Single-Source Shortest Paths

📘 Single-Source Shortest Path (SSSP) — Exam Notes

1. What is the Shortest Path Problem?

- **Definition:** A path's length = sum of its edge weights.
- A **shortest path** from uuu to vvv is any path with minimum total weight.
- **SSSP (Single-Source Shortest Path):** Given graph GGG and source sss, find shortest paths from sss to **every vertex** in GGG.lecture14_sssp

2. Dijkstra's Algorithm

Procedure

1. Initialize: `dist[source]=0`, all others `∞`.
2. Put `(0, source)` in a priority queue.
3. While queue not empty:
    - Extract node `u` with **smallest tentative distance**.
    - If already settled, skip.
    - Settle `u`: its distance is now final.
    - Relax all edges `(u,v,w)`: if `dist[u]+w < dist[v]`, update `dist[v]` and push `(dist[v],v)` in queue.lecture14_sssp

Usage

- Solves **SSSP** in graphs with **non-negative edge weights**.
- Returns `dist[]` (shortest distances) and optionally `pred[]` (to reconstruct paths).

Why it works

- Once a node `u` is popped from the PQ, it has the **smallest tentative distance**.
- With **non-negative weights**, no alternative later path can make it smaller.
- Therefore `dist[u]` is final at that moment.

Complexity

- Each edge is pushed/popped at most once.
- With a **binary heap** PQ:
    - Push/pop = O(log ∣ V ∣)O(\log |V|)O(log ∣ V ∣).
    - At most ∣ E ∣ |E| ∣ E ∣ operations.
    - Total: **O(|E| log |V|)**.lecture14_sssp
- With a **Fibonacci heap**: improve to O(∣ E ∣ + ∣ V ∣ log ∣ V ∣)O(|E| + |V|\log |V|)O(∣ E ∣ + ∣ V ∣ log ∣ V ∣), but less practical.
- **Space:** store `dist[]`, `pred[]`, and PQ → O(∣ V ∣ + ∣ E ∣)O(|V| + |E|)O(∣ V ∣ + ∣ E ∣).

Limitations: negatives

- If graph has **negative edges**, Dijkstra fails:
    - Assumes once a vertex is settled, its distance is final.
    - A later negative edge can lower it, breaking correctness.
- **Negative cycles:** shortest path not defined (can reduce cost infinitely).lecture14_sssp

---

3. Bellman–Ford Algorithm

Procedure

1. Initialize: `dist[source]=0`, others `∞`.
2. Repeat **|V|-1 times**:
    - Relax every edge `(u,v,w)`.
    - If `dist[u]+w < dist[v]`, update `dist[v]` and set `pred[v]=u`.

3. After these rounds, check all edges once more:
    - If any edge can still be relaxed, a **negative cycle** exists.lecture14_sssp

Usage

- Solves **SSSP** even with **negative edges**.
- Detects if there is a **negative cycle** reachable from the source.

Why it works

- **Optimal substructure:** a shortest path of length kkk edges is built from shortest paths of length k−1k-1k−1.
- Relaxing all edges once finds all shortest paths with ≤1 edge, then ≤2 edges, …
- No shortest path needs more than ∣V∣−1|V|-1∣V∣−1 edges (since paths with cycles can't be shorter unless it's a negative cycle).
- After ∣V∣−1|V|-1∣V∣−1 rounds, all shortest paths are guaranteed found.lecture14_sssp

Complexity

- Inner loop relaxes all edges = O(∣E∣)O(|E|)O(∣E∣).
- Repeated ∣V∣−1|V|-1∣V∣−1 times.
- Total: **O(|V||E|)**.lecture14_sssp
- **Space:** store `dist[]`, `pred[]` = O(∣V∣)O(|V|)O(∣V∣).

---

4. Differences — Dijkstra vs Bellman–Ford

Feature	Dijkstra	Bellman–Ford
Handles negative weights?	❌ No (only non-negative)	✅ Yes
Detects negative cycles?	❌ No	✅ Yes
Time complexity	(O(	E

Feature	Dijkstra	Bellman–Ford
Space complexity	(O(	V
Why it works	Greedy: once settled, distance is final (non-negative edges)	DP-like: relax edges repeatedly until convergence

5. Intuition Summary

- **Dijkstra = Greedy:** Expands frontier by always choosing the next closest vertex. Works only if edge weights ≥ 0.
- **Bellman–Ford = Dynamic Programming:** Gradually improves estimates by relaxing all edges. Works even with negative edges, detects negative cycles, but slower because it doesn't assume anything about edge signs.

✅ With this, you now have:

- Clear procedure for both algorithms.
- When/why to use each.
- Complexity with reasoning tied to their implementation.
- Why negatives break Dijkstra and why Bellman–Ford still works.