Graph and tree algorithms cheat sheet

0. Core Setup: Graph Representations

Adjacency List (sparse graphs – most common)

```python
from collections import defaultdict

# Directed or undirected depending on whether you add both ways
graph = defaultdict(list)  # node -> list[(neighbor, weight)]

def add_edge(u, v, w=1, undirected=False):
    graph[u].append((v, w))
    if undirected:
        graph[v].append((u, w))
```

- **Space:** `O(n + m)`
- **Use for:** almost everything (BFS/DFS, Dijkstra, MST, etc.)

Adjacency Matrix (dense graphs / small n)

```python
INF = float('inf')
n = 5
mat = [[INF]*n for _ in range(n)]
for i in range(n):
    mat[i][i] = 0

def add_edge_matrix(u, v, w=1, undirected=False):
    mat[u][v] = min(mat[u][v], w)
    if undirected:
        mat[v][u] = min(mat[v][u], w)
```

- **Space:** `O(n²)`
- **Use for:** Floyd–Warshall, dense graphs, or `n ≤ 500` kind of problems.

---

1. Breadth-First Search (BFS)

Classic BFS (unweighted shortest path)

```python
from collections import deque

def bfs(start, graph):
    dist = {start: 0}
    parent = {start: None}
    dq = deque([start])

    while dq:
        u = dq.popleft()
        for v, _ in graph[u]:  # ignore weights
            if v not in dist:
                dist[v] = dist[u] + 1
                parent[v] = u
                dq.append(v)
    return dist, parent
```

- **Time:** `O(n + m)`
- **Use for:**
  - Shortest path with **equal weights** (e.g., edges cost 1).
  - Level-order traversal, computing distances in unweighted graphs.

Pattern: Multi-source BFS

Start from **multiple nodes simultaneously**.

```python
def multi_source_bfs(starts, graph):
    from collections import deque
    dq = deque()
    dist = {}
    for s in starts:
        dist[s] = 0
        dq.append(s)
    while dq:
        u = dq.popleft()
        for v, _ in graph[u]:
            if v not in dist:
                dist[v] = dist[u] + 1
                dq.append(v)
    return dist
```

- **Use for:** nearest special node, distance to closest shop/hospital, etc.

Pattern: 0–1 BFS (edges with weight 0 or 1)

```python
from collections import deque

def zero_one_bfs(start, graph):
    dist = {start: 0}
    dq = deque([start])

    while dq:
        u = dq.popleft()
        for v, w in graph[u]:
            nd = dist[u] + w
            if v not in dist or nd < dist[v]:
                dist[v] = nd
                if w == 0:
                    dq.appendleft(v)
                else:
                    dq.append(v)
    return dist
```

- **Time:** `O(n + m)`
- **Use for:** grids with 0/1 cost, binary transformations, etc.

---

2. Depth-First Search (DFS)

Iterative DFS (avoids recursion limit)

```python
def dfs_iter(start, graph):
    visited = set()
    parent = {start: None}
    stack = [start]

    while stack:
        u = stack.pop()
        if u in visited:
            continue
        visited.add(u)
        for v, _ in reversed(graph[u]):  # reversed = DFS order similar to
recursive
            if v not in visited:
                parent[v] = u
                stack.append(v)
    return visited, parent
```

- **Time:** `O(n + m)`
- **Use for:** connectivity, topological sort, cycle detection, etc.

Pattern: Connected Components (Undirected)

```python
def connected_components(nodes, graph):
    visited = set()
    comps = []

    for s in nodes:
        if s in visited:
            continue
        stack = [s]
        comp = []
        while stack:
            u = stack.pop()
            if u in visited:
                continue
            visited.add(u)
            comp.append(u)
            for v, _ in graph[u]:
                if v not in visited:
                    stack.append(v)
        comps.append(comp)
    return comps
```

---

3. Shortest Paths

Dijkstra (non-negative weights)

```python
import heapq

def dijkstra(start, graph):
    dist = {start: 0}
    parent = {start: None}
    pq = [(0, start)]  # (dist, node)

    while pq:
        d, u = heapq.heappop(pq)
        if d != dist[u]:  # stale entry
            continue
        for v, w in graph[u]:
            nd = d + w
            if v not in dist or nd < dist[v]:
                dist[v] = nd
                parent[v] = u
                heapq.heappush(pq, (nd, v))
    return dist, parent
```

- **Time:** `O(m log n)`
- **Use for:** positive edge weights, single-source shortest paths.

Pattern: Reconstructing the Path

```python
def reconstruct_path(parent, target):
    if target not in parent:
        return None
    path = []
    cur = target
    while cur is not None:
        path.append(cur)
        cur = parent[cur]
    return path[::-1]
```

Bellman–Ford (handles negative weights, detects negative cycles)

```python
def bellman_ford(n, edges, start):
    # edges: list of (u, v, w)
    INF = float('inf')
    dist = [INF] * n
    dist[start] = 0

    for _ in range(n - 1):
        updated = False
        for u, v, w in edges:
            if dist[u] != INF and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                updated = True
        if not updated:
            break

    # detect negative cycles
    in_neg_cycle = [False] * n
    for _ in range(1):  # one more relaxation is enough to detect
        for u, v, w in edges:
            if dist[u] != INF and dist[u] + w < dist[v]:
                dist[v] = -INF
                in_neg_cycle[v] = True

    return dist, in_neg_cycle
```

- **Time:** `O(nm)`
- **Use for:** negative edges, but no negative cycles reachable from source.

Floyd–Warshall (all-pairs shortest path)

```
def floyd_warshall(mat):
    # mat[i][j] initialized with INF or weights, mat[i][i] = 0
    n = len(mat)
    for k in range(n):
        for i in range(n):
            if mat[i][k] == float('inf'):
                continue
            for j in range(n):
                if mat[k][j] == float('inf'):
                    continue
                if mat[i][k] + mat[k][j] < mat[i][j]:
                    mat[i][j] = mat[i][k] + mat[k][j]
    return mat
```

- **Time:** `O(n³)`
- **Use for:** dense graphs, all-pairs shortest paths, or when `n` is small.

---

4. Minimum Spanning Tree (MST) – Undirected, Connected

Disjoint Set Union (Union-Find)

```
class DSU:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0]*n

    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x

    def union(self, x, y):
        xr, yr = self.find(x), self.find(y)
        if xr == yr:
            return False
        if self.rank[xr] < self.rank[yr]:
            xr, yr = yr, xr
        self.parent[yr] = xr
        if self.rank[xr] == self.rank[yr]:
            self.rank[xr] += 1
        return True
```

Kruskal’s Algorithm

```
def kruskal_mst(n, edges):
    # edges: list of (w, u, v)
    edges.sort()
    dsu = DSU(n)
    mst_weight = 0
    mst_edges = []

    for w, u, v in edges:
        if dsu.union(u, v):
            mst_weight += w
            mst_edges.append((u, v, w))
    return mst_weight, mst_edges
```

- **Time:** `O(m log m)`
- **Use for:** static edge list; easy and fast.

Prim’s Algorithm (with heap)

```
import heapq
from collections import defaultdict

def prim_mst(start, graph, n):
    visited = [False]*n
    pq = [(0, start, -1)]  # (weight, node, parent)
    mst_weight = 0
    mst_edges = []

    while pq and len(mst_edges) < n - 1:
```

```python
            w, u, p = heapq.heappop(pq)
            if visited[u]:
                continue
            visited[u] = True
            mst_weight += w
            if p != -1:
                mst_edges.append((p, u, w))
            for v, vw in graph[u]:
                if not visited[v]:
                    heapq.heappush(pq, (vw, v, u))

        return mst_weight, mst_edges
```

- **Time:** `O(m log n)`

---

5. DAG Algorithms (Directed Acyclic Graphs)

Topological Sort (Kahn’s Algorithm – BFS with indegree)

```python
from collections import deque, defaultdict

def topo_sort(nodes, graph):
    indeg = defaultdict(int)
    for u in nodes:
        for v, _ in graph[u]:
            indeg[v] += 1

    dq = deque([u for u in nodes if indeg[u] == 0])
    order = []

    while dq:
        u = dq.popleft()
        order.append(u)
        for v, _ in graph[u]:
            indeg[v] -= 1
            if indeg[v] == 0:
                dq.append(v)

    if len(order) != len(nodes):
        raise ValueError("Graph has a cycle")
    return order
```

- **Use for:** course scheduling, DP on DAG, dependency resolution.

Longest Path in a DAG (weights can be positive)

```
def longest_path_in_dag(nodes, graph, topo_order, src):
    # graph: u -> list of (v, w)
    dist = {u: float('-inf') for u in nodes}
    dist[src] = 0

    for u in topo_order:
        if dist[u] == float('-inf'):
            continue
        for v, w in graph[u]:
            if dist[u] + w > dist[v]:
                dist[v] = dist[u] + w
    return dist
```

---

6. Strongly Connected Components (SCC) – Directed Graph

Kosaraju’s Algorithm (two DFS passes)

```
from collections import defaultdict

def kosaraju_scc(nodes, graph):
    # build reverse graph
    rgraph = defaultdict(list)
    for u in nodes:
        for v, _ in graph[u]:
            rgraph[v].append(u)

    visited = set()
    order = []

    # 1st pass: order by finish time
    def dfs1(start):
        stack = [start]
        it_stack = [iter(graph[start])]
        visited.add(start)
        while stack:
            u = stack[-1]
            for v, _ in it_stack[-1]:
                if v not in visited:
                    visited.add(v)
                    stack.append(v)
                    it_stack.append(iter(graph[v]))
                    break
            else:
                order.append(u)
                stack.pop()
                it_stack.pop()

    for u in nodes:
        if u not in visited:
            dfs1(u)

    # 2nd pass: on reversed graph
    visited.clear()
    comps = []

    def dfs2(start):
        stack = [start]
        comp = []
        visited.add(start)
        while stack:
            u = stack.pop()
            comp.append(u)
            for v in rgraph[u]:
                if v not in visited:
                    visited.add(v)
                    stack.append(v)
        return comp

    for u in reversed(order):
        if u not in visited:
            comps.append(dfs2(u))

    return comps
```

- **Use for:** condensation of graph into DAG, finding cycles, 2-SAT.

---

7. Lowest Common Ancestor (LCA) – Tree Pattern

Preprocessing via Binary Lifting

```python
import math
from collections import deque

def build_lca(root, n, tree):
    LOG = math.ceil(math.log2(n+1))
    up = [[-1]*n for _ in range(LOG)]
    depth = [0]*n

    # BFS to compute depth and immediate parent
    dq = deque([root])
    parent = [-1]*n
    parent[root] = -1
    while dq:
        u = dq.popleft()
        for v, _ in tree[u]:
            if v == parent[u]:
                continue
            parent[v] = u
            depth[v] = depth[u] + 1
            dq.append(v)

    for v in range(n):
        up[0][v] = parent[v] if parent[v] != -1 else v

    for k in range(1, LOG):
        for v in range(n):
            up[k][v] = up[k-1][up[k-1][v]]

    return up, depth, LOG

def lca(u, v, up, depth, LOG):
    if depth[u] < depth[v]:
        u, v = v, u

    # Lift u up
    diff = depth[u] - depth[v]
    for k in range(LOG):
        if diff & (1 << k):
            u = up[k][u]

    if u == v:
        return u

    # Lift both up until parents differ
    for k in reversed(range(LOG)):
        if up[k][u] != up[k][v]:
            u = up[k][u]
            v = up[k][v]

    return up[0][u]
```

- **Preprocessing:** `O(n log n)`
- **Query:** `O(log n)`

---

8. Common Graph Problem Patterns

1. Grid as Graph

Index `(r, c)` as node, edges to neighbours.

```
def grid_neighbors(r, c, R, C):
    for dr, dc in [(-1,0),(1,0),(0,-1),(0,1)]:
        nr, nc = r+dr, c+dc
        if 0 <= nr < R and 0 <= nc < C:
            yield nr, nc
```

Apply BFS/DFS/Dijkstra over `(r, c)` states.

---

2. Graph + State (multi-dimensional BFS/shortest path)

Node is `(position, extra_state)` – e.g., `(node, used_coupon)`.

```
from collections import deque

def bfs_state(start_state, transitions):
    # transitions(state) -> iterable of next_state
    dq = deque([start_state])
    dist = {start_state: 0}

    while dq:
        s = dq.popleft()
        for ns in transitions(s):
            if ns not in dist:
                dist[ns] = dist[s] + 1
                dq.append(ns)
    return dist
```

---

3. Topological DP Pattern

- Compute `topo_order`.
- Process in topological order, update DP for outgoing edges.

```
def dag_dp(nodes, graph, topo_order, base):
    dp = {u: base(u) for u in nodes}
    for u in topo_order:
        for v, w in graph[u]:
            dp[v] = max(dp[v], dp[u] + w)
    return dp
```

---

4. Cycle Detection

- **Undirected:** if you see a visited neighbour that is **not parent**, there is a cycle.
- **Directed:** DFS with 3-color marking (0 = unvisited, 1 = visiting, 2 = done).

```
def has_cycle_directed(nodes, graph):
    color = {u: 0 for u in nodes}  # 0=unvisited,1=visiting,2=done

    def dfs(u):
        color[u] = 1
        for v, _ in graph[u]:
            if color[v] == 1:
                return True
            if color[v] == 0 and dfs(v):
                return True
        color[u] = 2
        return False

    return any(color[u] == 0 and dfs(u) for u in nodes)
```

---

9. Quick “When to Use What” Table

Problem Type	**Typical Algorithm / Pattern**
Reachability / any path?	DFS / BFS
Shortest path, all edges same weight	BFS / multi-source BFS
Shortest path, non-negative weights	Dijkstra (heap)
Shortest path, may contain negative weights	Bellman–Ford / Johnson / DP on DAG
All-pairs shortest path, small dense graph	Floyd–Warshall
Network with 0/1 edge weights	0–1 BFS
Grouping connected nodes (undirected)	DFS/BFS components / DSU
Minimum spanning tree (undirected)	Kruskal (DSU) / Prim (heap)
Scheduling / ordering with dependencies	Topological Sort
Strongly connected clusters in directed graph	Kosaraju / Tarjan SCC
LCA / distance queries in tree	Binary Lifting / Euler Tour + RMQ
Grid mazes / maps	BFS / Dijkstra on grid graph
Two or more constraints / toggles	State-space BFS or Dijkstra on `(node, state)`