R11 Weighted Graphs

3.29 https://github.com/GUMI-21/MIT6.006_note

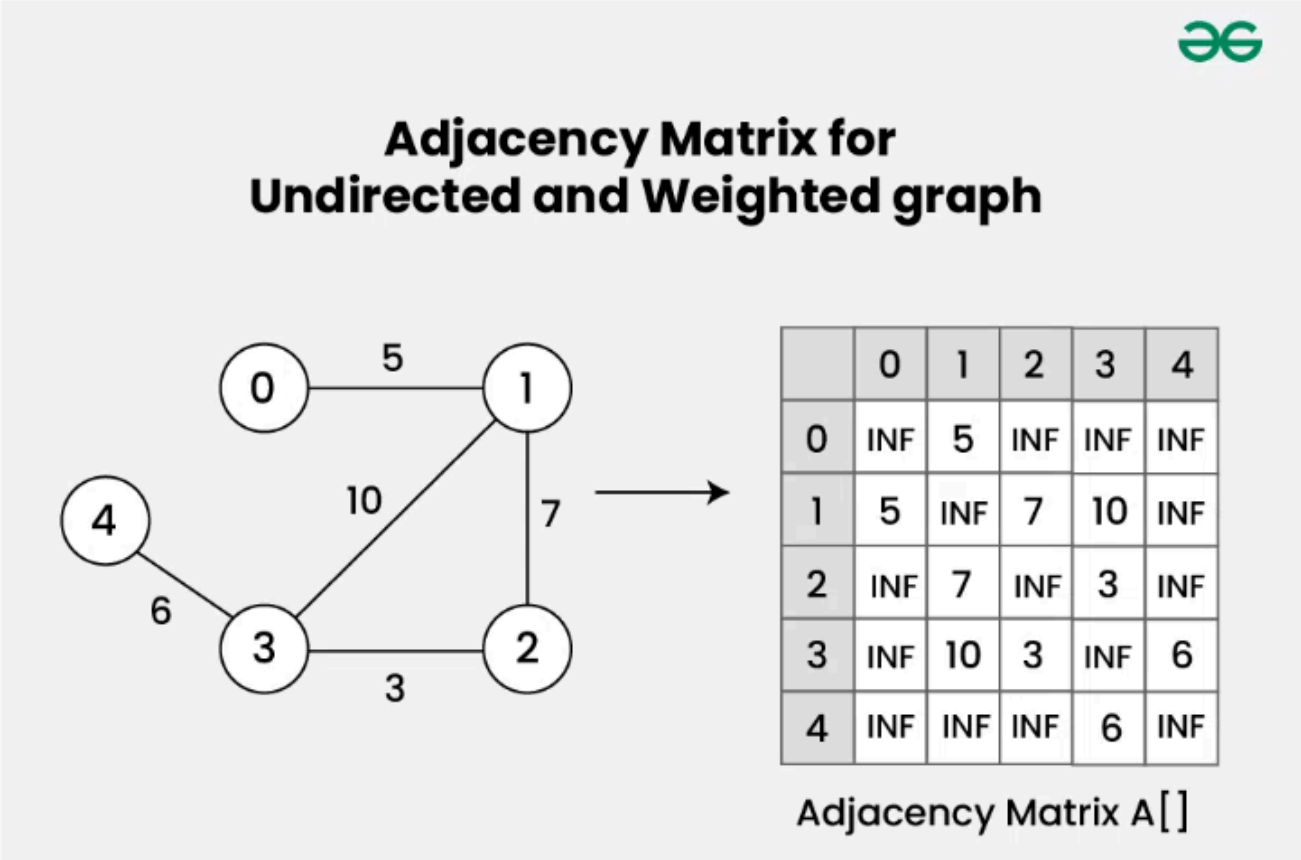
Weighted Graphs

associate a numerical weight to edges in a graph.
A weighted graph is then a graph $G = (V, E)$ together with a *weight function* $w : E \to R$, mapping edges to real-valued weights.
or store each weight as a value in an *adjacency matrix*, or inside an edge object stored in an *adjacency list* or set.

```
W1 = [0:{1:-1}, 1:{2,0}, 2:{0,:1}, 3:{4:3}}
W2 = {0:{1:1, 3:2, 4:-1}, 1:{0:1}, 2:{3,0}, 3:{0:2, 2:0}, 4:{0:-1}}
def w(u,v): return W[u][v]
```

w can be stored using O(|E|) space, and return in constant time.

- Adjacency Matrix



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF

Weighted Shortest Path

the weight of the path is the sum of the weights from edges in the path.

- *notation*
  $\pi = (v_1, \ldots, v_k)$ is a weighted path,
  then let $w(\pi)$ denote the path's weight $\sum_{i=1}^{k-1} w(v_i, v_i + 1)$
- undifined path
  when a graph contains a negative weight cycle, we will say the shortest path from s to v is undifined, with weight $-\infty$.
  If no path exists from s to v, then we will say the shortest path from s to v is undifined, with weight $+\infty$
- *Algorithm*
  when all edges have same positive weight we can use BFS to slove shortest weight path.
  Weighted Signle Source Shortest Path Algorithms (WSSSP)

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time $O(\cdot)$
General	Unweighted	BFS	$\lvert V\rvert + \lvert E\rvert$
DAG	Any	DAG Relaxation	$\lvert V\rvert + \lvert E\rvert$
General	Any	Bellman-Ford	$\lvert V\rvert \cdot \lvert E\rvert$
General	Non-negative	Dijkstra	$\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert$

DAG Relaxation

one relaxation wa shown in LEC, this is another framework.
Relaxation: As a general algorithmic paradigm, a relaxation algorithm searches for a solution to an optimization problem by starting with a solution that is not optimal, then iteratively improves the solution until it becomes an optimal solution to the original problem.

- *Algorithm*
  During the relaxation algorithm, we will repeatedly relax some path estimate d(s, v), decreasing it toward the true shortest path weight δ(s, v).
  If ever d(s, v) = δ(s, v), we say that estimate d(s, v) is fully relaxed.

```
def gengeral_relax(Adj, w, s): # adj: Adjacenry list, w:weight, s: start
    d = [float('inf') for _ in Adj] # shortest path estimates d(s,v)
    parent = [None for _ in Adj]    # parent pointers
    d[s], parent[s] = 0,s
    while True:
```

```
        relax some d[v] ??   # relax a shortest path estimate d(s,v)
    return d, parent         # return weights, paths via parents.
```

if at any time d(s, u) + w(u, v) < d(s, v), we can relax the edge by setting d(s, v) = d(s, u) + w(u, v), strictly improving our shortest path estimate.

```
def try_to_relax(Adj, w, d, parent, u, v):
    if d[v] > d[u] + w[u, v]:
        d[v] = d[u] + w[u,v]
        parent[v] = u
```

If ever we arrive at an assignment of all shortest path estimates such that no edge in the graph can be relaxed, then we can prove that shortest path estimates are in fact shortest path distances.

- *Termination Lemma*: If no edge can be relaxed, then d(s, v) ≤ δ(s, v) for all v ∈ V .
  Proof. Suppose for contradiction δ(s, v) < d(s, v) so that there is a shorter path π from s to v. Let (a, b) be ther first edge of π such that d(b) > δ(s, b). Then edge (a, b) can be relaxed, a contradiction.
  =>

```
while some_edge_relaxable(Adj, w, d):
    (u, v) = get_relaxable_edge(Adj, w, d)
    try_to_relax(Adj, w, d, parent, u, v)
```

in DAG Relaxation

```
def DAG_Relaxation(Adj,w,s):
    _, order = dfs(Adj, s)   # run depth-first search on graph
    order.reverse()          # reverse returned order
    d = [float['inf'] for _ in Adj]  # shortest path estimates d(s,v)
    parent = [None for _ in Adj]
    d[s], parent[s] = 0, s
    for u in odrder:
        for v in Adj[u]:
            try_to_relax(Adj, w, d, parent, u, v)
    return d, parent

def try_to_relax(Adj, w, d, parent, u, v):
    if d[v] > d[u] + w[u, v]:
        d[v] = d[u] + w[u,v]
        parent[v] = u
```

obey the topological order

The topological sort order ensures that edges of the path are relaxed in the order in which they appear in the path.
Proofed by induction

- runtime
  Since depthfirst search runs in linear time and the loops relax each edge exactly once, this algorithm takes O(|V | + |E|) time