v.

Adjacency matrix representation

We have an nxn matrix A such that rows and columns are in 1 to 1 correspondence with G vertices, with

$$A[i_1, i_2] = \begin{cases} \text{null} & \text{if} (i_1, i_2) \notin E \\ \text{pointer to } e = (i_1, i_2) \in L_E & \text{if such edge exists} \end{cases}$$

Note that:

- such structure requires the vertices to be represented by integers and allows O(1) access to and edge and its information
- the adjacency matrix requires space $\Theta(n^2)$ which can be superlinear in the size of G. Therefore, the representation is used mostly for dense graphs.

Graph traversal

Systematic exploration of G starting from a vertex s and visiting all vertices. We have:

- Breadth-first Search: after visiting a vertex, visit all its neighbors before moving to the neighbors of its neighbors.
- Depth-first search: after visiting a node, visit one of its neighbors, then a neighbor of the neighbor.

BFS

It is an iterative algorithm that, starting from vertex s, visits all the vertices and the edges in the connected component C_s containing s, touching all the vertices and the edges of C_s and partitioning the vertices in levels L_i based on their distance i from s. Every vertex $v \in V$ has a field v.ID with v.ID=1 if v hase been visited, and v.ID=0 otherwise. Every edge $e \in E$ has a field e.label with e.label = null if e has not bee labeld yet, or it has a label between DISCOVERY EDGE or CROSS EDGE.

Algorithm

Input: undirected graph G=(V,E), vertex $s \in V$ not visited.
Output: all the vertices of C_s are visited and all edges C_s labeled as DISCOVERY or CROSS EDGE

```
visit s; s.ID← 1;
create list L_0 containing only s; i ← 0;
while (!L_i.isEmpty())do
        create empty L_{i+1}
        forall v ∈ L_i
                forall e ∈ G.incidentEdges(v)
                        if(e.lameb == null)
                                w ← G.opposite(v, e);
                                 if (w.ID==0)
                                        e.label ←DISCOVERY EDGE;
                                        visit w
                                        w.ID ← 1
                                        insert w in L_{i+1}
                                e.labelCROSS EDGE;        i ← i + 1
```

Proposition: Consider the execution of BFS(G,s). Assume that at the beginning, no vertex of C_s is visited and no edge of C_s is labeled. At the end of the execution:

1. all vertices of C_s are visited and all edges of C_s are labeled as DISCOVERY or CROSS EDGE
2. the DISCOVERY EDGES are a spanning tree T of C_s rooted in s, called BFS tree

3. $\forall v \in L_i$ the path in T from s to v has i edges and i is the minimum number of edges among any path from s to v in G; i is the distance between s and v.

4. if $(u, v) \in E$ and $(u, v) \notin T (\Rightarrow (u, v)$ is a CROSS EDGE) then the indices of u and v differ by at most 1.(I NEED TO ADD THE PROOF AFTERWARDS CAUSE IT IS NOT ON THE SLIDES)

Theorem

Consider the execution of BFS(G,s). Assume that at the beginning, no vertex of C_s is visited and no edge of C_s is labeled. The complexity of BFS(G,s) is $\Theta(m_s)$, where m_s is the number of edges of C_s.
Proof: Define n_s as the number of vertices in C_s and m_s as the number of edges in C_s. Assume that L_i is a list $\forall i$.

- $\forall v \in V$ with $v \in C_s$, there is exactly 1 iteration of the first forall loop that considers v and exactly degree(v) iterations of the second forall loop where an edge of the type $v,)$ is considered.
- every iteration of the second forall loop takes time $\Theta(1)$
- every operation for L_1 takes time $\Theta(1)$

$\Rightarrow$ the complexity of $BFS(G, s) \in \Theta(\sum_{v \in C_s}(v)) \in \Theta(m_s)$.

Corollary

If G=(V,E) is connected, BFS(G,s) has complexity $\Theta(|E|) \forall s \in V$.

Visiting the whole Graph

Note that BFS(G,s) visits only the connected component C_s of s. To visit the whole graph we can use the following approach:
forall $v \in V$ $v.ID \leftarrow 0$;
forall $v \in V$
    if(v.ID==0)
        BFS(G,v)
This algorithms is $\Theta(n + m)$, and we assume that there is an iterator on L_V that allows to access the next vertex in time O(1). We'll now show an analysis:
Assume c be the number of connected components of G. note that:

- the second forall lop: the call BFS(G,v) is made exactly c times, each time on a different connected component.
- let m_j be the number of edges in the j-th connected component of G, for $1 \leq j \leq c$. Note that $\sum_{j=1}^{c} m_j = m = |E|$

The cos of all the call BFS(G,v) is $\Theta(\sum_{j=1}^{c} m_j) \in \Theta(m)$. Other operations have complexity $\Theta(n) \Rightarrow$ the overall complexity is $\Theta(n + m)$.

BFS: connectivity

Input: Graph=(V,E)
Output: Number of connected components of G and each connected component with a different ID values for its vertices. Let BFS(G,v,k) be a modified version of BFS(G,v), where the instruction $w.ID \leftarrow 1$ is substituted by $w.ID \leftarrow k$. The following algorithm computes the number of connected components of G and assigns to all vertices of each connected component the same value ID.
for $v \leftarrow 1$ to n $v.ID \leftarrow 0$;
$k \leftarrow 0$;
for $v \leftarrow 1$ to n
    if(v.ID==0)
        $k \leftarrow k + 1$;
        BFS(G,v,K)
return k;
The complexity is the same as before due to the same considerations.
The following propositions can be easily proven: Given G=(V,E), with $|V| = n$ and $|E| = m$, the following problems can be solved in time $O(n + m)$ using the BFS:

- test if G is connected;
- find the connected components of G;
- find a spanning tree of G, if G is conected:
  Proof: run BFS(G,1), the BFS tree is a spanning tree.
- find a shortest path between vertices s and t, if it exists:
  Proof: Start BFS(G,s) at the end: if t has been visited then: the length of the shortest path between s and t is i, if $t \in L_i$
- find a cycle, if it exists:
  Proof: run BFS(G,1) at the end: consider a cross edge if G is conected, otherwise run the algorithm above for each connected component.

Depth first search

- Recursive algorithm: starting from vertex s, visits all the vertices in the connected component of C_s of s and labeling all edges of C_s
- Every vertex $v \in V$ has a field v.ID, with v.ID=1 if v has been visited, 0 otherwise
- Every edges $e \in E$ has a field e.label with e.label=null if e has not been labeled, and one between the labels DISCOVERY EDGE or BACK EDGE otherwise.
- During the execution of the algorithm, we say that a vertex u is discoverable from v if there exists a path from u to v made of vertices not already visited.

The algorithm is:
input: undirected graph G=(V,E), vertex $v \in V$ not visited
output: all vertices discoverable from v visited, and all edges incident to them labeled as DISCOVERY or BACK EDGE.
visit v; $v.ID \leftarrow 1$

forall $e \in G.incidentEdges(v)$
    if(e.label==null)
        $w \leftarrow G : opposite(v, e)$;
        if(w.ID==0)
            $e.label \leftarrow$ DISCOVERY EDGE;
            DFS(G,w);
         else $e.label \leftarrow$ BACK EDGE;

For this implementtion we make the following assumption:

- incidentEdges(v) returns an iterator on the neighbors of v, and the forall loop access them iteratively, in time $\Theta(1)$ for each of them
- opposite(v,e) returns the vertex of e opposite to v, in time $\Theta(1)$
- when the first call DFS(G,s) is made, all vertices are not visited and all edges are not labeled.

Proposition

Assume to execute DFS(G,s) and that, at the beginning, the vertices and edges of C_s are not visited and not labeled. At the end of execution: all vertices of C_s are visited and all edges of C_s are labeled as DISCOVERY or BACK EDGE; the DISCOVERY EDGES are a spanning tree T of C_s rooted in S.

Theorem

Assume to execute DFS(G,S and that, at the beginning, the vertices and edges of C_s are not visited and not labeled. The complexity of DFS(G,s) is $\Theta(m_s)$, where m_s is the number of edges of C_s.

Proposition

Given G=(V,e), with $|V| = n$ and $|E| = m$, the following problems can be solved in time O(n+m) using DFS:

- test if G is connected;
- find the connected components of G;
- find a spanning tree of G, if G is connected;
- find a path between s and t, if it exists;
- find a cycle if it exists.

Node analytics

The easiest score we can derive from a network is the node degree: degree(u). The complexity to compute the node degree for all $v \in V$ is $O(n+m)$. Several scores have been proposed to measure the centrality of a node in a network. We'll see closeness centrality and betweenness centrality.

Closeness centrality

The intuition is that a ode is central if it is fairly close to the other nodes in the network.
Let G=(V,E) be a connected, undirected graph, we define the closeness centrality of v,c(v), as the average distance of v to the other nodes in the graph g:

$$c(v) = \frac{n-1}{\sum_{u \neq v, u \in V} d(u,v)}$$

Since d(v,v)=0 for all $v \in V$, we can rewrite it as

$$c(v) = \frac{n-1}{\sum_{u \in V} d(v,u)}$$

Distance

Given $u, v \in V$, let d(u,v) the distance between u and v. If G is unweighted, d(u,v) is the number of edges in a shortest path between u and v. In case G is weighted, then:

- edge weight function $w : E \to \mathbb{R}^+ (w(e) \in \mathbb{R}^+)$
- length of path $u_1, ..., u_k$ is:
$$\sum_{i=1}^{k-1} w(u_i, u_{i+1})$$
- d(u,v) is the minimum length of a path from u to v
- a path of length d(u,v) between u and v is a shortest path.

Computing closeness centrality: One node

Given $v \in V$, how can we compute its closeness centrality c(v)? We need the values d(v,u)$\forall u \in V$.
Definition: Given a simple, undirected graph G=(V,E) and a vertex $v \in V$, the Single-Source Shortest Paths (SSSP) problem requires to find all the distances between v and the other vertices in V(and the relative shortest paths).
Let distBFS(G,v) be BFS(G,v) with the following changes:

- every node $u \in V$ has a field u.distance
- initially:u.distance has value 0,$\forall u \in V$
- when node w is visited and then inserted in list L_{i+1}, we set w.distance to i+1. That is, the instruction "visit w" is substituted with "$w.distance \leftarrow i+1$"

When distBFS(G,v) terminates, u.distance=d(v,u). The algorithm is the following:
Input: unweighted graph G=(V,E) with $|V| = n$ and $|E| = m; v \in V$.
Output: closeness centrality c_v of v.
distBFS(G,v);
$sum_v \leftarrow 0$;
forall $u \in V$:
    $sum_v \leftarrow sum_v + u.distance$;
return $(n-1)/sum_v$;
The complexity is $\Theta(m)$
Now to compute the closeness centrality of all the nodes we do the following:
$forall\ v \in V$:
    $c(v) \leftarrow$ ClosnessCentrality(G,v);
return values c(v);
The complexity is then $\Theta(nm)$
In case of computing the closeness centrality weighted graph, Let G be the edge weighted: given $v \in V$, how can we compute its closeness centrality c(v)? We need the values d(v,u)$\forall u \in V$... By using Dijkstra algorithm with a priority queue we can solve the SSSP problem. The complexity of the algorithm depends on how the priority queue is implemented, if we use an heap we have $\Theta(min\{n^2, (n+m)logn\})$, if we use a Fibonacci heap, then $\Theta(m + nlogn)$. Let ClosenessCentralityW(G,v) be the algorithm, based on Dijkstra algorithm, to compute the closeness centrality for node v. Analogously to before, the node centralities of all nodes in G are computed as follows:
Input: weighted graph G=(V,E,w) with $|V| = n$, $|E| = m.w : E \to \mathbb{R}^+$.
Output: closeness centrality c(v) for all $v \in V$.
for all $v \in V$: $c(v) \leftarrow$ ClosenessCentralityW(G,v);
return values c(v);
The complexity is $\Theta(n(m + nlogn))$ if the Dijkstra algorithm is based on Fibonacci heaps.
We need to calculate all the values d(v,u) $\forall u, v \in V$.
Definition: Given a simple graph G=(V,E) and a vertex $v \in V$, the All-Pairs Shortest Path (APSP) problem requires to find all the distances d(u,v) for all pairs $u, v \in V$(and the relative shortest paths).
This problem can be solvd by the Floyd-Warshall algorithm but has a complexity of $\Theta(n^3)$. There is also the Johnson's algorithm in time $\Theta(n^2 logn + nm)$, which is the same complexity as dijkstra with Fibonacci. Even though this algorithms are not improving the performance compared to Dijkstra, they can be used when negative weights are present.
These algorithms are impractical for large networks, so we need to approximate the Closeness centrality.

Approximating Closeness Centralities

Consider one node v:

- to compute c(v) exactly, we need d(v,u) for all $u \in V$ to compute $\sum_{u \in V} d(v,u)$.

- What about using only some terms d(v,u)? If we use k such terms, then we need to rescale the sum by a factor n/k.

What about c(v) for all nodes $v \in V$?

- We can use the terms d(v,u) for the same vertices u.
- how can we compute such distances d(u,v)? Solve the SSSP problem from u. To this purpose we can use the Eppstein-Wang algorithm.

Eppstein-Wang algorithm

Input: weighted/unweighted graph G=(V,E) with $|V| = n, |E| = m; k \in \mathbb{N}$
Output: approximation $\hat{c}(v)$ of c(v) for all $v \in V$
$sum_v \leftarrow 0$ for all $v \in V$;
for $i \leftarrow 1$ to k:
    $v_i \leftarrow$ random vertex chosen uniformly at random from V;
    Solve SSSP problem with source v_i
    forall $v \in V$:
        $sum_v \leftarrow sum_v + d(v_i, v)$;
forall $v \in V$:
    $\hat{c}(v) \leftarrow 1/(\frac{n \dot{s} um_v}{k(n-1)})$;
return values $\hat{c}(v)$;
For simplicity, we are going to consider the inverse centrality estimator $\frac{1}{\hat{c}(v)}$ and the inverse centrality $\frac{1}{c(v)}$, since $\mathbb{E}[\frac{1}{\hat{c}(v)}] = \frac{1}{c(v)}$.
Proof:
Let X_i be the random variable whose value is $d(v, v_i).\frac{n}{n-1}$ (v_i is the vertex chosen in the i-th iteration). Then:

$$\frac{1}{\hat{c}(v)} = \frac{1}{k}\sum_{i=1}^{k} X_i$$

The expected value then becomes

$$\mathbb{E}[\frac{1}{\hat{c}(v)}] = \mathbb{E}[\frac{1}{k}\sum_{i=1}^{k} X_i] = \frac{1}{k}\sum_{i=1}^{k}[X_i]$$

Since in each iteration i we pick v_i uniformly at random from V, each vertex has probability 1/n to be picked. Therefore, $\forall 1 \leq i \leq k$:

$$\mathbb{E}[X_i] = \frac{1}{n}d(1,v)\frac{n}{n-1} + ... + \frac{1}{n}d(n,v)\frac{n}{n-1} = \sum_{u \in V}\frac{d(u,v)}{n-1} = \frac{1}{n-1}(\sum_{u \in V} d(u,v))$$

Then:

$$\mathbb{E}[\frac{1}{\hat{c}(v)}] = \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}[X_i] = \frac{1}{k}\sum_{i=1}^{k}(\frac{1}{n-1}\sum_{u \in V} d(u,v))$$

$$= \frac{1}{k}\frac{1}{n-1}k(\sum_{u \in V} d(u,v)) = \frac{1}{n-1}\sum_{u \in V} d(u,v) = \frac{1}{c(v)}$$

We are now going to prove that the estimates $\frac{1}{\hat{c}(v)}$ are close to their expectations $\frac{1}{c(v)}$ if the number k of iterations in the algorithm is large enough. We use the following concentration result:

Hoefdding's inequality: Let $X_1, ..., X_k$ be independent random variables, with $a_i \leq x_i \leq b_i$ for all $i \in \{1, ..., k\}$ and let $\mu = \mathbb{E}[\frac{\sum_{i=1}^{k} X_i}{k}]$. Then for $\epsilon > 0$:

$$\mathbb{P}\left[|\frac{\sum_{i=1}^{k} X_i}{k} - \mu| \geq \epsilon\right] \leq 2e^{-2l^2\epsilon^2/\sum_{i=1}^{k}(b_i - a_i)^2}$$

Let $\Delta(G) = max_{u,v \in V} d(u, v)$ be the diameter of a graph, then:
Proposition: Let $\epsilon > 0$ and $\delta \in (0, 1)$ be constants. If $k \geq \frac{1}{2\epsilon^2}(log\frac{2n}{\delta})(\frac{n}{n-1})^2$ then the inverse centrality estimator $\frac{1}{\hat{c}(v)}$ from ApproximateClosenessCentralities(G,k) is within an additive factor $\epsilon\Delta(G)$ of $1/c(v)$ for all vertices $\in V$ with probability $\geq 1 - \delta$. Note that in several real-world networks $\Delta(G) \in O(logn)$.
Proof: Consider an arbitrary vertex $v \in V$. Let X_i be the random variable whose value is $d(v, v_i)\frac{n}{n-1}$, with v_i the vertex chosen in the i-th iteration. As before: $\frac{1}{\hat{c}(v)} = \frac{1}{k}\sum_{i=1}^{k} X_i$. We already proved that $\mathbb{E}[\frac{1}{\hat{c}(v)}] = \frac{1}{c(v)}$. Note that $\forall i, 1 \leq i \leq k : 0 \leq X_i \leq \Delta(g)\frac{n}{n-1}$. We can then apply Hoeffding's inequality with:

$$\mu = \frac{1}{c(v)}, a_i = 0, b_i = \frac{n}{n-1}\Delta(G), \epsilon_H = \epsilon\Delta(G)$$

We obtain the following:

$$Pr[|\frac{1}{\hat{c}(v)} - \frac{1}{c(v)}| > \epsilon\Delta(G)] = Pr[|\frac{\sum_{i=1}^{k} X_i}{k} - \mu| > \epsilon_H] \leq 2e^{-2k^2\epsilon_H^2/\sum_{i=1}^{k}(b_i - a_i)^2}$$

$$= 2e^{-2k^2\epsilon^2/k(\frac{n}{n-1}\Delta(G))^2} = 2e^{-\frac{2k\epsilon^2}{(\frac{n}{n-1})^2}} \leq 2e^{-2\frac{1}{\epsilon^2}lg(\frac{2n}{\delta})(\frac{n}{n-1})^2\frac{\epsilon^2}{(\frac{n}{n-1})^2}} = 2\frac{\delta}{2n}$$

$$= \frac{\delta}{n}$$

$\forall v \in V$, let $E_v = |\frac{1}{\hat{c}(v)} - \frac{1}{c(v)}| > \epsilon\Delta(G)$. We just proved that $Pr[E_v] \leq \frac{\delta}{n} \forall v \in V$. Let consider now the event: $(*)\exists$ at least one vertex v for which we have $|\frac{1}{\hat{c}(v)} - \frac{1}{c(v)}| > \epsilon\Delta(G) = \cup_{v \in V} E_v$. Now:

$$Pr[\cup_{v \in V} E_v] \leq \sum_{v \in V} Pr[E_v] \leq n\frac{\delta}{n} = \delta$$

Then Pr[all vertices v:$|\frac{1}{\hat{c}(v)} - \frac{1}{c(v)} \leq \epsilon\Delta(G)]$ = 1-Pr[$(*)$] $\geq 1 - \delta$.

Corollary

If $k \in \Theta(\frac{logn}{\epsilon^2})$ for a costant $\epsilon > 0$, then the inverse centrality estimator $\frac{1}{\hat{c}(v)}$ from Approximate-ClosenessCentralities(G,k) is within an additive factor $\epsilon\Delta(G)$ of $\frac{1}{c(v)}$ for all vertices $v \in V$ with high probability, i.e. with probability $\geq 1 - 1/n$.

Complexity of Eppstein-Wang

The solution of k SSSP problems are needed. If G is unweighted then $\Theta(km)$, if G is weighted $\Theta(k(m + nlogn))$ are the complexities. By fixing $k \in \Theta(\frac{logn}{\epsilon^2})$ as in the corollary, the complexity is $\Theta(\frac{mlogn}{\epsilon^2})$ in the case of unweighted graphs, or $\Theta(\frac{logn}{\epsilon^2}(m + nlogn))$ in the case of weighted graph. This algorithm provides rogorous guarantees and useful results, the error of the estimates may be large when the distribution of distances in G is skewed (e.g, the diameter $\Delta(G)$ is large).

Chechik-Cohen-Kaplan algorithm

The basic idea is to use sampling to choose a sample S of k nodes for which the SSSP problem is solved and use the corresponding distance to estimate the centralities for all nodes. The main difference with Eppstein-Wang is that S is not picked by choosing uniformily at random:

- for each vertex $v \in V$, compute a probability p_v, then include v in S with probability p_v independently of all other events.
- p_v's computation:
  - first draw a small sample S_0 and compute $W_s = \sum_{v \in V} d(s, v)$ for all $s \in S_0$
  - $p_v = max\{1/n, max_{s \in S_0} \frac{d(s,v)}{W_s}\}$

Provides better guarantees than Eppstein-Wang since it does not depend on the diameter of G.

Closeness for disconnected graphs

Let G=(V,E) be an undirected graph. Let $C_v$be the connected component of v,i.e, of the set of vertices reachable from v. Let $n_v = |C_v|$. We define the **Lin's index** for $v \in V$ as:

$$c(v) = \frac{(n_v - 1)^2}{(n-1)\sum_{u \in C_v} d(v, u)}$$

Betweenness Centrality

a node is central if it appears on several shortest paths in the network. Let G=(V,E) be an undirected graph with $|V| = n$. Let $\sigma_{s,t}$ be the number of shortest paths from node s to node t. Let $\sigma_{s,t}(v)$ be the number of shortest paths from node s to node t that pass through node v. We define the betweenness centrality b(v) of v as:

$$b(v) = \sum_{s,t \in V : s \neq v \neq t} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$$

Note that b(v) can be > 1.

Definition of normalized betweennes centrality

given a node v, the normalized betweenness centrality b(v) of v is

$$b(v) = \underbrace{\frac{1}{n(n-1)}}_{\text{normalization factor}} \sum_{s,t \in V : s \neq v \neq t} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$$

Note: given G, $n = |V|$ is fixed, then the two definitions are equivalent. $0 \leq b(v) < 1$.
To compute the b(v), we need to compute for each $s, t \in V (s \neq v \neq t) : \sigma_{s,t}(v)$ and $\sigma_{s,t}$. To achieve so we can use Brandes's algorithm. The idea is to use an augmented BFS, which works also for weighted graphs. The complexities are O(nm) if the graph is unweighted and =(n(m+nlogn)) if the graph is weighted. Which is still unfeasible for large graphs.

Approximating Betweenness centrality

Input: weighted/unweighted graph G, with $|V| = n, |E| = m; k \in \mathbb{N}$.
Output: approximation $\hat{b}(v)$ of b(v) for all $v \in V$.
$\hat{b}(v) \leftarrow 0 \forall v \in V$;
for $i \leftarrow 1$ to k:
    $(s,t) \leftarrow$uniform vertex pair from V;
    $P_{s,t} \leftarrow$all shortest paths from s to t;
    $\pi_i \leftarrow$shortest path chosen uniformly at random from $P_{s,t}$;
    forall $v \in \pi_i$:
        $if v \neq s$ and $v \neq t$: $\hat{b}(v) \leftarrow \hat{b}(v) + 1/k$;
return values $\hat{b}(v)$;
Note that $P_{s,t}$ can be computed by a modified SSSP algorithm that keeps track of all shortest paths. The SSSP algorithm starts from s and can stop as soon ass all shortest paths to t are found.

Proposition

For any $v \in V$ we have that

$$\mathbb{E}[\hat{b}(v)] = b(v)$$

Proof: Consider $v \in V$. Let $X_i, 1 \leq i \leq k$ be a random variable with $X_i = \begin{cases} 1 \text{ it v is in } \pi \\ 0 \text{ otherwise} \end{cases}$.
Then $\hat{b}(v) = \frac{1}{k}\sum_{i=1}^{k} X_i$. Then:

$$\mathbb{E}[\hat{b}(v)] = \mathbb{E}[\frac{1}{k}\sum_{i=1}^{k} X_i] = \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}[X_i]$$

By the low of total probability:
$= \sum_{s\neq v\neq t} Pr(\text{v is in } \pi | \text{s,t are selected}) \text{Pr(s,t are selected)}$
$= \frac{1}{n(n-1)}\sum_{s\neq v\neq t} Pr(\text{v is in } \pi | \text{s,t are selected})$.
Since π_i is selected uniformly at random from $P_{s,t}$: Pr(v is in π_i|s,t are selected)=
$= \frac{\text{number of shortest paths in } P_{s,t} \text{containing v}}{|P_{s,t}|} = \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$. Therefore:

$$\mathbb{E}[\hat{b}(v)] = \frac{1}{k}\sum_{i=1}^{k}\mathbb{E}[X_i] = \frac{1}{k}k\mathbb{E}[X_i] = \mathbb{E}[X_i]$$

$$= \frac{1}{n(n-1)}\sum_{s\neq v\neq t}\frac{\sigma_{s,t}(v)}{\sigma_{s,t}} = b(v).$$

How many samples are needed to have $\hat{b}(v)$ close to b(v)? We use the vertex diameter VD(G): maximum number of vertices among all shortest paths in G. (VD(G)-1 is equal to the diameter of G if G is unweighted).
Riondato,Karnaropoulos:
if $k \geq \frac{2}{\epsilon^2}(\lfloor log_2(VD(G)-1)\rfloor + ln(1/\delta))$, then

$$Pr[\exists v \in V s.t. |\hat{b}(v) - b(v)| > \epsilon] < \delta$$

The proof is based on the VC dimension of shortest paths.

Clustering coefficients

Local clustering coefficient

The idea is that it measures the degree to which nodes in a neighborhood/graph tend to cluster together. We want to count the number of triangles involving a node or in the graph. The number of triangles points out if a node belongs to something(local clustering coefficient) or in the graph(clustering coefficient). Let G be a weighted or unweighted graph. Given $v \in V$:

- let $N(v)$ be the set of neighbors of v in G: $N(v) = \{u : (v, u) \in E\}$
- let deg(v)=$|N(v)|$ be the degree of v

We define the local cluster coefficient as:
For a node $v \in V$, its local clustering coefficient cc(v) of v is:

$$cc(v) = \frac{|\{(u_1, u_2) : u_1 \in N(v), u_2 \in N(v), (u_1, u_2) \in E\}|}{deg(v)(deg(v) - 1)}$$

Note that cc(v) is the fraction of triangles containing v among all the potential triangles containing v.

Clustering coefficient

Given a graph G, weighted or unweighted, with $|V| = n$. We define the clustering coefficient as:
The clustering coefficient cc(G) of G is:

$$cc(G) = \frac{|\{(u, v, z) : (u, v) \in E, (v, z) \in E, (z, u) \in E\}|}{6\binom{n}{3}}$$

Sometimes the average local clustering coefficient is used as well, which is defined as follows:
The average local clustering coefficient $agg_{lcc}(G)$ of G is:

$$avg_{lcc}(G) = \frac{1}{n}\sum_{v \in V} cc(v)$$

How do we compute the local clustering coefficient for a single node $v \in V$?
Input: graph G with $|V| = n$ and $|E| = m; v \in V$.
Output: clustering coefficient cc_v of v.
$num_t \leftarrow 0$;
forall $u_1 \in N(v)$
    forall $u_2 \in N(v)$

if $u_1 \neq u_2$ and $(u_1, u_2) \in E$
$num_t \leftarrow num_t + 1$;
$deg_v \leftarrow |N(v)|$;
return $\frac{num_t}{deg_v(deg_v-1)}$;
The complexity of this algorithm is $\Theta(deg(v)^2)$
To compute all the global clustering coefficients for all the nodes in the network we use the following algorithm:
Input: graph G with $|V| = n$ and $|E| = m$
Output: clustering coefficient cc_v of v for each $v \in V$
forall $v \in V$
$cc_V \leftarrow LCC(G, v)$;
return values cc_v;
Let $\delta(G)$ be the maximum degree of nodes in G: $\delta(G) = max_{v \in V} deg(v)$. Then the complexity of the algorithm is $O(m\delta(G))$.

Approximating the local clustering coefficients

WE want a limited number of sequential scans of the data stored in the secondary memory. The idea is that for every edge $(u, v) \in E$, the number of triangles containing (u, v) is $|N(u) \cap N(v)|$, and that the number of triangles containing $v \in V$ is

$$\frac{1}{2} \sum_{u \in N(v)} |N(u) \cap N(v)|$$

The last step of the idea is to build on basic building block of estimating the cardinality of the intersection of two sets.

Estimating the intersection

Let A,B b two sets with $A, B \subseteq U$, where U is a universe of elements. Consider a random permutation π of the elements of U, and let $\pi(A)$ the restriction of π to the elements of A. Let $min(\pi(A))$ be the first element of $\pi(A)$, then

$$Pr[min(\pi(A)) = min(\pi(B))] = \underbrace{\frac{|A \cap B|}{A \cup B}}_{J(A,b) \text{ is the Jaccard coefficient}}$$

Proof: Since π is a random permutation $\min(\pi(A))$ is an element of A chosen uniformly at random. Since π is a random permutation, the firs element of $A \cup B$ in π is chosen uniformly at random. Then:

$$Pr[min(\pi(A)) = min(\pi(B))] = Pr[\text{The first element of } A \cup B \; in \; \pi \; comes \; from \; A \cap B] = \frac{|A \cap B|}{A \cup B}$$

The algorithm to estimate the intersection is the following:
Input: sets $A, B \subseteq U; k \in \mathbb{N}^+$
Output: estimate $\hat{I}$ of $|A \cap B|$
$count \leftarrow 0$;
forall $i \leftarrow 1$ to k
$\pi_i \leftarrow$ random permutation of U;

$if\ min(\pi_i(A)) == min(\pi_i(B))$ then $count \leftarrow count + 1$;
$\hat{I} \leftarrow \frac{count}{count+k}(|A| + |B|)$;
return $\hat{I}$;
In practice: pseudo-random hash functions $h : U \rightarrow \{0, ..., |U| - 1\}$ are used.

Proposition

$$\mathbb{E}[\frac{count}{k}] = J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Proof: For i=1,...,k, let the random variable $X_i = \begin{cases} 1 \text{ if } min(\pi_i(A)) = min(\pi_i(B)) \\ 0 \text{ otherwise} \end{cases}$. Then at the end of the algorithm, count=

$$\sum_{i=1}^{k} X_i \Rightarrow \mathbb{E}[\frac{count}{k}] = \frac{1}{k}\mathbb{E}[count]$$

$$= \frac{1}{k}\sum_{i=1}^{k} \mathbb{E}[X_i]$$

We know that $\mathbb{E}[X_i] = Pr[X_i = 1] = Pr[min(\pi_i(A)) = min(\pi_i(B))] = J(A, B)$. Then we have:

$$\mathbb{E}[\frac{count}{k}] = \frac{1}{k}\sum_{i=1}^{k} J(A, B) = J(A, B)$$

Proposition

For any constant $\epsilon \in (0, 1)$:

$$Pr[|\frac{count}{k} - J(A, B)| \geq \epsilon J(A, B)] \leq 2e^{-\frac{\epsilon^2}{3}kJ(A,B)}$$

To prove this Proposition we need the Cherroff bound:

Chernoff Bound

Let $X_1, ..., X_n$ be 0-1 random variables. Let $X = \sum_{i=1}^{n} X_i$. Let $\mu = \mathbb{E}[X]$. Then for every $0 \leq \delta \leq 1$:

- $Pr(X \geq (1 + \delta)\mu) \leq e^{-\frac{\delta^2\mu}{2}}$
- $Pr(X \leq (1 - \delta)\mu) \leq e^{-\frac{\delta^2\mu}{3}}$

Proof Proposition: Let $X_i = \begin{cases} 1 \text{ if } min(\pi_i(A)) = min(\pi_i(B)) \\ 0 \text{ otherwise} \end{cases}, 1 \leq i \leq k$. As before: $X = \sum_{i=1}^{k} X_i \Rightarrow$ count=X$\Rightarrow \frac{count}{k} = X/k$ and $\mathbb{E}[count/k] = J(A, B) \Rightarrow \mathbb{E}[count] = kJ(A, B)$.
From Chernoff bound we have:

$$Pr(|\frac{count}{k} - J(A, B)| \geq \epsilon J(A, B)) = Pr(|count - kJ(A, B)| \geq \epsilon kJ(A, B))$$

$$\leq Pr(\underbrace{count}_{X} \geq (1+\underbrace{\epsilon}_{\delta})\underbrace{kJ(A,B)}_{\mu=\mathbb{E}[X]}) + Pr(\underbrace{count}_{X} \leq (i-\underbrace{\epsilon}_{\delta})\underbrace{kJ(A,B)}_{\mathbb{E}[X]})$$

$$\leq e^{-\frac{\epsilon^2}{3}kJ(A,B)} + e^{-\frac{\epsilon^2}{3}kJ(A,B)}$$

$$= 2e^{-\frac{\epsilon^2}{3}kJ(A,B)}$$

Proposition

$\hat{I}$ is an estimate of $|A \cap B|$.
Proof: Previously: count/k is an estimate of $J(A,B) = \frac{|A\cap B|}{|A\cup B|} \Rightarrow \frac{count}{k} \approx \frac{|A\cap B|}{|A\cup B|}$
$= \frac{|A\cap B|}{|A|+|B|-|A\cap B|} \Leftrightarrow count(|A|+|B|-|A\cap B|) \approx k|A\cap B|$
$\Leftrightarrow count(|A|+|B|) - count(|A\cap B|) \approx k|A\cap B|$
$\Leftrightarrow |A\cap B| \approx \frac{count}{count+k}(|A|+|B|) = \hat{I}$.

Estimating local clustering coefficients

Input: graph G;$k \in \mathbb{N}^+$
Output: estimates cc_v of cc(v) for all $v \in V$
forall $(u,v) \in E$ $Z_{u,v} \leftarrow 0$;
forall $i \leftarrow 1$ to k:
    $\pi_i \leftarrow$ random permutation of V;
    forall $v \in V$:
        $min_v \leftarrow min(\pi_i(N(V)))$;
    forall $(u,v) \in E$
        if $min_u == min_v$: $Z_{u,v} \leftarrow Z_{u,v} + 1$;
forall $v \in V$:
    $cc_v \leftarrow (\frac{1}{2}\sum_{u\in N(V)} \frac{Z_{u,v}}{Z_{u,v}+k}(|N(u)|+|N(v)|))$;
    $cc_v \leftarrow cc_v \frac{1}{deg(v)(deg(v)-1)}$;
return values cc_v;

Proposition

The values cc_v reported by EstimateLCCS(G,k) are estimates of the values cc(v).
Sketch of the proof: $\forall u,v \in E$, by the previous algorithm $\frac{Z_{u,v}}{Z_{u,v}+k}(|N(u)|+|N(v)|)$ is an estimate of $|N(u)\cap N(v)|$. From the previous lecture: the number of triangles containing a vertex v is $\frac{1}{2}\sum_{u\in N(v)}|N(u)\cap N(v)|$.Then
$\Rightarrow \frac{1}{2}\sum_{u\in N(v)} \frac{Z_{u,v}}{Z_{u,v}+k}(|N(u)|+|N(v)|)\frac{1}{deg(v)(deg(v)-1)}$ is an estimate of cc(v).

Proposition

EstimateLCCs(G,k) can be implemented with 2(k+1) passes on E in the streaming model(all edges incident to a vertex are consecutive):

- The loop "$for\ i \leftarrow 1$ to k" can be implemented with 2 passes on E for each iteration:
  - 1 pass to compute $min_v = min(\pi_i(N(v)))$
  - 1 pass to increment $Z_{u,v}$ if needed, by checking $min_u = min_v$

- The first for loop initializes $Z_{u,v}$ to 0 $\forall (u,v) \in E$(1 pass)
- Last for loop (computation of cc_v given $Z_{u,v}$)(1 pass)

Hence the total number of passes is 2k+1+1 = 2(k+1).
Corollary: The complexity of EstimateLCCs(G,k) is O(km).

Clustering coefficient: Naive algorithm

The idea is that for each triplet u,v,z of vertices, with $u \neq v \neq z \neq u$, check if it is a triangle.
Input: Graph G
Output: clustering coefficient of G
$num_t \leftarrow 0$;
forall $u \in V$
    forall $v \in V$
        forall $z \in V$
            if $u \neq v$ and $u \neq z$ and $v \neq z$:
                if $(u,v) \in E$ and $(u,z) \in E$ and $(v,z) \in E$
                $num_t \leftarrow um_t + 1$;
return $\frac{num_t}{6\binom{n}{3}}$
The complexity is $\Theta(n^3)$;

Better Algorithm

The idea is that instead of considering all triplets of nodes, start from edges, and consider all edges incident to the same vertex v one after the other. Note that this is equivalent to the exact algorithm to compute cc(v) for all $v \in V$.
Input: Graph G
Output: clustering coefficient of G
$num_t \leftarrow 0$;
forall $u \in V$
    forall $v \in N(u)$
        forall $z \in N(u)$
            if $v \neq z$ and $(v,z) \in E$:
                $num_t \leftarrow num_t + 1$
return $\frac{num_t}{6\binom{n}{3}}$
The complexity is $O(m\delta(G))$, which is too high for large networks.

Approximation

We are going to look at a specific algorithm with the following assumption: all edges incident to a vertex v are stored subsequently. The idea is: sample a path u-v-z of length 2 uniformly at random and check if the edge (z,u) closes a triangle. To sample a path u-v-z uniformly at random means choosing with probability 1/P one of the P possible paths. More specifically:

- for a vertex v with degree deg(v), the number of distinct paths of the type u-v-z is $\frac{deg(v)}{2}(deg(v)-1)$
- use the fact above to count the total number P of paths of length 2. $P = \sum_{v \in V} \frac{deg(v)}{2}(deg(v)-1)$
- Pick a value uniformly at random in $\{1, .., P\}$
- scan the edges to find the vertex v in the middle of the path, and pick the correct path.

Algorithm for approximate clustering coefficient

Input: graph G and $k \in \mathbb{N}^+$
Output: approximation of clustering coefficient of G
$P \leftarrow 0$
forall $u \in V$
    $d_u \leftarrow |N(u)|$;
    $P \leftarrow P + \frac{d_u}{2}(d_u - 1)$;
forall $i \leftarrow 1$ to k
    $(u, v, z) \leftarrow$ path of length 2 chosen uniformly at random;
    $if (u, z) \in E$
        $\beta_i \leftarrow 1$
    $else$
        $\beta_i \leftarrow 0$;
$num_t \leftarrow \frac{1}{k}(\sum_{i=1}^{k} \beta_i) \underbrace{(\frac{\sum_{v \in V} d_v(d_v - 1)}{6})}_{P/3}$;
return $\frac{num_t}{\binom{n}{3}}$;

Let T_i be the set of subsets of 3 nodes having exactly i edges among them

$T_0 = \{u, v, z : v, u, z \text{are not connected}\}, T_1 = \{u, v, z : v, u, z \text{only a combination of two nodes are connected}\},$

Hence, for each i=1,...,k we have that

$$\mathbb{E}[\beta_i] = \frac{3|T_3|}{|T_2| + 3|T_3|}$$

Proof: for each i=1,...,k: β_1 is a 0-1 random variable $\Rightarrow \mathbb{E}[\beta_i] = Pr[\beta_i = 1]$.
$Pr[\beta_i = 1] = Pr[\text{path u-v-z is part of a traingle}] =$
$= \frac{\text{number of paths of length 2 that are part of a traingle}}{number of paths of length 2} =$
$= \frac{3|T_3|}{|T_2|+3|T_3|}$.

Proposition

$\mathbb{E}[num_t] = |T_3|$.
Proof: $num_t = \frac{1}{k}(\sum_{i=1}^{k} \beta_i)(\sum_{v \in V} \frac{deg(v)}{6}(deg(v) - 1))$
$= \frac{1}{k}(\sum_{i=1}^{k} \beta_i)\frac{1}{3}) \underbrace{(\sum_{v \in V} \frac{deg(v)}{2}(deg(v) - 1)}_{P=\text{number of paths of length 2}=|T_2|+3|T_3|}$.

Therefore $\mathbb{E}[num_t] = \frac{1}{k}\frac{1}{3}(|T_2| + 3|T_3|)\mathbb{E}[\sum_{i=1}^{k} \beta_i]$
$= \frac{1}{k}\frac{1}{3}(|T_2| + 3|T_3|) \sum_{i=1}^{k} \mathbb{E}[\beta_i]$
$= \frac{1}{k}\frac{1}{3}(|T_2| + 3|T_3|)k\frac{3|T_3|}{|T_2|+3|T_3|} = |T_3|$

Proposition

Let $\epsilon > 0, \delta \in (0, 1)$ be constants. If

$$k \geq \frac{1}{3}\frac{|T_2| + 3|T_3|}{|T_3|} ln\frac{2}{\delta}$$

then

$$\mathbb{P}[(1-\epsilon)|T_3|] \leq num_t \leq \mathbb{P}[(1+\epsilon)|T_3|] \geq 1 - \delta$$

Proof: $X = \sum_{i=1}^{k} \beta_i$, X is the sum of 0-1 random variables. $\Rightarrow \mathbb{E}[X] = \mathbb{E}[\sum_{i=1}^{k} \beta_i] = k\mathbb{E}[\beta_i]$. Then

$$\mathbb{P}[(1-\epsilon)|T_3|] \leq num_t \leq 1+\epsilon)|T_3|] = 1 - \mathbb{P}[num_t \notin [1-\epsilon)|T_3|, 1+\epsilon)|T_3|]]$$

$$\mathbb{P}[num_t \notin [1-\epsilon)|T_3|, 1+\epsilon)|T_3|]] \leq \mathbb{P}[num_t < (1-\epsilon)|T_3|] + \mathbb{P}[num_t > (1+\epsilon)|T_3|](*)$$

$$\mathbb{P}[num_t < (1-\epsilon)|T_3|] = \mathbb{P}[\frac{1}{k}(\sum_{i=1}^{k} \beta_i) \underbrace{(\frac{1}{6})\sum_{v \in V} d_v(d_v - 1)}_{\frac{1}{3}(|T_2|+3|T_3|)}]$$

$$= \mathbb{P}[\underbrace{\sum_{i=1}^{k} \beta_i}_{X} < (1 - \underbrace{\epsilon}_{\delta}) \underbrace{k \underbrace{\frac{3|T_3|}{|T_2| + 3|T_3|}}_{\mathbb{E}[\beta_i]}}_{\mathbb{E}[X]}]$$

By chernoff bound

$$\leq e^{-\frac{\epsilon^2}{3} k \frac{3|T_3|}{|T_2|+3|T_3|}}$$

$$\leq e^{-\frac{\epsilon^2}{3} \frac{3|T_3|}{|T_2|+3|T_3|} - \frac{1}{\epsilon^2}\frac{|T_2|+3|T_3|}{|T_3|} ln\frac{2}{\delta}}$$

$$= e^{-ln\frac{2}{\delta}} = \frac{\delta}{2}$$

Analogously:

$$\mathbb{P}[num_t > (1+\epsilon)|T_3|] \leq \frac{\delta}{2}$$

Therefore $(*) \leq \frac{\delta}{2} + \frac{\delta}{2} = \delta$
$\Rightarrow \mathbb{P}[(1-\epsilon)|T_3| \leq num_t \leq (1+\epsilon)|T_3|] \geq 1 - \delta$

Proposition

ApproximateCC(G,k) can be implemented with 3 passes on the data:

- 1 pass to compute P
- the forall i=1 to k loop can be implemented with 2 passes:

- once i know P: choose k values in $\{1, ..., P\}$ uniformly at random $\Rightarrow$ i know the indeces of the paths of length 2 that we need to check
- 1 pass: select the k paths of length 2
- 1 pass: check forall paths of length 2 if it is a triangle

Significance and random Graphs

We need to describe how to compare the observation in the real network with the observation from a random network and how do we compare the measure on a random network.
We are interested in verifying whether simple characteristics of the graph explain our measure feature F(e.g., the clustering coefficient). We use a framework corresponding to statistical hypothesis testing. We assume null hypothesis H_0 that f well conforms with the distribution observed in a random graph. A **random graph** is a graph taken uniformly at random among all graph with the required simple characteristics.
The question is : is cc(G) well conforms with the distribution of cc(G) in a random graph?
null hypothesis H_0: cc(G) well conforms with the distribution of cc(G) in a random graph.
Random graph: graph taken uniformly at random among all grpahs with $|V|$ vertices and $|E|$ edges (or where the expected number of edges is $|E|$).
How do we assess whether the null hypothesis H_0 is true or not? Note that given that we are considering random graphs, our observation is almost always obtainable in at least one or few random graphs. We need a way to quantitatively assess how likely it is that the value f of feature F arises when H_0 is true. The commonly used measures are the z-score or the p-value.

z-score

Let f be the measure of the feature $\mathcal{F}$ of interest. Let $X_{\mathcal{F}}$ be the random variable corresponding to the value of the feature $\mathcal{F}$ when the null hypothesis H_0 is true, i.e. in a random graph. We define the z-score of H_0 as

$$\frac{f - \mathbb{E}[X_{\mathcal{F}}]}{\sigma[X_{\mathcal{F}}]}$$

Note that the expectation $\mathbb{E}[X_{\mathcal{F}}]$ and the standard deviation $\sigma[X_{\mathcal{F}}]$ are with respect to the distribution of random graphs given by the null hypothesis H_0 and that the z-score can be positive or negative.

p-value

Measures the probability of observing a value for feature $\mathcal{F}$ at least as extreme as f when the null hypothesis H_0 is true. Meaning of "at least as extreme" depends on the feature and the specific situation. For example, given the clustering coefficient cc(G), "at least as extreme" could be either $\geq cc(g)$ either $\leq cc(G)$. We define the p-value $p(H_0)$ of H_0 as

$$p(H_0) = \mathbb{P}[X_{\mathcal{F}} \text{ is at least as extreme as f} | H_0 \text{ is ture}]$$

Note that:

- if we are interested in understanding if the value of f is higher than expected(when H_0 is true): $p(H_0) = \mathbb{P}[X\mathcal{F} \geq f | H_0 \text{ is true}]$
- if we are interested in understanding if the value of f is lower than expected(when H_0 is true): $p(H_0) = \mathbb{P}[X\mathcal{F} \leq f | H_0 \text{ is true}]$
- if $p(H_0)$ is small it is unlike that the observed value f is obtained when H_0 is true.

Small digression

Usually the p-value is used to reject null hypothesis H_0 using the following rule:
Rejection rule: given a value $\alpha \in (0, 1)$, reject H_0 if $p(H_0 \leq \alpha)$. Common values of α are 0.05 or 0.1. Rejecting H_0 means flagging feature f as significant. The following proposition provides the ground for the rejection rule above:
Proposition: If the rejection rule above is used, then $\mathbb{P}[H_0 \text{ rejected} | H_0 \text{ ture}] \leq \alpha$.
Note that in statistical hypothesis testing two errors are possible:

- H_0 is true but H_0 is rejected (false positive or type I errors)
- H_0 is not true but H_0 is not rejected (false negative or type II error)

The previous proposition shows that the rejection rule provides guarantees on false positives.

Erdos-Renyi Random Graphs

The idea is: the simple characteristics that is preserved is the number of edges in the graph.

Definition

A random graph from the G(h,m) model is a graph chosen uniformly at random among all graphs with n vertices and m edges.

Note that the vertices are implicitly considered labeled.

Erdos-Renyi-Gilbert Random Graphs

Similar to the previous mmodel, but the number of edges is preserved only in expectation.

Definition

A random graph from the G(n,p) model is a graph with n vertices and were each edge appears with probability p independently of all other events.

Proposition

Let m(n,p) be the number of edges of a random graph from G(n,p). Then

$$\mathbb{E}[m(n,p)] = \binom{n}{2} p$$

Proof: Consider all possible edges in a graph from G(n,p)(according to an arbitrary order):

$$m(n,p) = \sum_{i=1}^{\binom{n}{2}} X_i, X_i \begin{cases} 1 \text{ if i-th edge appears} \\ 0 \text{ otherwise} \end{cases}$$

$$\mathbb{E}[m(n,p)] = \mathbb{E}[\sum_{i=1}^{\binom{n}{2}} X_i] = \sum_{i=1}^{\binom{n}{2}} \mathbb{E}[X_i] = \sum_{i=1}^{\binom{n}{2}} \mathbb{P}(X_i = 1)$$

$$\sum_{i=1}^{\binom{n}{2}} p = p\binom{n}{2}$$

Informally: if $p = \frac{m}{\binom{n}{2}}$, then G(n,m) and G(n,p) are relatively similar models. Note that the two models are not identical, but several properties are the same, and that G(n,p) is usually preferred because it allows easier analytical results.
However, not all properties can be easily computed analytically. Depending on the analysis/property/measure we need the expectation and standard deviation for the z-score or the distribution for the p-value.

Monte-Carlo Approach

Let f be the measure of the feature $\mathcal{F}$ of interest on your real network. If you can generate a random graph, then you use the following Monte-Carlo approach:

- Generate P instances $G_1, ..., G_P$ of a random graph
- compute the measure of interest on the P instances: $\mathcal{F}(G_1), ..., \mathcal{F}(G_P)$
- use $\mathcal{F}(G_1), ..., \mathcal{F}(G_P)$ to estimate the z-value or p-value for f

Note that this approach can be very expensive computationally an it is embarrassingly parallel.

Back to G(n,p)

While useful, G(n,p) does not resemble the real world networks.

Random graphs with given degree sequence

Problem: given a graph G, you wan to generate a random graph where each vertex has the same degree as in G. More formally:

- $V = \{1, ..., n\}$
- d_i=deg(i) the degree of vertex i in G.

Given d_i for i=1,...,n the goal is to generate a random graph $G' = (V, E')$ where the degree of each vertex i is d_i. We want to start form G, swap pairs of edges in G while preserving the degree of

each vertex. Then repeat the previous steps many times.

RandomGraphFixedDegrees(G,k)
Input: Graph G=(V,E);$k \in \mathbb{N}^+$;
Output: Random graph G' where each vertex has the same degree as in G
$E' \leftarrow E; G' = (V, E')$;
for i$\leftarrow$ 1 to k
sample edges (u,v) and (w,z) form E' uniformly at random;
$if (u, z) \notin E'$ and $(w, v) \notin E'$
$E' \leftarrow E' \setminus \{(u, v), (w, z)\} \cup \{(u, z), (w, v)\}$
return G';

Analysis

We want that the output of RandomGraphFixedDegrees(G,k) is a graph chosen uniformly at random among all the graphs with the same degree sequence as G. Equivalently, every graph with the same degree sequence as G is produced in output by RandomGraphFixedDegrees(G,k) with the same probability.
RandomGraphFixedDegrees(G,K) can produce in output every graph with the same degree sequence as G. Here is why:

RandomGraphFixedDegrees(G,k)
Proposition
Every graph with the same degree sequence as G can be obtained by a sequence of edge swap operation.

Every graph with the same degree sequence as G have the same probability to be produced in output by RandomGraphFixedDegrees if k is large enough.
Intuition: From the theory of Markov chains: if k is large enough, in a random walk on a connected graph R(N,T) where "for each edge $u \to v$ there is an edge $v \to u$" starting from any node $G \in N$ we have

$$\mathbb{P}[\text{ being in node G' after k steps}] = \frac{d_{G'}}{\sum_{G'' \in N} d_{g''}} = \pi(G')$$

where $d_{G'}$ is the outdegree of node G' in R. Since $d_{G'}$ is the same for every G' in R(thanks to the self loops), then $\pi(G')$ is the same for every graph with the same degree sequence as G. $\Rightarrow$ uniform distribution.
K should be large, for example, like $100|E|$. More precisely $k \in \Omega(|E|)$.

A different, but not correct algorithm

RandomGraphFixedDegrees(G,k)
Input: Graph G=(V,E);$k \in \mathbb{N}^+$;
Output: Random graph G' where each vertex has the same degree as in G
$E' \leftarrow E; G' = (V, E')$;
for i$\leftarrow$ 1 to k
sample edges (u,v) and (w,z) form E', with $(u, z) \notin E'$ and $(w, v) \notin E'$, uniformly at random;
$E' \leftarrow E' \setminus \{(u, v), (w, z)\} \cup \{(u, z), (w, v)\}$
return G';

This algorithm is wrong since it does not generate a random graph because the distribution is not uniform in general due to the absence of self-loops in R.

The Chung-Lu Model

The idea is that instead of having the exact degree sequence, each node has approximately the same degree as i G in expectation. Here is how:
Let deg(u) be the degree of $u \in V$ in G, and $|E| = m$. For each pair (u,v), let $p_{u,v} = \frac{deg(u)deg(v)}{2m}$.
In the random graph G', every edge(u,v) is gonna appear with probability $p_{u,v}$ independently of all other events.

Proposition

Let d_u the degree of u in random graph from the Chung-Lu model. Then

$$\mathbb{E}[d_u] = deg(u)(1 - \frac{deg(u)}{2m})$$

Proof: $d_u = \sum_{v \neq u, v \in V} X_{u,v}$, where $X_{u,v} = \begin{cases} 1 \text{ it edge (u,v) appears} \\ 0 \text{ otherwise} \end{cases}$ then:

$$\mathbb{E}[d_u] = \mathbb{E}[\sum_{v \neq u, v \in V} X_{u,v}] = \sum_{v \neq u, v \in V} \mathbb{E}[X_{u,v}]$$

$$= \sum_{v \neq u, v \in V} \mathbb{P}(X_{u,v} = 1) = \sum_{v \neq u, v \in V} p_{u,v}$$

$$\sum_{v \neq u, v \in V} \frac{deg(u)deg(v)}{2m} = \frac{deg(u)}{2m} \sum_{v \neq u, v \in V} deg(v)$$

$$\frac{deg(u)}{2m}(\underbrace{\sum_{v \in V} deg(v)}_{2m} - deg(u))$$

$$= deg(u)(1 - \frac{deg(u)}{2m})$$

Note that $\mathbb{E}[d_u] \approx deg(u)$. Some nodes u may have a degree fairly different from deg(u). The model is fairly amenable to analytical results. It is the underlying model for a commonly used heuristic algorithm to fin communities in networks.

Graphlets and Motifs

Graphlets

Other features that ca be computed for a graph or for a node is the counts of graphlets. A graphlet is a small connected subgraph. We are interested in counting the number of times a graphlet appears exactly as a subgraph.

Definition

Given a graph G=(V,E), let $S \subset V$. The induced subgraph $G[S] = (S, E')$ is the graph with

- vertex set S;
- edge set E' with $E' = \{(u, v) \in E : u \in S, v \in S\}$

Definition

Two graphs $G = (V_g, E_g)$ and $H = (V_H, E_H)$ are isomorphic, denoted as $G \simeq H$ if there exists a bijection $f : V_G \to V_H$ such that $(u, v) \in E_g$ if and only if $(f(u), f(v)) \in E_H$.

More informally: the count of a graphlet $H = (V_H, E_H)$ in a graph G is the number of subgraphs of G which H is isomorphic to. The count/score of a graphlet can be a feature at the graph level or at the node level.

Definition

Given a graphlet $H = (V_H, E_H)$ and a graph G=(V,E), the count c(H,G) of H in G is c(H,G)=$|\{S \subset V : H \simeq G[S]\}|$

Let's define the corresponding computational problem:

Definition

Given a graphlet $H = (V_H, E_H)$ and a graph G=(V,E), the graphlet counting problem asks to compute c(H,G).

Let's define the Subgraph Isomorphism problem:

Definition

Given a graph H and a graph G, the subgraph isomorphism problem asks to determine whether G contains a subgraph isomorphic to H.

This problem is NP-complete. A polynomial time algorithm for the graphlet counting problem implies a polynomial time algorithm for the subgraph isomorphism problem.

Algorithms for graphlet counting

Naive algorithm:
Input: graph G=(V,E),$k \in \mathbb{N}^+$ Output: c(H,G) for all graphlets H of size k
1) for all graphlest H of size k do: $c(H,G) \leftarrow 0$; 2) for all $S \subset V, \lvert S\rvert = k$: for all graphlets H of size k: if $H \simeq G[S]: \ c(H,G) \leftarrow c(H,G)+1$; 3) return $c(H,G) \forall \ H$;

The complexity is $\Omega(n^k)$, where $n = |V|$.

ESU algorithm

It takes as input G and an positive integer k, and gives as output the couts in G of all graphlets with k vertices. It works in two phases:

- first phase: enumerate all connected subgraphs with k nodes in G
- second phase: compute the count C(H;,G) of each graphlet H with vertices.

First phase

Maintains two sets: $V_{Subgraph}$, which is the currently constructed subgraph, and $V_{Extension}$, which is the set of candidates nodes to extend the subgraph. The idea is to start with a node v, add those nodes u to the set $V_{Extension}$ that have two properties:

- the ID of node u must be larger than the ID of node v
- u may only be a neighbor of some newly added node w nut not of any node already in $V_{Subgraph}$.

Esu is implemented as a recursive algorithm: its execution can be represented by its recursion tree, called ESU-Tree, which has depth k, the root corresponds to the first call of the algorithm and each internal node is labeled with the pair $(V_{Subgraph}, V_{Extension})$.

EnumerateSubgraphs(G,k)(ESU):
Input: graph G=(V,E),$1 \leq k \leq \lvert V\rvert$ Output: All size-k subgraphs in G
for each vertex $v \in V$ $V_{Extension} \leftarrow \{u \in N(\{v\}) : u > v\}$ call ExtendSubgraph$(\{v\}, V_{Extension}, v)$ return
ExtendSubgraph$(V_{Subgraph}, V_{Extension}, v)$

if$\lvert V_{Subgraph}\rvert == k$: output $G[V_{Subgraph}]$ and return;
while $\lvert V_{Extension}\rvert \neq \emptyset$:
Remove an arbitrarly chosen vertex w from $V_{Extension}$
$V'_{Extension} \leftarrow V_{Extension} \cup \{u \in N_{excl}(w, V_{subgraph}) : u > v\}$
call ExtendSubgraph$(V_{Subgraph} \cup \{w\}, V'_{Extension}, v)$
return;

Where $N_{excl}(w, V_{Subgraph})$: all nodes that are neighbors of w but not in $V_{Subgraph}$ or among the neighbors $\mathcal{N}(V_{Subgraph})$ of nodes in $V_{Subgraphs}$. Hence:

$$N_{excl}(w, V_{Subgraph}) = \underbrace{\mathcal{N}(w)}_{\text{neighbors of w}} \setminus (V_{Subgraph} \cup \underbrace{\mathcal{N}(V_{Subgraph})}_{\text{neighbors of nodes in } V_{Subgraph}})$$

Second phase

Once all connected subgraphs with k nodes of V are enumerated: obtain the count of all graphlets with k nodes. Requires to solve the graph isomorphism problem. When k is fairly small it can be done efficiently.

Motifs

A motif for a graph is a graphlet that is significantly overrepresented in G. How do we measure if a graphlet is significantly overrepresented? we can do comparisons with appropriate randome graphs: large z-score or small p-value.
Sometimes the normalized z-scores of all graphlets of size up to k are used as graph-level feature.

Definition
Given a graph G and $k \in \mathbb{N}^+$, the network significance profile of G is the vector of the normalized z-scores of the graphlets of size $\leq k$ in G. If $\mathcal{G}(k)$ is the set of graphlets of size $\leq k$ and $Z_G(H)$ is the z-score for graphlet H in G, then the normalized z-score of H in G is: $$\frac{Z_G(H)}{\sum_{H' \in \mathcal{G}}(K) Z_G(H')}$$

Graphlets and Motifs: Node level

Graphlets and motifs can be used to define node level features as well. Given a graphlet H, its orbits are teh different positions in which a node can appear (up to automorphism).

Definition
Given a graph G=(V,E),$k \in \mathbb{N}^+$, and a node $v \in V$, the graphlet degree vector GDV(v,G) of v in G is the vector where each component is the number of occurrences of v in a given orbit of a given graphlet of size $\leq k$.

You could also define a similar vector based on motifs instead. But how can we obtain the GDV(v,G) for a node v? We can use the following simple algorithm:

- based on the graphlet size k, compute the maximum distance r between v and a node in a graphlet H of size $\leq k$
- collect the subgrap G' of G containing v and all its neighbors up to distance r
- use the first phase of the ESU algorithm to enumerate all appearances of graphlets in G'
- compute GDV(v,G)

Node Embedding

So far we have seen how to calculate some values, that is because we want to focus on how theu can be used as features in a vector representation of nodes/graphs to be used as input for a machine learning task. The traditional framework for machine learning with graphs is the following

$$\text{Input graph} \rightarrow \text{Structured features} \rightarrow \text{Learning algorithm} \rightarrow \text{prediction}$$

Given a graph we want to extract node-level or graph-level features and learn a model that maps feature vectors to labels.
Representation learning: automatically learn the features to be used the downstream prediction task. We'll see:
Node Embedding task: map nodes into an embedding space
Graph embedding task: map graphs into an embedding space.
The goal is to find an efficient and task-independent feature learning for machine learning on graphs.
In this chapter we'll see node embedding for the following reasons:

- the similarity of embeddings indicates the similarity in the network
- encode the network information
- can be used for many downstream task predictions

General framework

Given a graph G=(V,E), we want to embed its nodes so that the similarity in the embedding space approximates the similarity in the graph.

Encoder-Decoder framework

The overall idea is the following:

- encoder: maps nodes to their embeddings
- similarity function: defines the similarity of pairs of nodes in the original network
- decoder: maps embeddings to the estimated similarity score
- optimize the parameters of the encoder so that the similarity of two nodes in the network is approximated by their embeddings.

Encoder

The encoder is a function $ENC : V \to \mathbb{R}^d$. Given $v \in V$, ENC generates the embedding $ENC(v) = z_v$ of node v. The encoder ENC depends on several parameters that are trained/optimized during the learning phase.

Similarity function

We consider pairwise similarity functions: $S : VxV \to \mathbb{R}$. Given u and v in V, S(u,v) measures the similarity between y ad v in G. Usually S(u,v) is represented by a matrix S with $S_{u,v} = S(u, v)$.

Decoder

In general it is a function that, given a set of node embeddings, computes a (user specified) graph statistic. We focus on commonly used pairwise decoders: $DEC : \mathbb{R}^d x \mathbb{R}^d \to \mathbb{R}$. Given embeddings z_u and z_v of u and v in V, it reconstructs the similarity of nodes u and v in G. Usually DEC has no trainable parameters.

Loss Function

Given nodes u,v$\in V$, the loss function is:

$$\ell(DEC(z_u, z_v), S(u, v))$$

which measures the discrepancy between the decoded similarity value $DEC(z_u, z_v)$ and the true similarity S(u,v). The overall goal is to optimize the parameters of ENC so that each pair $u, v \in V$:

$$DEC(z_u, z_v) = DEC(ENC(u), ENC(v)) \approx S(u, v)$$

$\approx$: captured by $\ell(DEC(ENC(u), ENC(v)), S(u, v))$.
In practise we set ENC parameters to minimize the **Empirical loss**

$$\mathcal{L} = \sum_{u,v \in VxV} \ell(DEC(z_u, z_v), S(u, v))$$

Sometimes a set $\mathcal{D}$ of training pairs of nodes is used:

$$\mathcal{L} = \sum_{u,v \in \mathcal{D}} \ell(DEC(z_u, z_v), S(u, v))$$

Shallow encoding

We now consider the simplest type of encoder: ENC is a look-up function:

$$ENC(V) = z_v = Zxv$$

where:

- $Z \in \mathbb{R}^{dx|V|}$: matrix where each column is a node embedding
- $v \in \{0, 1\}^{|V|}$: indicator vector, all 0's except a single 1 indicating node v.

We directly optimize the embedding of each node. The parameters are all entries of Z.
With shallow embeddings we are learning a representation(vector in $\mathbb{R}^d$) for each node $v \in V$, but we are not learning a function to obtain the representation of a node $v \in V$. It is unsupervised/self-supervised way of learning node embeddings: we are note using node labels and features. These embeddings are task independent: they are not trained for a specific task but can be used for may tasks.

Adjacency-based similarity embeddings

Let A be the adjacency matrix of G. We define the similarity as S(u,v)=$A_{u,v}$, the shallow encoder as ENC(v)=Zxv=z_v, and the decoder as $DEC(z_u, z_v) = z_u^T z_v$. We use as loss function the squared loss:

$$\ell(DEC(z_u, z_v), S(u, v)) = (z_u^T z_v - A_{u,v})^2$$

The empirical loss then becomes:

$$\mathcal{L} = \sum_{u,v \in VxV} \ell(DEC(z_u, z_v), S(u, v)) = \sum_{u,v \in VxV} (z_u^T z_v - A_{u,v})^2$$

Therefore the embeddings z_v for each $v \in V$ are given by the solution Z^* to

$$Z^* = \text{argmin}_z \sum_{u,v \in VxV} (z_u^T z_v - A_{u,v})^2$$

We can compute Z^* in two ways: matrix decomposition, but it doesn't scale, or general optimization method such as Gradient descent.

Gradient descent for Adjacency-based similarity embeddings

We want to find the vextor $z_u, \forall u \in V$ that minimizes:

$$\mathcal{L} = \sum_{u,v \in VxV} (z_u^T z_v - A_{u,v})^2 = \sum_{u \in V} \sum_{v \in V} (z_u^T z_v - A_{u,v})^2$$

The gradient descent algorithm becomes:

GD
1) initialize z_u to a random vector $\forall u \in V$
2) iterate until convergence:
$for u \in V$:
i) compute $\frac{\partial \mathcal{L}}{\partial z_u}$
ii)$z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}}{\partial z_u}$
3) return $z_u \forall u \in V$;

The complexity is $\Theta(|V|^2)$. To improve it we can use the Stochastic gradient descent.

SGD
1) initialize z_u to a random vector in $\mathbb{R}^d \forall u \in V$
2) iterate until convergence:
i)pick $u \in V$ uniformly at random:$\mathcal{L}^{(u)} = \sum_{v \in V} (z_u^T z_v - A_{u,v})^2$
ii) for all vertices $v \in V$
compute $\frac{\partial \mathcal{L}^{(u)}}{\partial z_u}$
$z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}^{(u)}}{\partial z_u}$
3) return $z_u \forall u \in V$;

The algorithm has complexity $\Theta(|V|)$. The improvement is the following: consider batches of nodes, so in step 2 you pick b nodes, and compute the contribution to the loss of such nodes to the loss and the gradient with respect to such nodes.

Limitation

While useful, adjacency-based similarity only considers direct connections among nodes.

Jaccard-similarity

We want to define similarity between u and v as a measure of similarity between their neighborhoods $\mathcal{N}(u)$ and $\mathcal{N}(v)$.

Definition

Given two vertices u and v, the Jaccard similarity between their neighborhoods is:

$$J(\mathcal{N}(u),\mathcal{N}(v)) = \frac{|\mathcal{N}(u) \cap \mathcal{N}(v)|}{|\mathcal{N}(u) \cup \mathcal{N}(v)|}$$

Here $\mathcal{N}(u)$ also includes node u: $\mathcal{N}(u) = \{u\} \cup \{v : \{u, v\} \in E\}$. If $v \in \mathcal{N}(u)$, it is also contained in $|\mathcal{N}(u) \cap \mathcal{N}(v)|$

Embedding with Jaccard similarity

We use as similarity the jaccard similarity,i.e. S(u,v)=$J(\mathcal{N}(u),\mathcal{N}(v))$, we use the shallow encoder and the decoder as for the adjacency. The loss function is the squared loss defined as follows:

$$\ell(DEC(z_u, z_v), S(u, v)) = (z_u^T z_v - J(\mathcal{N}(u),\mathcal{N}(v)))^2$$

The empirical loss then becomes:

$$\mathcal{L} = \sum_{u,v \in VxV} \ell(DEC(z_u, z_v), S(u, v)) = \sum_{u,v \in VxV} (z_u^T z_v - J(\mathcal{N}(u),\mathcal{N}(v)))^2$$

Therefore the embeddings z_v for each $v \in V$ are given by the solution Z^* to

$$Z^* = \text{argmin}_z \sum_{u,v \in VxV} (z_u^T z_v - J(\mathcal{N}(u),\mathcal{N}(v)))^2$$

How can we compute Z^*? as before, with SGD. This approach can be generalized to consider an extended neighborhood of nodes, for example, all nodes within distance k from u.

Embedding with Random Walk

The idea is that the similarity between u and v is given by the probability of visiting v on a random walk of length k starting at node u. More precisely:

Definition

Given a graph G, we define $\mathbb{P}[v|u]$ as the probability of visiting node v on a random walk of length k starting at u.

The algorithm for random walks is the following:

RandomWalk(u,k)
Input: node $u \in V$ of graph G; $k \in \mathbb{N}^+$; Output: random walk of length k starting from u; $v^{(0)} \leftarrow u$; for i=1 to k: $v^{(i)} \leftarrow$random vertex chosen uniformly at random from $\mathcal{N}(v^{(i-1)})$; return;

In general $\mathbb{P}[v|u] \neq \mathbb{P}[u|v]$. The components of the random-walk approach are: similarity S(u,v)=$\mathbb{P}[v|u]$; the encoder is the shallow encoder as before; the decoder $DEC(z_u, z_v) = \frac{(e_u^z)^T z_v}{\sum_{i \in V}(e_u^z)^T z_i}$(the softmax function). Note that the values of the decoder, for all $v \in V$ define a probability distribution. The loss function is then:

$$\ell(DEC(z_u, z_v), S(u, v)) = -\mathbb{P}[v|u] log \frac{(e_u^z)^T z_v}{\sum_{i \in V}(e_u^z)^T z_i}$$

Hence the empirical loss is:

$$\mathcal{L} = -\sum_{u,v \in VxV} \mathbb{P}[v|u] log \frac{(e_u^z)^T z_v}{\sum_{i \in V}(e_u^z)^T z_i}$$

Therefore the embeddings z_v for each $v \in V$ are given by the solution Z^* to

$$Z^* = \text{argmin}_z - \sum_{u,v \in VxV} \mathbb{P}[v|u] log \frac{(e_u^z)^T z_v}{\sum_{i \in V}(e_u^z)^T z_i}$$

The loss function measures the distance between the true values $\mathbb{P}[v|u]$ for all $v \in V$ and the corresponding predicted values $DEC(z_u, z_v) = \frac{(e_u^z)^T z_v}{\sum_{i \in V}(e_u^z)^T z_i}$ for all $v \in V$. It measures the difference between two sets of probabilities.

Cross entropy

Cross entropy H(p,q) between two probability distributions p,q is a measure of the difference between p and q:

Definition
given two discrete probability distribution p and q, with the same support $\mathcal{X}$, the cross entropy between p and q is: $$-\sum_{x \in \mathcal{X}} p(x) log q(x)$$

Therefore, given u, the difference between the true values $\mathbb{P}[v|u]$ and thei predicted value(softmax) is the loss we wrote. Summing such difference for all nodes u we obtain the empirical loss. The complexity to evaluate the empirical loss is $\Theta(|V|^2)$.

Optimization

Instead of considering all terms $\mathbb{P}[v|u]$, we consider only the ones with a high $\mathbb{P}[v|u]$. To do so, we consider that $\mathbb{P}[v|u]$ as the probability of visiting node v on a random walk of length k at u, hence, we run random walks of length k from u: if $\mathbb{P}[v|u]$ is high, v will appear in the random walks.

Definition
Let $\mathcal{N}_{RW,k}(u)$ be the set of nodes visited on a random walk of length l starting from node u.
Definition
Consider s random walks of length k starting from u. Let $n_{RW,k,s}(v\|u)$ be the number of such walks containing v. This term is a random variable.
Proposition
$\mathbb{E}[\frac{n_{RW,k,s}(v\|u)}{s}] = \mathbb{P}[v\|u]$

Proof: For i=1,...ms let $X_{v,i} = \begin{cases} 1 \text{ if v appears in the i-th RW of lenght k from u} \\ 0 \text{ otherwise} \end{cases}$ Then: $n_{RW,k,s}(v|u) = \sum_{i=1}^{s} X_{v,i}$ and

$$\mathbb{E}[\frac{n_{RW,k,s}(v|u}{s}] = \frac{1}{s}\mathbb{E}[n_{RW,k,s}(v|u] = \frac{1}{s}\mathbb{E}[\sum_{i=1}^{s} X_{v,i}]$$

$$= \frac{1}{s}\sum_{i=1}^{s}\mathbb{E}[X_{v,i}] = \frac{1}{s}\sum_{i=1}^{s}\mathbb{P}[X_{v,i}=1] = \frac{1}{s}\sum_{i=1}^{s}\mathbb{P}[v|u]$$

$$= \mathbb{P}[v|u]$$

Let $\mathcal{M}_{RW,k,s}(u)$ be the multiset of nodes visited during s random walks of length k starting from u, where vertex v appears in $\mathcal{M}_{RW,k,s}(u)$ exactly $n_{RW,k,s}(v|u)$ times.
We can use $\frac{n_{RW,k,s}(v|u)}{s}$ instead of $\mathbb{P}[v|u]$ in the empirical loss, hence we obtain:

$$\mathcal{L} = -\sum_{u\in V}\sum_{v\in V}\frac{n_{RW,k,s}(v|u)}{s} log(softmax)$$

Hence we have that

$$Z^* = \text{argmin}_z -\sum_{u\in V}\sum_{v\in V} n_{RW,k,s}(v|u) log(softmax)$$

we removed the 1/s term since it is a constant. Let us now consider $sum_{v\in V} n_{RW,k,s}(v|u) log(softmax)$. Note that:

- if v is not visited by the s random walks of length k from node u, then $n_{RW,k,s}(v|u) = 0 \Rightarrow$ ignore v in the sum above
- if $n_{RW,k,s}(v|u) = \ell_v$, then the term log(softmax) appears ℓ_v times in the sum, and ℓ_v is the number of times v appears in the multiset $\mathcal{M}_{RW,k,s}(u)$

Therefore:

$$\sum_{v\in V} n_{RW,k,s}(v|u) log(softmax) = \sum_{v\in \mathcal{M}_{RW,k,s}(u)} log(softmax)$$

Thus:

$$Z^* = \text{argmin}_z \sum_{u\in V}\sum_{v\in \mathcal{M}_{RW,k,s}(u)} log(softmax)$$

The overall approach is then:

- run s random walks of length k from each node $u \in V$ to obtain $\mathcal{M}_{RW,k,s}(u)$

- solve the following problem:

$$\text{argmin}_z - \sum_{u \in V} \sum_{v \in \mathcal{M}_{RW,k,s}(u)} log(\frac{(e_u^z)^T z_v}{\sum_{i \in V} (e_u^z)^T z_i})$$

The complexity is $\Theta(|V|^2)$. With the deep walks we obtined a complexity of $\Theta(|V|log|V|)$

Node2walk

The idea is to approximate the logarithm of the softmax function. Therefore: use a different function to obtain a probabiliy for a node v given a node u. Then sample random nodes to obtain the background probabilities(negative sampling).

Negative sampling

node2vec uses the following to compare the decoded probability for node v, given u, and the background probability:

$$log(\sigma(z_u^T z_v)) - \sum_{i=1}^{r} log(\sigma(z_u^T z_i))$$

where i is a node chosen from $|V|$ with probability proportional to its degree and $\sigma()$ is the sigmoid function. The r samples are called negative events. Large r allows us to have a robust estimate and stronger focus on negative events. in practice we have $r \in \{5, ..., 20\}$. Putting everything together we have that:

$$Z^* = \text{argmin}_z - \sum_{u \in V} \sum_{v \in \mathcal{M}_{RW,k,s}(u)} (log(\sigma(z_u^T z_v)) - \sum_{i=1}^{r} log(\sigma(z_u^T z_i)))$$

We can find Z^* with GD or SGD.

Graph embeddings

We want to embed an entire graph or subgraph G inyo an embedding $z_g \in \mathbb{R}^d$. The approach can be 2:

- first run a node technique on the graph G, then calculate the embedding of G as the sum of the embeddings of the nodes in G;
- first introduce a virtual node to represent the graph G and use its embedding as embedding of G. So, introduce a node g connected to all nodes in G, run a node embedding on the graph G including g, then the embedding of G is the embedding z_g of g.

Limitations of shallow embedding

It is an high complexity model, it cannot generate embeddings for nodes that are not present during the training phase(Inherently transductive). It do not incorporate node features: many networks have node features that can/should be used to produce embedding. They are unsupervised methods: learned embeddings are independent of the machine learning task.

Graph neural network embeddings

The main idea is that instead of learning z_v for each $v \in V$, we learn a function $f : V \to \mathbb{R}^d$. f is computed by a neural network that depends on the structure of the graph, hence a Graph neural network. Given G and features of the nodes $v \in V$, the GNN can compute the embedding of any node $u \in V$. The encoder-decoder framework then becomes: ENC() is a neural network, the other components are as for shallow embeddings. The parameters to be learned from the data are in the neural network.
Given a graph G, with $|V| = n$, and A is the adjecency matrix of G. Each node $v \in V$ has vector $x_v \in \mathbb{R}^r$ of r features, and $X \in \mathbb{R}^{rxn}$ is the matrix of nodes features. The goal is to combine the features x_v of a node and the structural information from G to obtain encodings for each node $v \in V$.

Naive approach

The idea is the following:

- for each node $v \in V$, build an input vector that contains both the node features x_v and the adjacency matrix vector for v
- ENC(v) is the output of a deep neural network whose input are the vectors above.

The number of parameters is $\Omega(n)$, and it is not applicable to graphs of different size. It depends on the node ordering.
We would like the embedding to be independent of the node ordering. Let f be the function that given the adjacency matrix A of G and the feature matrix X of the nodes G produces in output the embedding matrix Z:

$$Z = f(A, X)$$

We would like f to be permutation invariant or permutation equivariant.

Permutation invariance and equivariance

Let us define the permutation matrix P as: a matrix that per each column/row has exactly one 1, and all other entries are 0. It is a matrix $|V|x|V|$.

Definition
f is permutation invariant if $f(PAP^T, XP^T) = f(A, X)$, where P is a permutation matrix

Definition
f is permutation equivariant if $f(PAP^T, XP^T) = f(A, X)P^T$, where P is a permutation matrix

Neural Message Passing Framework

The idea is that the computation proceeds in iterations. In iteration k, the hidden embedding $h_v^{(k)}$ for node v is updated/computed according to the hidden embeddings of nodes $u \in \mathcal{N}(v)$. The output embedding for node v is the embedding $h_v^{(k)}$ after K iterations. The initialization is $h_v^{(0)} = x_v$ for all $v \in V$. More formally:

- $AGGREGATE^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\}) = m_{\mathcal{N}(u)}^{(k)}$: function that given the hidden embeddings of neighbors of u at iteration k produces message $m_{\mathcal{N}(u)}^{(k)}$
- $UPDATE^{(k)}(h_u^{(l)}), m_{\mathcal{N}(u)}^{(k)}$

AGGREGATE and UPDATE are arbitrarily differentiable function, hence neural networks. Then for each $u \in V$:

$$h_u^{(k+1)} = UPDATE^{(k)}(h_u^{(k)}, AGGREGATE^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\}))$$

$$= UPDATE^{(k)}(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)})$$

The embedding z_u of a node $u \in V$ is the embedding after K iterations: $z_u = h_u^{(K)}$. The iterations of message passing are also called layers of the GNN. If we consider proper AGGREGATE functions and proper UPDATE function, the resulting GNN is permutation invariant. The local feature-aggregation behavior of GNNs is analogous to the behavior of the convolutional filters in CNNs.

Basic GNN

The most basic version of a GNN is given by:

$$AGGREGATE^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\}) = m_{\mathcal{N}(u)}^{(k)} = \sum_{v \in \mathcal{N}(u)} h_v^{(k)}$$

Therefore

$$h_u^{(k+1)} = UPDATE^{(k)}(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)})$$

$$= \sigma(W_{self}^{(k+1)} h_u^{(k)} + W_{neigh}^{(k+1)} m_{\mathcal{N}(u)}^{(k)} + b^{(k+1)})$$

The two W matrices are trainable parameters, with $W_{self}^{(k+1)}, W_{neight^{(k+1)}} \in \mathbb{R}^{d^{(k+1)} x d^{(k)}}$, $b^{(k+1)}$ is the bias and $\sigma()$ is an elementwise non-linear function.

Parameter sharing

The neural networks in different iterations can share parameters. The two weight matrices and the bias can be shared across the GNN iterations. That is $W_{self}^{(k+1)} = W_{self}$,$W_{neigh}^{(k+1)} = W_{neigh}$ and $b^{(k+1)} = b$ for all $k \in \{0, ..., K-1\}$. In this case:

$$h_u^{(k+1)} = \sigma(W_{self} h_u^{(k)} + W_{neigh} \sum_{v \in \mathcal{N}(u)} h_v^{(k)} + b)$$

This implies that $d^{(k)} = d \; \forall k = 1, .., k$.

Graph-level equations

Many GNNs can be succinctly defined using graph-level equations. For the basic GNN, they are:

$$H^{(k)} = \sigma(AH^{(k-1)}W_{neigh} + H^{(k-1)}W_{self} + B)$$

where:

- A is the adjacency matrix of G
- $H^{(k)} \in \mathbb{R}^{|V|xd^{(k)}}$ is the matrix node representations at layer k of the GNN: each row is the representation of a node at layer k
- $B \in \mathbb{R}^{|V|xd^{(k)}}$ is the bias matrix, where each row it equal to the bias vector b.

Message Passing with Self-loops

Simplification of neural message passing: omit the update operator by adding self loops to the graph. With this modification, each iteration of a general GNN is:

$$h_u^{(k+1)} = AGGREGATE^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u) \cup \{u\}\})$$

This is a simplification of a general GNN, it reduces overfitting but it limits the expressiveness of the GNN. For the basic GNN: adding self-loops means sharing parameters between W_{self}, W_{neigh}, that is $W_{self} = W_{neigh} = W$.

Generalized Neighborhood Aggregation

The basic GNN can be improved and generalized in several ways. We now see how to generalize the AGGREGATE operator.
For each $u \in V$

$$h_u^{(k+1)} = UPDATE^{(k)}(h_u^{(k)}, AGGREGATE^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\}))$$

$$UPDATE^{(k)}(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)})$$

Neighborhood Normalization

The basic GNN are insensitive to node degrees. To cope with taht we normalize the AGGREGATE operator taking into account the degree:

$$m_{\mathcal{N}(u)^{(u)}} = \frac{\sum_{v \in \mathcal{N}(u)} h_v^{(k)}}{|\mathcal{N}(u)|}$$

In practice, the better solution is:

$$m_{\mathcal{N}(u)^{(k)}} = \sum_{v \in \mathcal{N}(u)} \frac{h_v^{(k)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}$$

Graph convolutional networks

They use the neighborhood normalization with self-loops. For each $u \in V$

$$h_u^{(k+1)} = \sigma(W^{(k+1)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{h_v^{(k)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}})$$

GraphSAGE

The main difference with GCNs is that it uses the generalized AGGREGATE operator, and that for each node u, a transformation $h_u^{(k)}$ is concatenated with the transformation of $m_{\mathcal{N}(u)}^{(k)}$.
For each node $u \in V$

$$h_u^{(k+1)} = \sigma([W^{(k+1)} m_{\mathcal{N}(u)}^{(k+1)}, B^{(k+1)} h_u^{(k)}])$$

where $m_{\mathcal{N}(u)}^{(k)} = AGGREGATE^{(k)}(\{h_v^{(k)} \forall v \in \mathcal{N}(u) \cup \{u\}\})$. Common choices for aggregate are:

- mean of $h_v^{(k)}$:

$$m_{\mathcal{N}(u)}^{(k)} = \sum_{v \in \mathcal{N}(u)} \frac{h_v^{(k)}}{|\mathcal{N}(u)|}$$

- first apply a simple NN to each $h_v^{(k)}$(with some learnable parameters θ) , and then apply an elementwise operator γ:

$$m_{\mathcal{N}(u)}^{(k)} = \gamma(\{NN_\theta(h_v^{(k)}), \forall u \in V\})$$

Graph Attention Network GAT

Neighborhood attention: assign an attention weight to each neighbor, which is used to weight this neighbor's influence during aggregation:

$$m_{\mathcal{N}(u)^{(k)}} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} h_u^{(k)}$$

where $\alpha_{u,v}$ is the attention node $v \in \mathcal{N}(u)$ when aggregating information at node u- They are learned from data. Various definitions of $\alpha_{u,v}$ have been used. The original one is:

$$\alpha_{u,v} = \frac{e^{a^T [W h_u^{(k)}, W h_v^{(k)}]}}{\sum_{v \in \mathcal{N}(u)} e^{a^T} [W h_u^{(k)}, W h_v^{(k)}]}$$

with $[W h_u^{(k)}, W h_v^{(k)}]$ as the concatenation of $W h_u^{(k)}$ and $W h_v^{(k)}$, W is a matrix of parameters and a^T is an attention vector.

Generalized Update operators

Informal version: when using a GCN-style network with K layers, the influence of node u on node v is proportional the probability $\mathbb{P}[v|u]$ of reaching node v on a random walk with K steps starting from node u.

GNNs: Graph level

What about using GNNs to learn the embedding z_G of a graph G? Approaches seen for graph embeddings usually work weel, at least for small graphs:

- $z_g = \sum_{u \in V} z_u$
- add a node g connected to all $u \in V$, then $z_G = z_g$

The second approach above can be made more flexible by using a specific neural network for g. We will see a theoretical motivation for GNNs, by comparing them with an algorithm for the graph isomorphism problem.

Definition
Given two graphs G_1 and G_2, the graph isomorphism problem requires to determine whether G_1 is isomorphic to G_2: $G_1 \simeq G_2$

It is not known to be solvable in polynomial time nor to be NP-complete.
Ideally we would like a graph embedding technique to solve the graph isomorphism problem, that is $z_{G_1} = z_{G_2}$ if and only if $G_1 \simeq G_2$.

Weisfieler-Leman Algorithm

It is for the graph isomorphism problem. It does not always produce the correct answer. If WL outputs that $G_1 \not\simeq G_2$ then $G_1 \not\simeq G_2$, on the other hand, if it outputs that $G_1 \simeq G_2$, then it may be that $G_1 \not\simeq G_2$. It is known that WL produces the correct answer for a large class of graphs.
This algorithm builds on a color refinement algorithm for the vertices of a graph G. Giving a coloring C_V of the vertices V of G, let $P(C_V)$ the partition of the vertices V defined by the coloring C_V: two vertices u,v are in the same set of the partitioning if and only if u and v have the same color.

ColorRefinement(G)
Input: graph G Output: coloring of V
$C_{curr} \leftarrow$ assign the same color u to all nods $u \in V$; repeat $C_{prev} \leftarrow C_{curr}$; $C_{curr} \leftarrow$for each pair u and v where u and v have the same color in C_{prev}: assign different colors to u and v if and only if there is some color c such that u and v have different number of neighbors of color c in C_{prev} until $P(C_{prev}) = P(C_{curr})$; return C_{curr};

Analysis:

- ColorRefinement(G) stops agter at most $|V|$ iterations;
- each iteration requires $= (|V|^2)$ operations;
- the complexity is $O(|V|^3)$;

$WL(G_1, G_2)$
Input: graphs G_1, G_2 Output: yes,no
$C_{G_1} \leftarrow$ColorRefinement(G_1); $C_{G_1} \leftarrow$ColorRefinement(G_2); return histogram$(P(C_{G_1}))$=histogram$(P(C_{G_2}))$;

The complexity of the algorithm is $O(|V_1|^3 + |V_2|^3)$.

Theorem

Consider a GNN with K message passing layers of the following form:

$$h_u^{(k+1)} = UPDATE^{(k)}(h_u^{(k)}, AGGREGATE^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\}))$$

where AGGREGATE is differentiable and permutation invariant and UPDATE is differentaible. Assume the the input is made of discrete features: $h_u^{(0)} = x_u \in \mathbb{Z}^d, \forall u \in V$. Then $h_u^{(K)} \neq h_v^{(K)}$ only if u and v have different labels after K iterations of the WL algorithm.

Informally, GNNs are no more powerful than the WL algorithm when we have discrete information as node features. The result generalizes to graphs: If the WL algorithm does not distinguish graphs G_1 and G_2, then any GNN is incapable of distinguishing G_1 and G_2.

Theorem

There exists a GNN with the form defined in the previous theorem such that $h_u^{(K)} = h_v^{(K)}$ only if u and v have the same labels after K iterations of the WL algorithm

Informally: there are GNNs that are as powerful as the WL algorithm, such as basic GNN. GraphSAGE could be as powerful but it depends on the AGGREGATION operator. Basic GG with neighborhood normalization and GCN are not as powerful as WL.

GNNs for node embeddings

In general GNNs can be used to obtain node embeddings. The encoder is the GNN, while all the other components of the framework are as in the shallow embeddings. However, GNNs can also be used for supervised tasks, such as node or graph classification.

Node classification

The input are some nodes with a label which can be use to train the GNN. Each node u in the training set has a label, encoded by a 0-1 vector y_u of dimension c. The Loss function is the negative log-likelihood loss of softmax classification function.
Given the embedding z_u(learned by the GNN) of node u and the corresponding vector y_u, the softmax classification function is

$$softmax(z_u, y_u) = \sum_{i=1}^{k} y_u[i] \frac{e^{z_u^T w_i}}{\sum_{j=1}^{c} e^{z_u^T w_j}}$$

with w_i for i=1,..,c trainable parameters. Then the loss function becomes:

$$\mathcal{L} = \sum_{u \in V_{train}} -log(softmax(z_u, y_u))$$

where V_{train} is the set of training nodes. Note: there are different types of nodes for node classification:

- the set V_{train} of training nodes: both the nodes and labels are used during training
- the set V_{train} of transductive test nodes: the nodes, but not their label, are used during training. These nodes do not contribute to $\mathcal{L}$
- the set V_{ind} of inductive test nodes: the nodes are completely unobserved during training.

Graph Classification

Input: set of graphs, where each graph G has a label represented by a 0-1 vector y_G of dimension c. The loss is the same as for the node classification, where the embedding z_G is computed for a graph G:

$$softmax(z_u, y_u) = \sum_{i=1}^{k} y_G[i] \frac{e^{z_G^T w_i}}{\sum_{j=1}^{c} e^{z_G^T w_j}}$$

$$\mathcal{L} = \sum_{G \in \mathcal{G}_{train}} -log(softmax(z_G, y_G))$$

where $\mathcal{G}_{train}$ is the set of subgraphs used for training.

Other tasks

GNNs can be used for several other tasks as: edge prediction(predict missing edges), regression task for nodes, regression task for graphs...

Graph Clustering

Given a graph G we want to partition V into clusters so that similar vertices are in the same cluster and different vertices are in different clusters. The intuition is that the similarity between the vertices are represented by the edges. Given a connected graph G, the goal is to partition V so that there are many edges within each cluster and few edges between clusters. We have many different formalizations based on this intuition.

Hierarchical clustering

The output is a dendogram, representing the clustering structure of the whole graph G. We have two general approaches for this type of clustering: agglomerative approach(start with each node in a cluster, iteratively join clusters) or divisive approach(start with all nodes in a cluster, iteratively split clusters).

AgglomerativeClustering(G)
Input: connected graph G Output: dendogram whose leaves are the elements of V
1 assign each node u to its own cluster C_u; 2 for all pairs $u, v \in V, u \neq v$ compute their similarity sim(u,v) 3 Repeat untill all nodes are in a single cluster: 4 find the pair of clusters C_1, C_2 with the highest similarity $sim(C_1, C_2)$ 5 merge clusters C_1, C_2 in a single cluster C' 6 compute similarity between C' and all other clusters 7 return the corresponding dendogram

The complexity of the algorithm is $\Theta(|V|^2)$. Different variants depending on the definition of sim(u,v) and the definition of $sim(C_1, C_2)$. The common choices for sim(u,v):

$$sim(u, v) = \frac{|\mathcal{N}(u) \cap \mathcal{N}(v)| + A_{uv}}{min\{deg(u), deg(v)\} + 1 - A_{uv}}$$

where A is the adjacency matrix of G. Common choices for $sim(C_1, C_2)$ define different types of linkage clustering:

- single linkage clustering: $sim(C_1, C_2) = min_{u \in C_1, v \in C_2} sim(u, v)$
- average link clustering:

$$sim(C_1, C_2) = \frac{1}{|C_1||C_2|} \sum_{u \in C_1, v \in C_2} sim(u, v)$$

- complete linkage clustering:$sim(C_1, C_2) = max_{u \in C_1, v \in C_2} sim(u, v)$

Link betweenness

Let $\sigma_{s,t}$ be the number of shortest paths from node s to node t. Let $\sigma_{s,t}(e)$ be the number of shortest paths from node s to node t that pass through edge e.

Definition
Given a connected graph G and an edge $e \in E$, the link betweennes b(e,G) of e in G: $$b(e,G) = \sum_{s,t \in V, s \neq t} \frac{\sigma_{s,t}(e)}{\sigma_{s,t}}$$

The complexity is $\Theta(|V||E|)$.

Girvan-Newman Algorithm

GNClustering(G)
Input: connected graph G Output: dendogram whose leaves are the elements of V
1 assign all nodes u to a single cluster C; 2 Repeat untill all nodes are in different clusters: 3 for each cluster C: 4for each edge $e \in C$ compute b(e,C) 5 let e_{max} the edge of maximum betweenness, and let C(e) its cluster; 6 remove e from C(e); 7 report the corresponding dendogram

The complexity is $\Theta(|E|^2|V|)$.
The output of hierarchical clustering is a dendogram, how do we get a clustering? we need to cut the dendogram at a given level. If we know the number k of cluster we want, we cut in the level which will give us k clusters. In case we do not know k: define a score for clustering, and pick the clustering from the dendogram of maximum score.

Cost based clustering

The common approach i clustering is to define a cost function over possible partitions of the objects and then find the partition of minimal cost.

Modularity

The idea is that a cluster should contain more edges than expected in a random graph.

Definition
Given a graph G with $\|V\| = n, \|E\| = m$ the modularity M(S) of a subset $S \subseteq V$ of the vertices of G is $$M(S) = \frac{1}{2m} \sum_{u,v \in S} (A_{uv} - \frac{deg(u)deg(v)}{2m})$$

The intuition is that it measures the difference between the number of edges within each cluster with the expected number of edges under the Chung-Lu model for random graphs.
The modularity of a clustering of G is the sum of the modularity of each cluster:

Definition

Given a clustering $\mathcal{C} = C_1, ...,$ of a graph G with $|V| = n, |E| = m$, the modularity $M(\mathcal{C})$ of $\mathcal{C}$ is:

$$M(\mathcal{C}) = \sum_{C \in \mathcal{C}} M(C) = \frac{1}{2m} \sum_{C \in \mathcal{C}} \sum_{u,v \in S} (A_{uv} - \frac{deg(u)deg(v)}{2m})$$

Proposition

Given a clustering $\mathcal{C} = C_1, ...,$ of a graph G with $|V| = n, |E| = m$, the modularity $M(\mathcal{C})$ of $\mathcal{C}$ is equal to:

$$M(\mathcal{C}) = \sum_{C \in \mathcal{C}} (\frac{|E(C)|}{m} - (\frac{\sum_{u \in C} deg(u)}{2m})^2)$$

where E(C) are the edges between nodes in cluster C: $E(C) = \{(u, v) \in E : u \in C, v \in C\}$

Proof:

$$M(\mathcal{C}) = \frac{1}{2m} \sum_{c \in \mathbb{C}} \sum_{u,v \in C} (A_{uv} - \frac{deg(u)deg(v)}{2m})$$

$$\frac{1}{2m} \sum_{C \in \mathbb{C}} \sum_{u,v \in C} A_{u,v} = \frac{1}{m} \sum_{C \in \mathcal{C}} |E(C)| = \sum_{C \in \mathcal{C}} \frac{|E(C)|}{m}$$

$$\frac{1}{2m} \sum_{C \in \mathbb{C}} \sum_{u,v \in C} \frac{deg(u)deg(v)}{2m}$$

$$= \sum_{C \in \mathcal{C}} \frac{1}{(2m)^2} (\sum_{u,v \in C} deg(u)deg(v))$$

$$= \sum_{C \in \mathcal{C}} \frac{1}{(2m)^2} (\sum_{v \in C} (deg_(v)) (\sum_{u \in C} deg(u)))$$

$$= \sum_{C \in \mathcal{C}} \frac{1}{(2m)^2} ((\sum_{v \in C} deg(v)) (\sum_{u \in C} deg(u)))$$

$$= \sum_{C \in \mathcal{C}} \frac{(\sum_{u \in C} deg(u))^2}{(2m)^2} = \sum_{C \in \mathcal{C}} (\frac{\sum_{u \in C} deg(u)}{2m})^2$$

Modularity based clustering

The input is the graph G, and the goal is to find the clustering $\mathcal{C} = C_1, ...$ that maximises the modularity

$$M(\mathcal{C}) = \sum_{C \in \mathcal{C}} (\frac{|E(C)|}{m} - (\frac{\sum_{u \in C} deg(u)}{2m})^2)$$

We can write an equivalent formulation where we consider to find the minimum value of $-M(\mathcal{C})$. Informally finding a clustering of maximum modularity is hard.

Problem (Modularity Clustering Problem)

Given a graph G and a value K, is there a clustering $\mathcal{C}$ of G such that $M(\mathcal{C}) \geq K$

The modularity clustering problem is NP-complete.

GreeduModularityClustering(G))

Input: connected graph G

Output: clustering of the elements of V

1 $\mathcal{C}_1 \leftarrow$clustering where each node u is assigned to its own clustering $C_u; i \leftarrow 1$;

2 repeat untill all nodes are in a single cluster:

3 for each pair of clusters C_1,C_2 such that there exists one edge between C_1 and C_2: compute $\delta(\mathcal{C}_i, C_1, C_2) = M(\mathcal{C}_i - C_1 - C_2 + (C_1 \cup C_2)) - M(\mathcal{C}_i)$;

4 find C',C" that maximize $\delta(\mathcal{C}, C', C'')$

5 $\mathcal{C}_{i+1} \leftarrow \mathcal{C}_i - C' - C'' + (C' \cup C''); i \leftarrow i + 1$;

6 return the clustering $\mathcal{C}^*$, across iterations, of maximum modularity: $\mathcal{C}^* = \text{argmax}_{\mathcal{C}_i, i=1,2,..} M(\mathcal{C}_i)$

The complexity in general is $O(|E||V|)$.

Proposition

Let $E(C_1, C_2)$ be the edges between cluster C_1 and cluster C_2: $E(C_1, C_2) = \{(u, v) \in E : u \in C_1, v \in C_2\}$. Then

$$\delta(\mathcal{C}_i, C_1, C_2) = \frac{|E(C_1, C_2)|}{m} - \frac{(\sum_{u \in C_1} deg(u))(\sum_{v \in C_2} deg(v))}{2m^2}$$

Proof: By definition:

$$\delta(\mathcal{C}_i, C_1, C_2) = M(\mathcal{C}_i - C_1 - C_2 + (C_1 \cup C_2)) - M(\mathcal{C}_i) = (\sum_{C \in \mathcal{C}_i - C_1 - C_2 + (C_1 \cup C_2)} M(C)) - (\sum_{C \in \mathcal{C}_i} M(C))$$

$$= M(C_1 \cup C_2) - (M(C_1) + M(C_2))$$

$$= \frac{|E(C_1 \cup C_2)|}{m} - (\frac{\sum_{u \in C_1 \cup C_2} deg(u)}{2m})^2 - (\frac{|E(C_1)|}{m} - (\frac{\sum_{u \in C_1} deg(u)}{2m})^2 + \frac{|E(C_2)|}{m} - (\frac{\sum_{u \in C_2} deg(u)}{2m})^2)$$

$$= \frac{|E(C_1 \cup C_2)| - |E(C_1)| - |E(C_2)|}{m} - (\frac{(\sum_{u \in C_1} deg(u)) + (\sum_{u \in C_2} deg(u))}{2m})^2 +$$

$$+ (\frac{\sum_{u \in C_1} deg(u)}{2m})^2 + (\frac{\sum_{u \in C_2} deg(u)}{2m})^2)$$

$$= \frac{|E(C_1, C_2)|}{m} - \frac{(\sum_{u \in C_1} deg(u))(\sum_{u \in C_2} deg(u))}{2m^2}$$

Proposition

In every iteration of the repeat-until loppp, the values $|E(C_1, C_2)|$ for all $C_1, C_2 \in \mathcal{C}$ and $\sum_{u \in C} deg(u)$ for all $C \in \mathcal{C}$ can be efficiently updated in total time $O(|E|)$.

Proof: At the beginning: $\forall u \in V$ there is a cluster C_u, therefore $|E(C_1, C_2)|$ depends on 1 edge and the degree can be computed in time $\Theta(m)$;

Assume vales are computed correctly before an iteration starts. In such iteration $C', C'' \to C' \cup C''$. For all pairs C_1, C_2 that do not include $C' \cup C''$: everything is already computed. Consider a pair of type $C_1, C' \cup C''$:

$E(C_1, C' \cup C'') = E(C_1, C') \cup E(C_1, C'')$ and the intersection is empty. Then $|E(C_1, C' \cup C'')| = |E(C_1, C')| + |E(C_1, C'')|$. Since there are $\leq |V|$ pairs of the type $C_1, C' \cup C''$, the computation is done in time $O(|E|)$.

$\forall$ clusters $C_1 \neq C', C''$, the degree of u is already computed. For the union of C' and C" the degree is already available from the previous iterations, then this step is $\Theta(1)$. (Note to Future Francesco, for degree i mean $\sum_{u \in C} deg(u)$ something like this).