

Graph Algorithms: Dijkstra, A*, Topological Sort & Union-Find

Shortest-path selection matrix, Kahn's topological sort, Disjoint Set Union (DSU), and Tarjan's SCC.

EXECUTIVE SUMMARY

Complete decision matrix and production Python implementations for graph traversal, shortest paths, cycle detection, minimum spanning trees, and union-find with path compression.

KEY EQUATIONS & INVARIANTS

* Dijkstra Priority Queue Complexity:

$$\text{LaTeX: } T(V, E) = O(\bigl((V + E) \log V\bigr))$$

Note: Requires non-negative edge weights ($w(u, v) \geq 0$) so settled vertices never need relaxation.

* A* Heuristic Admissibility & Consistency:

$$\text{LaTeX: } 0 \leq h(u) \leq d^*(u, t) \quad \text{and} \quad h(u) \leq w(u, v) + h(v)$$

Note: Admissible and consistent heuristics guarantee A* expands the minimum number of nodes.

1. Shortest-Path Algorithm Selection Matrix

Select the shortest-path algorithm based on edge weight constraints and query cardinality.

Algorithm | Edge Weights | Time Complexity | Space | Best For

BFS | Unweighted ($w = 1$) | $O(V + E)$ | $O(V)$ | Minimum hops in unweighted graphs
 0-1 BFS (Deque) | $w \in \{0, 1\}$ | $O(V + E)$ | $O(V)$ | Grid mazes with free/cost-1 edges
 Dijkstra (Min-Heap) | $w \geq 0$ | $O((V + E) \log V)$ | $O(V)$ | Single-source non-negative road/network graphs
 Bellman-Ford | Any real w (detects neg cycles) | $O(V * E)$ | $O(V)$ | Arbitrage detection & K-stop constrained paths
 Floyd-Warshall | Any real w (no neg cycles) | $O(V^3)$ | $O(V^2)$ | All-pairs shortest paths on dense graphs ($V \leq 400$)

[PYTHON] Lazy-Deletion Dijkstra & Disjoint Set Union (Path Compression + Rank)

```
import heapq

def dijkstra(n: int, adj: list[list[tuple[int, int]]], src: int) -> list[ float]:
    dist = [float('inf')] * n
    dist[src] = 0.0
    pq: list[tuple[float, int]] = [(0.0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue # Skip stale heap entry
        for v, weight in adj[u]:
            if dist[u] + weight < dist[v]:
                dist[v] = dist[u] + weight
                heapq.heappush(pq, (dist[v], v))
    return dist

class UnionFind:
    def __init__(self, n: int) -> None:
        self.parent = list(range(n))
        self.size = [1] * n

    def find(self, x: int) -> int:
        while x != self.parent[x]:
```

```
        self.parent[x] = self.parent[self.parent[x]]
    x = self.parent[x]
    return x
```

COMMON PRODUCTION & INTERVIEW PITFALLS

- Forgetting `if d > dist[u]: continue` in Dijkstra causes $O(E^2)$ blowup on dense graphs when duplicate vertex entries accumulate in the heap.
- Marking nodes visited upon popping from the queue in BFS instead of upon enqueueing causes exponential duplicate queue insertions.

FREQUENTLY ASKED QUESTIONS

Q: How do I detect a cycle in a directed graph vs an undirected graph?

A: For a directed graph, use Kahn's in-degree topological sort (if processed count < V , a cycle exists) or 3-color DFS (White/Gray/Black). For an undirected graph, use Union-Find or parent-tracked DFS.