无向图Density = 2*|E| / (|V| * |V-1|)
有向图无自环Density = |E| / (|V| * |V-1|)
有向图无自环Density = |E| / (|V|^2)

Compressed Sparse Row (CSR)

1. 求Adj list
2. 把adj list按顺去放进edge array
3. 将每个edge array的切片放进vertex array，
4. 注意结尾是开闭合，index + 1

Edge ID: 0 1 2 3 4 5 6 7 8
edge−array = [0, 1, 1, 2, 0, 2, 3, 1, 3]

vertex−array = [0, 2, 4, 7, 9]
Vertex ID: 0 1 2 3

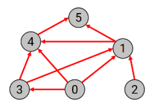
拓扑排序 (Topological Sort)

1. 标注每个node 的 in deg
2. 删除 in deg 为 0 的node，并放进order list
3. 持续删除，持续更新 indeg

检测图中是否存在环

- 如果在拓扑排序过程中，所有顶点都被处理完毕，
- 但队列为空，说明图中存在环。
- 时间复杂度：O(|V|+|E|)

Transitive closure



The original graph G

	0	1	2	3	4	5
0	1	1	0	1	1	1
1	0	1	0	0	1	1
2	0	1	1	0	1	1
3	0	1	0	1	1	1
4	0	0	0	0	1	1
5	0	0	0	0	0	1

The transitive closure of G

图的同态（Graph Homomorphism）

- **定义**: 存在一个函数

$$f : V_G \to V_H$$

若 $\{a, b\} \in E_G$，则 $\{f(a), f(b)\} \in E_H$。

- **直观**: 保持边结构的映射（不要求一一对应）。
- **用途**: 映射结构，表达"G 可嵌入 H"。

Graph Isomorphism （图的同构）

- **定义**: 存在一个**双射**

$$f : V_G \leftrightarrow V_H$$

保边：$\{a, b\} \in E_G \iff \{f(a), f(b)\} \in E_H$。

- **直观**: 结构完全相同，只是顶点标签不同。
- **用途**: 判断两图是否"本质上一样"。

Subgraph Matching:

在大图里找一个与查询图 isomorphic 的子图

Triangle Counting

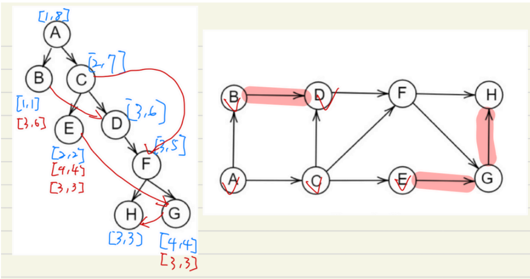
Algorithm 1: CF(G)
Input : G : an undirected graph
Output : All triangles in G
1 $G \leftarrow$ Orientation graph of G based on degree-order;
2 **for** each vertex $u \in G$ **do**
3   **for** each out-going neighbor v **do**
4     $T \leftarrow N^+(u) \cap N^+(v)$;
5   **for** each vertex $w \in T$ **do**
6   Output the triangle (u, v, w);

k-core

- **定义**: 最大的子图，使得每个顶点的度数 ≥ k。
- **算法**: 反复删除度 < k 的顶点。
- **复杂度**: $O(n+m)$。
- **用途**: 找"稳定圈子"，社交网络中的活跃群体。

Tree Cover

1. 用Leaf-Right-Root顺序遍历树，进行index标记
2. 确定每个node的可到达范围，如[1,1], [1,8]
3. 比较Graph，补全tree上面没有的edge，但是不要重复
4. 同时更新可到达范围



Betweenness Centrality

$$BC(v) = \sum_{s,t} \frac{\text{经过 v 的最短路径数}}{\text{从 s 到 t 的最短路径总数}}$$

Closeness Centrality

$$CC(v) = \frac{1}{\sum_u d(v,u)}$$

Graphlet degree vector

1. 会给几个小pattern和graph匹配
2. pattern里面的数字就是vector的index
3. 数字也代表了以这个node为root去matching
4. 数的时候记得graph会张开，不要给graph设限

Clustering Coe

$$C(v) = \frac{2 \times (\text{邻居之间实际存在的边数})}{d(v) \times (d(v)-1)}$$

Weisfeiler–Lehman (WL) Kernel

$$K_{WL}(G,H) = \sum_{i=0}^{h} (\text{两个图在第 i 轮标签频率向量的内积})$$

for i in 1..n: → O(n)
for i in 1..n: for j in 1..n: → O(n^2)
for i=1; i<=n; i*=2: → O(log n)
for i=1..n: for j=1..i: → O(n^2/2)=O(n^2)
for i=1..n: while x>0: x/=2 → O(n log X)
排序内循环：有 sort() 就乘 log：O(n log n)；
若在循环里 sort(k) → O(n·k log k)
数组：访问/赋值 O(1)；插入/删除中间 O(n)
链表：头插/删 O(1)；按位访问 O(n)
哈希表：查/插/删 均摊 O(1)；最坏 O(n)
二叉堆（优先队列）：push/pop/decrease-key = O(log n)
平衡树(Map/Set)：查/插/删 O(log n)
并查集：find/union = O(α(n))（近似常数）

k-truss

其中每条边至少属于 (k−2) 个三角形
先要找到k-1 core，再看有没有满足三角形个数
如果k-1core是全连接（clique），那么一定就是truss

k-ECC

k-ECC是一个最大子图，其中删除任意 k-1 条边后，图仍然保持连通。
每个 k-ECC 都是 (k-1)-核的一个子图，但每个(k-1)-核不一定是 k-ECC。

Pagerank

Used to determine the importance of a document based on the number of references to it and the importance of the source documents themselves.

A = A given page
T_1 …. T_n = Pages that point to page A (citations)
d = Damping factor between 0 and 1 (usually kept as 0.85)
C(T) = number of links going out of T
PR(A) = the PageRank of page A (initialized as 1/N for each page)

$$PR(A) = \frac{1-d}{N} + d\left(\frac{PR(T_1)}{C(T_1)} + \frac{PR(T_2)}{C(T_2)} + \cdots + \frac{PR(T_n)}{C(T_n)}\right)$$

参数数量

Node2Vec/Deepwalk

$$\text{Params} = |V| \times d$$

GCN

$$\text{Params} = d_{in} \times d_{out} + d_{out}$$

GraphSAGE（CONCAT）

$$\text{Params} = 2 \times (d_{in} \times d_{out}) + d_{out}$$

GAT (单头, 1-head)，多头就总数*heads

$$\text{Params} = d_{in} \times d_{out} + 3 \times d_{out}$$

稠密图GNN选择

I would use Graph Attention Networks (GAT) for node classification on the dense graph, since attention can assign different weights to neighbors instead of uniform averaging as in GCN. To optimize, I would apply multi-hop attention so that important 2-hop neighbors are considered without being overwhelmed by noise from too many direct neighbors. In addition, I would add skip-connections to prevent over-smoothing and keep the original node features available across layers. These improvements ensure both scalability and better representation quality on dense graphs.

稀疏图GNN选择

For a sparse graph, I would use GCN for node classification, since message passing across neighbors can enrich nodes that originally have limited information. To optimize, I would add more layers to enlarge the receptive field so that even distant nodes can contribute useful information. In addition, I would use graph augmentation to introduce new edges, which alleviates the sparsity issue. These optimizations ensure that nodes with few neighbors still receive sufficient context for classification.

Multi-hop attention	跨层考虑多跳邻居，增强全局信息	常见于 GAT 扩展
Skip-connection	类似 ResNet，让信息跨层传递	缓解 over-smoothing 问题
Additional layers	堆叠多层 GNN，捕捉更高阶邻居信息	但层数太多会 **过平滑**
Dropout	随机丢弃特征或边，防止过拟合	经典正则化手段
Graph augmentation	数据增强，比如随机删边、加边	提高模型鲁棒性

Locality Theorem

对一个顶点 v，若它的 core number = k，那么：

1. 它至少有 **k 个邻居的 core number ≥ k**；
2. 它不可能有 **k+1 个邻居的 core number ≥ (k+1)**。

2 hop cover

1. 标注每个node的out deg
2. 从大到小排序
3. 用表表示Lin和Lout
4. 每轮到一个node，在它对应的Lin和Lout里面放上自己
5. 注意不要重复

Model	学习方式	消息传递机制	是否需要全局结构	特征层级 (Node/Edge/Structural)	复杂度	稠密/稀疏适用	典型应用场景
Node2Vec	Transductive	随机游走 + Skip-gram	✅ 需要全图随机游走	Node ✅, Edge ❌, Structural ✅（通过游走）	**低-中**（取决于游走数）	**稀疏图更高效**（游走路径更有区分度）	社交网络嵌入、推荐系统、节点相似性
GCN	Transductive	邻居加权平均（卷积）	✅ 需要全图邻接矩阵	Node ✅, Edge ❌, Structural ✅（局部卷积）	**中**	**稠密/稀疏都可**，但大规模稀疏图需采样优化	引文网络分类、社区检测
GraphSAGE	Inductive	邻居采样 + 聚合（Mean/Pool/LSTM）	❌ 仅需局部子图	Node ✅, Edge ❌, Structural ✅（采样模式）	**低-中**（受采样数限制）	**稀疏图更合适**（减少采样偏差）	大规模动态图、社交推荐、归纳任务
GAT	Inductive	邻居注意力加权（self-attention）	❌ 局部计算即可	Node ✅, Edge ✅（注意力≈边权）, Structural 弱	**中-高**（注意力计算开销大）	**稀疏图更友好**（稠密图注意力开销大）	节点分类、异质图（不同重要性边）、推荐系统

spanning forest

- 是包含n-1条边的边集合，这些边连接所有的 n个顶点。

```
function Q1(adj):
    n ← length(adj)
    visited[0..n-1] ← false
    ST ← empty list of edges
    for s in [0..n-1]:
        if not visited[s]:
            visited[s] ← true
            Q ← empty queue
            enqueue(Q, s)
            while Q not empty:
                u ← dequeue(Q)
                for v in adj[u]:
                    if not visited[v]:
                        visited[v] ← true
                        enqueue(Q, v)
                        append(ST, [u, v])
    return ST
```

初始化 visited：O(n)

外层循环：每个顶点最多被设为已访问一次。

扫描邻接表：每条边在无向图里被查看两次（从两个端点各一次），合计 O(m)。

总时间：O(n + m)

空间：visited 与队列/栈 O(n)，结果边集 O(n)。

DIJKSTRA(Adj, n, source): O((n+m) log n)

```
for v in 0..n-1:
    dist[v] ← +∞
    parent[v] ← NIL
    visited[v] ← false
dist[source] ← 0

pq ← empty min-heap
heap_push(pq, (0, source))

while pq not empty:
    (d, u) ← heap_pop(pq)
    if visited[u]: continue
    visited[u] ← true

    for (v, w) in Adj[u]:
        if visited[v]: continue
        if d + w < dist[v]:
            dist[v] ← d + w
            parent[v] ← u
            heap_push(pq, (dist[v], v))

return dist, parent
```

Union Find(判断两点连通)

```
MAKE-SET(x):
    parent[x] = x
    rank[x] = 0

FIND(x):
    if parent[x] ≠ x:
        parent[x] = FIND(parent[x])
    return parent[x]

UNION(x, y):
    rootX = FIND(x)
    rootY = FIND(y)
    if rootX == rootY: return
    if rank[rootX] < rank[rootY]:
        parent[rootX] = rootY
    else if rank[rootX] > rank[rootY]:
        parent[rootY] = rootX
    else:
        parent[rootY] = rootX
        rank[rootX] = rank[rootX] + 1

CONNECTED(x, y):
    return FIND(x) == FIND(y)
```

KRUSKAL(E, n): O(m log m)

```
sort E by weight ascending

MAKE_SET(0..n-1)
T ← ∅

for (u, v, w) in E:
    ru ← FIND(u)
    rv ← FIND(v)
    if ru ≠ rv:
        T.add( (u, v, w) )
        UNION(ru, rv)
        if |T| = n - 1: break

return T
```

生成树的权重 (Weight of a Spanning Tree)：
在加权图中，生成树的权重是构成该生成树的所有边的权重之和。
最小生成树 (Minimum Spanning Tree)：
权重最小的生成树称为最小生成树。
Prim算法和Kruskal算法是寻找最小生成树的两种常用方法。

TOPOLOGICAL_SORT(G): O(V+E)

```
mark all vertices as unvisited
order = []

function DFS(u):
    mark u as visited
    for each neighbor v of u:
        if v is unvisited:
            DFS(v)
    prepend u to order

for each vertex v in G:
    if v is unvisited:
        DFS(v)

return order
```

Connected Component Detection（连通分量检测）

CONNECTED_COMPONENTS(G):

```
initialize UnionFind over all vertices

for each edge (u,v) in G:
    Union(u,v)

components = group vertices by Find(v)
return components
```

CONNECTED_COMPONENTS(G):

```
mark all vertices as unvisited
components = []
for each vertex v in G:
    if v not visited:
        current_comp = []
        DFS(v, current_comp)
        components.append(current_comp)
return components
```

DFS(u, current_comp):

```
mark u as visited
add u to current_comp
for each neighbor v of u:
    if v not visited:
        DFS(v, current_comp)
```

PRIM_HEAP(Adj, n, start=0): 稀疏图 O(m log n)

```
inMST[0..n-1] ← false
key[0..n-1]   ← +∞
parent[0..n-1] ← NIL
key[start] ← 0

pq ← empty min-heap
heap_push(pq, (0, start))

while pq not empty:
    (ku, u) ← heap_pop(pq)
    if inMST[u]: continue
    inMST[u] ← true

    for (v, w) in Adj[u]:
        if !inMST[v] and w < key[v]:
            key[v] ← w
            parent[v] ← u
            heap_push(pq, (key[v], v))

T ← ∅
for v from 0 to n-1:
    if parent[v] ≠ NIL:
        T.add( (parent[v], v, key[v]) )
return T
```

BFS(G, s):

```
for each vertex v in G:
    visited[v] = false
    dist[v] = ∞
    parent[v] = NIL
visited[s] = true
dist[s] = 0
Q = empty queue
enqueue(Q, s)

while Q not empty:
    u = dequeue(Q)
    for each neighbor v of u:
        if not visited[v]:
            visited[v] = true
            dist[v] = dist[u] + 1
            parent[v] = u
            enqueue(Q, v)
```

PRIM_MATRIX(W, n):O(n^3) 稠密图

```
inMST[0..n-1] ← false
key[0..n-1]   ← +∞
parent[0..n-1] ← NIL
key[0] ← 0

for i from 1 to n:
    u ← argmin_{v | !inMST[v]} key[v]
    inMST[u] ← true

    for v from 0 to n-1:
        if !inMST[v] and W[u][v] < key[v]:
            key[v] ← W[u][v]
            parent[v] ← u

T ← ∅
for v from 1 to n-1:
    if parent[v] ≠ NIL:
        T.add( (parent[v], v, W[parent[v]][v]) )
return T
```

FLOYD_WARSHALL(dist, n): O(n^3)

```
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            if dist[i][j] > dist[i][k] + dist[k][j]:
                dist[i][j] ← dist[i][k] + dist[k][j]
```

Tranagle Counting O(n^3)

```
for each triple (u, v, w):
    if (u,v), (v,w), (w,u) are edges:
        count++
```

A_STAR(G, start, goal, h): O(m log n)

```
for each v in V(G):
    g[v] ← +∞
    f[v] ← +∞
    parent[v] ← NIL
g[start] ← 0
f[start] ← h(start)

OPEN ← min-heap keyed by f[]
push(OPEN, (f[start], start))
CLOSED ← ∅

while OPEN not empty:
    (_, u) ← pop_min(OPEN)
    if u = goal:
        return RECONSTRUCT(parent, goal)

    add u to CLOSED

    for each (v, w) in G.adj[u]:
        if v in CLOSED: continue
        tentative ← g[u] + w
        if tentative < g[v]:
            parent[v] ← u
            g[v] ← tentative
            f[v] ← g[v] + h(v)
            push_or_decrease(OPEN, (f[v], v))
return FAIL
RECONSTRUCT(parent, t):
    path ← []
    while t ≠ NIL:
        prepend(path, t)
        t ← parent[t]
    return path
```

结构	空间	邻接查询 (u,v)?	扫描邻居	插边	删边	代表场景
Adj Matrix	O(n²)	O(1)	O(n)	O(1)	O(1)	稠密/小图；频繁存在性查询
Adj List	O(n+m)	O(deg)	快	O(1)	O(deg)	BFS/DFS、SSSP；一次性遍历多
Adj Set (hash)	O(n+m)	O(1) 摊还	一般	O(1)	O(1)	动态图；频繁判重/插删边
CSR	O(n+m)	需在本行二分/扫描	最快	差	差	静态+多轮 SpMV（PageRank）