Shortest Path Problems

Single-Source Shortest Path (SSSP)

Problems Covered

1. Shortest Path in an Unweighted Graph (Easy)
2. Shortest Distances with Non-Negative Weights (Easy - Medium)
3. Shortest Paths in a Weighted DAG (Medium)
4. Negative Weights and Negative Cycles (Medium)
5. Cheapest Flights Within K Stops (Medium - Hard)
6. Shortest Path with One Discount Coupon (Hard)

Shortest Path in an Unweighted Graph

Problem 1 | Easy

The Problem

- Given an undirected, **unweighted** graph with n vertices (0 to n−1) and a source vertex s, find the **shortest distance** (minimum number of edges) from s to every vertex.
- If a vertex cannot be reached from s, its distance is −1.

Example:

n = 7, s = 0

Edges: 0-1, 0-2, 1-3, 2-4, 3-4, 4-5

Output: [0, 1, 1, 2, 2, 3, −1]

Any ideas on how this can be solved?

Hints

- Every edge has the same cost. What does that say about the order in which vertices should be visited?
- Which data structure gives vertices back in the same order in which they were discovered: a stack or a queue?
- When v is reached for the first time from u, can a shorter way to v show up later?

The Solution

Key Idea

- All edges cost 1, so the first time we reach a vertex is along a shortest path.
- BFS visits vertices level by level: everything at distance k is processed before anything at distance k+1.

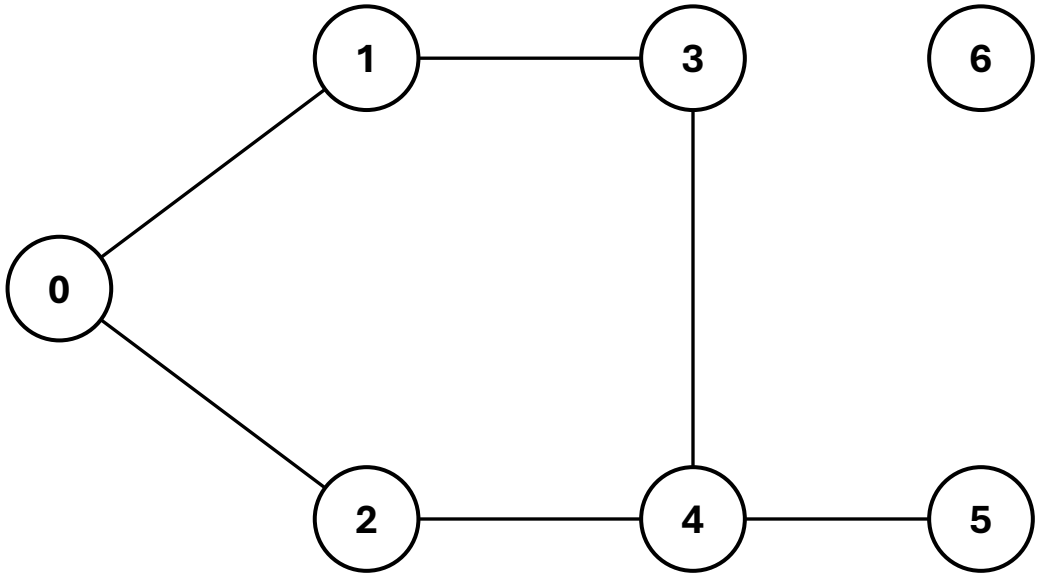
Steps

- Set dist[v] = −1 for all v, dist[s] = 0, push s into the queue.
- Pop u. For each neighbour v with dist[v] == −1: set dist[v] = dist[u] + 1 and push v.
- Repeat until the queue is empty.
- Vertices still at −1 are unreachable.

```
BFS(s):
  dist[all] = -1
  dist[s] = 0
  queue.push(s)
  while queue not empty:
    u = queue.pop()
    for v in adj[u]:
      if dist[v] == -1:
        dist[v] = dist[u] + 1
        queue.push(v)
  return dist
```

Example

n = 7, s = 0 (vertex 6 has no edges)



Pop	New distances	Queue
start	dist[0] = 0	[0]
0	dist[1] = 1, dist[2] = 1	[1, 2]
1	dist[3] = 2	[2, 3]
2	dist[4] = 2	[3, 4]
3	4 already visited	[4]
4	dist[5] = 3	[5]
5	no new vertex	[ ]

dist = [0, 1, 1, 2, 2, 3, −1]

Complexity Analysis

1. Time Complexity

- Each vertex enters the queue at most once, because dist[v] is set only the first time v is seen.
- Each edge is looked at when its endpoint is popped (twice in an undirected graph).
- Time Complexity = **O(V + E)**

2. Space Complexity

- dist[] array: O(V). Queue: at most O(V) vertices.
- Adjacency list of the graph: O(V + E).
- Space Complexity = **O(V)** extra (O(V + E) including the graph)

Shortest Distances with Non-Negative Weights

Problem 2 | Easy - Medium

The Problem

- Given a directed graph with n vertices and **non-negative edge weights**, and a source vertex s, find the shortest distance from s to every vertex.
- Also report the time needed for a signal sent from s to reach all vertices (the largest distance), or −1 if some vertex is unreachable.

Example:

n = 5, s = 0

Edges u → v (w): 0→1 (4), 0→2 (1), 2→1 (2), 1→3 (1), 2→3 (5), 3→4 (3)

Output: dist = [0, 3, 1, 4, 7], time to reach all = 7

Any ideas on how this can be solved?

Hints

- If every weight were 1, BFS would work. What goes wrong when weights differ? Is the first path found to a vertex always the cheapest?
- Among the vertices not finalised yet, which one can you be sure has reached its final distance?
- Once that vertex is finalised, what should you do with its outgoing edges?

The Solution

Key Idea (Dijkstra)

- Greedy: the unfinalised vertex with the smallest tentative distance cannot be improved any more, since no edge has negative weight.

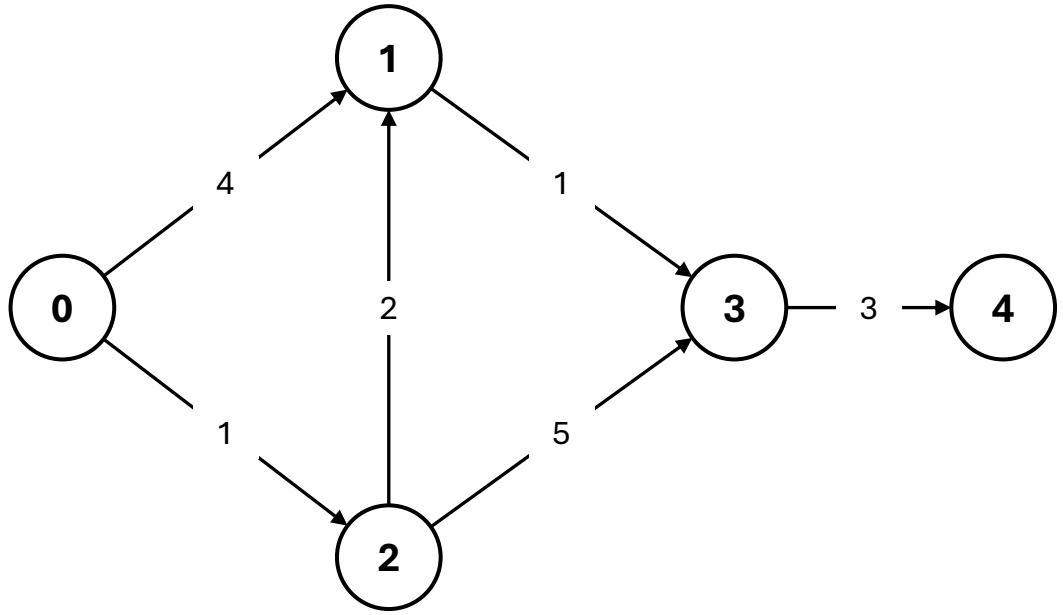
Steps

- dist[all] = ∞, dist[s] = 0, push (0, s) into a min-heap.
- Pop (d, u) with the smallest d. If d > dist[u], it is a stale entry: skip it.
- Relax every edge (u, v, w): if d + w < dist[v], update dist[v] and push (dist[v], v).
- Repeat until the heap is empty.

```
Dijkstra(s):
  dist[all] = INF
  dist[s] = 0
  pq.push((0, s))
  while pq not empty:
    (d, u) = pq.popMin()
    if d > dist[u]: continue
    for (v, w) in adj[u]:
      if d + w < dist[v]:
        dist[v] = d + w
        pq.push((dist[v], v))
  return dist
```

Example

n = 5, s = 0



Pop (u, d)	Updates	dist after
start	push (0, 0)	[0, ∞, ∞, ∞, ∞]
(0, 0)	d1 = 4, d2 = 1	[0, 4, 1, ∞, ∞]
(2, 1)	d1 = 3, d3 = 6	[0, 3, 1, 6, ∞]
(1, 3)	d3 = 4	[0, 3, 1, 4, ∞]
(3, 4)	d4 = 7	[0, 3, 1, 4, 7]
(4, 7)	no change	[0, 3, 1, 4, 7]

Stale heap entries (4, 1) and (6, 3) are skipped.

dist = [0, 3, 1, 4, 7], time to reach all = 7

Complexity Analysis

1. Time Complexity

- Each vertex is finalised once. Each edge is relaxed at most once, so the heap gets at most E pushes.
- Each heap push or pop costs O(log E) = O(log V).
- Time Complexity = **O((V + E) log V)**
- (Using a plain array instead of a heap gives $O(V^2)$.)

2. Space Complexity

- dist[]: O(V). Heap: up to O(E) entries. Adjacency list: O(V + E).
- Space Complexity = **O(V + E)**

Shortest Paths in a Weighted DAG

Problem 3 | Medium

The Problem

- Given a **directed acyclic graph (DAG)** with n vertices, edge weights that may be **negative**, and a source vertex s, find the shortest distance from s to every vertex.
- Unreachable vertices have distance ∞.

Example:

n = 6, s = 0

Edges u → v (w): 0→1 (5), 0→2 (3), 1→2 (2), 1→3 (6), 2→3 (7), 2→4 (4), 2→5 (2), 3→4 (−1), 4→5 (−2)

Output: [0, 5, 3, 10, 7, 5]

Any ideas on how this can be solved?

Hints

- Dijkstra's greedy argument breaks with negative edges. Do we even need a priority queue when the graph has no cycles?
- Is there an ordering of the vertices in which every edge goes from an earlier vertex to a later one?
- If vertices are processed in that order, when we reach u, have all the edges coming into u already been relaxed?

The Solution

Key Idea

- In a topological order, every edge into u starts at a vertex placed before u.
- So when we reach u, dist[u] is already final. We relax each edge exactly once, and negative weights cause no trouble.

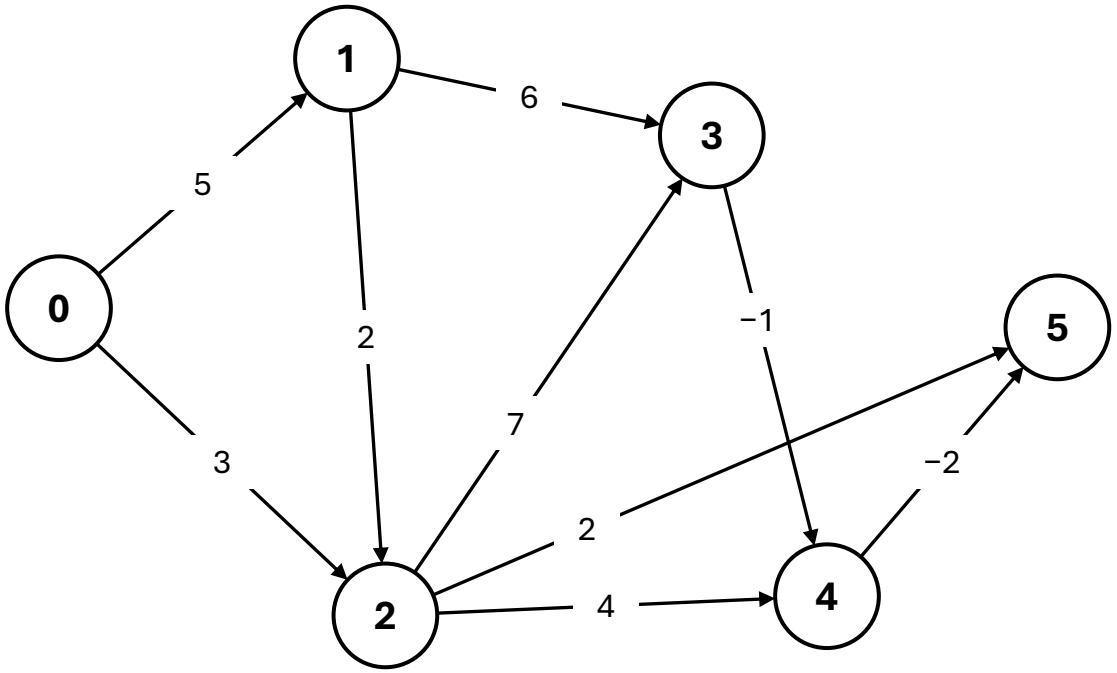
Steps

- Find a topological order (DFS or Kahn's algorithm).
- dist[all] = ∞, dist[s] = 0.
- For each u in that order (skip if dist[u] = ∞): relax every edge (u, v, w).

```
order = topologicalSort(G)
dist[all] = INF
dist[s] = 0
for u in order:
  if dist[u] == INF: continue
  for (v, w) in adj[u]:
    if dist[u] + w < dist[v]:
      dist[v] = dist[u] + w
return dist
```

Example

Topological order: 0, 1, 2, 3, 4, 5



Process	Updates	dist after
start	dist[0] = 0	[0, ∞, ∞, ∞, ∞, ∞]
0	d1 = 5, d2 = 3	[0, 5, 3, ∞, ∞, ∞]
1	d3 = 11 (5+2 > 3, no)	[0, 5, 3, 11, ∞, ∞]
2	d3 = 10, d4 = 7, d5 = 5	[0, 5, 3, 10, 7, 5]
3	10−1 = 9 > 7, no	[0, 5, 3, 10, 7, 5]
4	7−2 = 5, no gain	[0, 5, 3, 10, 7, 5]

dist = [0, 5, 3, 10, 7, 5]

Complexity Analysis

1. Time Complexity

- Topological sort takes O(V + E).
- Then every vertex is processed once and every edge is relaxed once: O(V + E).
- Time Complexity = **O(V + E)**
- This is faster than Dijkstra and works with negative weights.

2. Space Complexity

- dist[] and the topological order: O(V). DFS recursion stack: O(V).
- Space Complexity = **O(V)** extra (O(V + E) including the graph)

Negative Weights and Negative Cycles

Problem 4 | Medium

The Problem

- Given a directed graph with n vertices, edge weights that can be **negative**, and a source s, find the shortest distance from s to every vertex.
- If a **negative-weight cycle** is reachable from s, shortest distances do not exist: report "Negative cycle".

Example:

n = 5, s = 0

Edges u → v (w), in this order: 3→4 (3), 1→3 (−1), 2→1 (−2), 2→3 (4), 0→1 (6), 0→2 (5)

Output: [0, 3, 5, 2, 5]

Any ideas on how this can be solved?

Hints

- Why does Dijkstra fail when an edge has negative weight?
- A shortest path that contains no cycle can have at most how many edges?
- If we relax every edge once, which paths are guaranteed to be correct afterwards? What about after two rounds?
- If distances still improve after $V-1$ rounds, what does that tell you?

The Solution

Key Idea (Bellman-Ford)

- A shortest path without a cycle uses at most V−1 edges.
- After round i, every shortest path with at most i edges is correct. So V−1 rounds of relaxing all edges are enough.
- If a V-th round still improves some distance, a negative cycle is reachable from s.

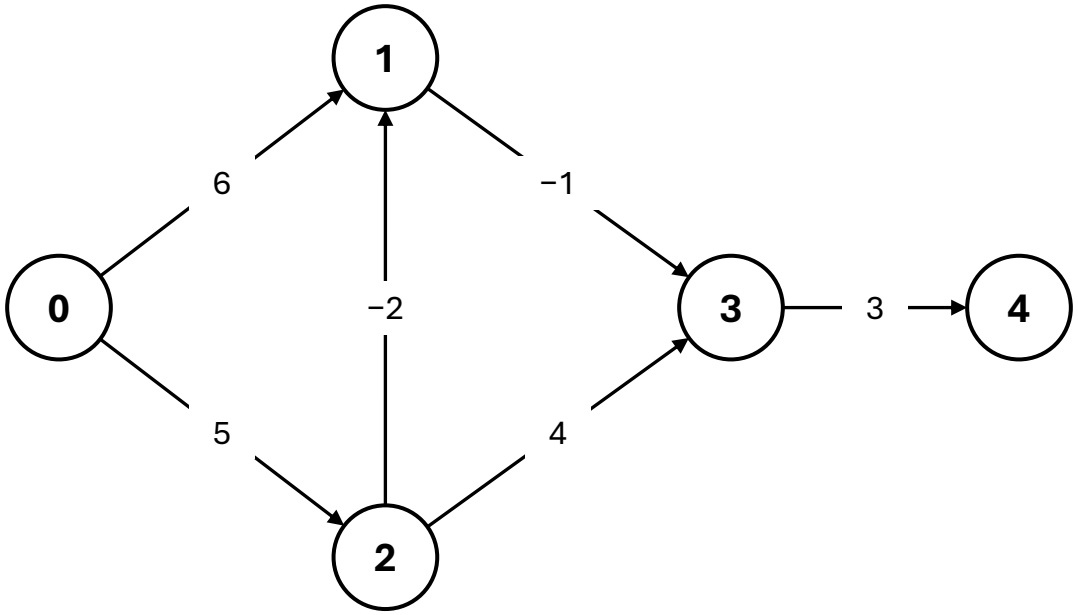
Steps

- dist[all] = ∞, dist[s] = 0.
- Repeat V−1 times: relax all edges (stop early if nothing changes).
- One more pass: if any edge still relaxes, report a negative cycle.

```
dist[all] = INF; dist[s] = 0
repeat V-1 times:
  for (u, v, w) in edges:
    if dist[u] != INF and
       dist[u] + w < dist[v]:
      dist[v] = dist[u] + w
for (u, v, w) in edges:
  if dist[u] != INF and
     dist[u] + w < dist[v]:
    return "Negative cycle"
return dist
```

Example

n = 5, s = 0. Edges are relaxed in the listed order, so V−1 = 4 passes are needed.



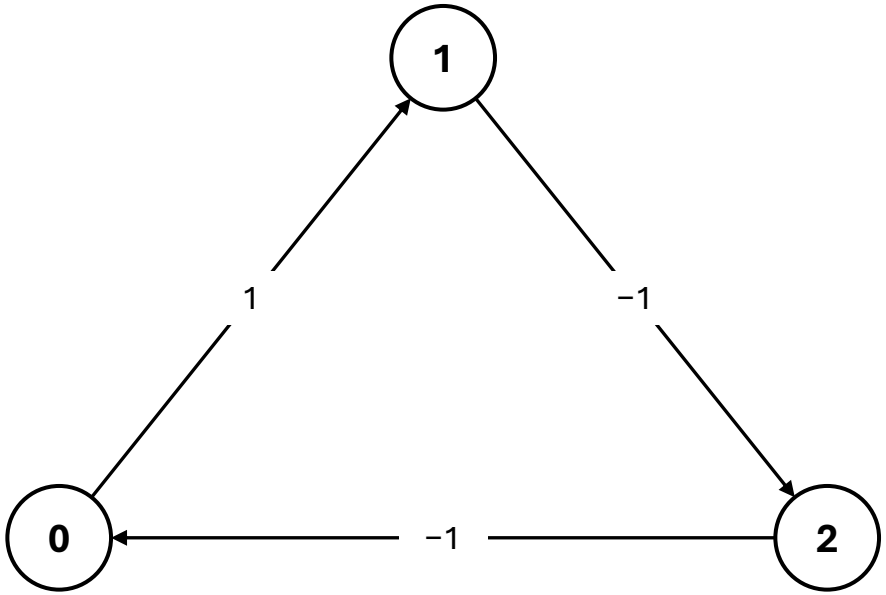
Pass	d[0]	d[1]	d[2]	d[3]	d[4]
start	0	∞	∞	∞	∞
1	0	6	5	∞	∞
2	0	3	5	5	∞
3	0	3	5	2	8
4	0	3	5	2	5
5 (check)	0	3	5	2	5

Pass 5 relaxes nothing, so there is no negative cycle.

dist = [0, 3, 5, 2, 5]

Example (contd.)

A graph with a negative cycle: 0→1 (1), 1→2 (−1), 2→0 (−1), s = 0



Pass	d[0]	d[1]	d[2]
start	0	∞	∞
1	−1	1	0
2	−2	0	−1
3 (check)	−3	−1	−2

- Cycle weight = 1 + (−1) + (−1) = −1.
- Each trip around the cycle lowers the distances by 1.
- V−1 = 2 passes are done, but pass 3 still relaxes edges.
- Hence the answer: "Negative cycle".

Complexity Analysis

1. Time Complexity

- Each pass relaxes all E edges, and there are V−1 passes, plus one extra pass for the check.
- Time Complexity = **O(V · E)**
- Slower than Dijkstra, but it handles negative weights and detects negative cycles.

2. Space Complexity

- Only the dist[] array is needed, besides the edge list.
- Space Complexity = **O(V)** extra

Cheapest Flights Within K Stops

Problem 5 | Medium - Hard

The Problem

- There are n cities connected by one-way flights. Each flight is (from, to, price).
- Given src, dst and an integer k, find the **cheapest price** from src to dst with **at most k stops** (intermediate cities). If no such route exists, return −1.

Example:

n = 4, src = 0, dst = 3, k = 1

Flights: (0,1,100), (1,2,100), (2,3,100), (0,2,500), (1,3,600)

Output: 600

The cheapest route overall is 0→1→2→3 = 300, but it has 2 stops, which is not allowed.

Any ideas on how this can be solved?

Hints

- What extra constraint does k add? Will plain Dijkstra, which always keeps the cheapest path, still be correct?
- k stops means a route uses at most how many edges?
- After i rounds of relaxing all edges, how many edges long are the paths you have considered? How do you make sure each round extends paths by exactly one edge?

The Solution

Key Idea

- At most k stops means at most **k + 1 edges**.
- Run Bellman-Ford for only k + 1 rounds.
- In each round, read from the previous round's distances and write into a copy. Otherwise a chain u→v→x could be relaxed within one round, using 2 edges in a single round.

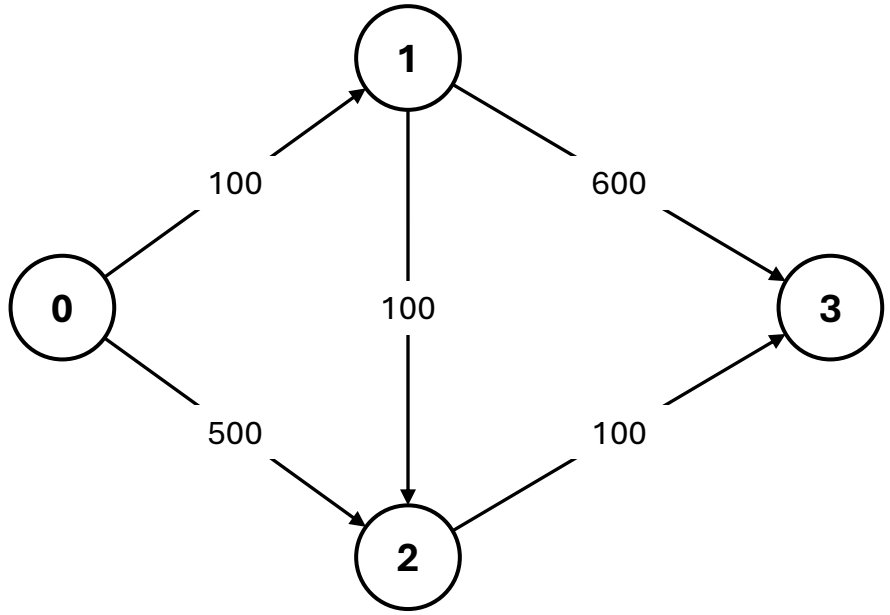
Steps

- dist[all] = ∞, dist[src] = 0.
- Repeat k + 1 times: temp = copy of dist; relax every flight using dist, store results in temp; dist = temp.
- Answer is dist[dst], or −1 if it is ∞.

```
dist[all] = INF
dist[src] = 0
repeat k + 1 times:
  temp = copy(dist)
  for (u, v, w) in flights:
    if dist[u] != INF and
       dist[u] + w < temp[v]:
      temp[v] = dist[u] + w
  dist = temp
if dist[dst] == INF: return -1
return dist[dst]
```

Example

src = 0, dst = 3, k = 1   (at most 2 edges, so 2 rounds)



Round	dist after the round
start	[0, ∞, ∞, ∞]
1 (1 edge, 0 stops)	[0, 100, 500, ∞]
2 (2 edges, 1 stop)	[0, 100, 200, 600]

- Round 2 uses the old dist[2] = 500 for the flight 2→3, giving 600.
- The cheaper dist[2] = 200 only becomes usable in round 3, which is not allowed (k = 1).

Answer = dist[3] = 600

Complexity Analysis

1. Time Complexity

- There are k + 1 rounds.
- Each round copies dist[] (O(V)) and relaxes all E flights (O(E)).
- Time Complexity = **O(k · (V + E))**
- In the worst case k = V−1, which matches Bellman-Ford: O(V · E).

2. Space Complexity

- Two arrays of size V (dist and temp).
- Space Complexity = **O(V)** extra

Shortest Path with One Discount Coupon

Problem 6 | Hard

The Problem

- Given a directed graph with n vertices and non-negative **even** edge weights, a source s and a destination t.
- You have **one coupon**: on at most one edge of your route, you can pay **half** of its weight.
- Find the minimum cost to travel from s to t.

Example:

n = 4, s = 0, t = 3

Edges u → v (w): 0→1 (6), 1→3 (6), 0→2 (2), 2→3 (12)

Output: 8

(Use the coupon on 2→3: cost 2 + 6 = 8.)

Any ideas on how this can be solved?

Hints

- Is knowing only the current vertex enough to decide how good a partial route is? What else matters?
- Can you make "coupon used or not" a part of the state, so that each vertex splits into two states?
- Try "find the shortest path, then halve its largest edge" on the example. Does it always work?

The Solution

Key Idea

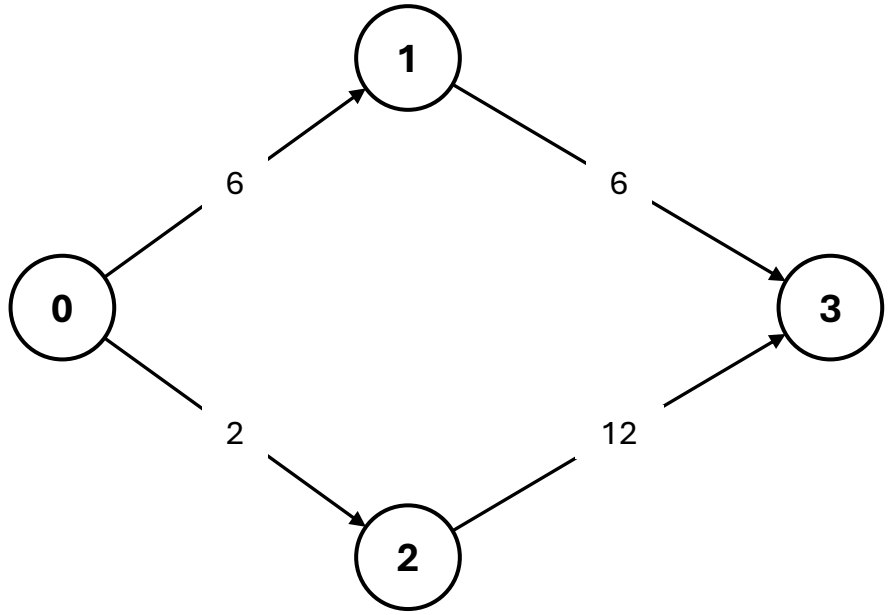
- State = (v, c), where c = 0 if the coupon is unused and c = 1 if it is used. This gives 2V states.
- Run Dijkstra on this state graph. From (u, c) along edge (u, v, w):
- normal: go to (v, c) with cost w
- coupon (only if c = 0): go to (v, 1) with cost w / 2
- Answer = min(dist[t][0], dist[t][1]), since using the coupon is optional.

```
dist[v][c] = INF for all v, c
dist[s][0] = 0
pq.push((0, s, 0))
while pq not empty:
  (d, u, c) = pq.popMin()
  if d > dist[u][c]: continue
  for (v, w) in adj[u]:
    relax(v, c, d + w)
    if c == 0:
      relax(v, 1, d + w / 2)
return min(dist[t][0], dist[t][1])

relax(v, c, nd):
  if nd < dist[v][c]:
    dist[v][c] = nd
    pq.push((nd, v, c))
```

Example

s = 0, t = 3. State (v, c): c = 1 means the coupon is already used.



Pop (v, c) : dist	Updates
(0, 0) : 0	(1,0)=6, (2,0)=2, (1,1)=3, (2,1)=1
(2, 1) : 1	(3,1)=13
(2, 0) : 2	(3,0)=14, (3,1)=8
(1, 1) : 3	3 + 6 = 9 > 8, no change
(1, 0) : 6	(3,0)=12; 6 + 3 = 9 > 8, no change
(3, 1) : 8	destination reached

Answer = 8

Without the coupon the best route is 0→1→3 = 12; halving its largest edge gives 9, not 8.

Complexity Analysis

1. Time Complexity

- There are 2V states. Each edge gives at most 3 transitions: 2 from (u, 0) and 1 from (u, 1).
- So there are O(V) states and O(E) transitions, each handled with a heap operation costing O(log V).
- Time Complexity = **O((V + E) log V)**

2. Space Complexity

- dist table with 2V entries; heap with up to 3E entries.
- Space Complexity = **O(V + E)**