CyberApocalypse 2026 — Coding — Rumour Spine

Category: Coding **Challenge:** Rumour Spine **Endpoint:** `http://154.57.164.76:31482` **Flag:** `HTB{0n3_c00rd1n4t3d_m0m3nt}`

---

1. Challenge

> *Suncourt doesn't march first — it primes first. Miren Vale has traced the Quiet March's invitation chain: a coordinator who never touches ash, only schedules, feeding a priming signal through a network of quiet hands — candle-buyers, shrine-attendants, penitent-scribes — until a target district falls still the night before the March arrives. Every hand who passes the signal on is a link Miren can expose.*
>
> *Miren means to break the chain before the district goes quiet. She can expose any quiet hand along the way — one flagged, one route burned, at the cost of one favour she can never call in again. The coordinator and the target district can't be touched directly: too visible, too soon. What is the minimum number of quiet hands that must be exposed to sever every path the priming signal could still travel?*

I/O specification

Input:

- Line 1: four space-separated integers `N E S T`
- `N` — number of quiet hands (nodes, labelled `0 .. N-1`)
- `E` — number of directed links (edges)
- `S` — coordinator node
- `T` — target district node

- Next `E` lines: two integers `u v` — a directed link along which the signal passes from hand `u` to hand `v`.

Output: a single integer — the minimum number of hands (**nodes other than `S` and `T`**) that must be exposed to disconnect all paths from `S` to `T`.

Provided example

```
Input:
7 8 0 6
0 1
0 2
1 3
2 4
3 6
4 6
2 3
1 4

Expected output:
2
```

The four `0 → 6` paths are `0→1→3→6`, `0→1→4→6`, `0→2→3→6`, `0→2→4→6`. No single hand lies on all four, but removing both `1` and `2` kills every path — hence `2`.

---

2. Recognising the problem

The narrative dressing hides a textbook graph problem. Stripped down:

> *Given a directed graph, a source `S` and a sink `T`, find the **minimum number of vertices other than `S` and `T`** whose removal leaves no directed `S → T` path.*

That is the **minimum S–T vertex cut** (also called the minimum internally-disjoint separator).

The key theoretical hook is **Menger's theorem** — the vertex-connectivity version:

> *The minimum number of internal vertices separating `S` from `T` equals the maximum number of **internally vertex-disjoint** `S → T` paths.*

And the maximum number of internally vertex-disjoint paths is computable as a **max-flow** value, provided we can express "each vertex may be used at most once" as a capacity constraint. Edges carry capacities in flow networks, not vertices — so we need a transformation.

Note the important distinction: this is a **vertex** cut, not an **edge** cut. A naive `S→T` edge max-flow gives the wrong answer. On the sample, the minimum *edge* cut happens to be 2 as well, so the sample alone does not distinguish the two readings — a trap worth noticing, because hidden tests certainly do distinguish them.

A graph that separates them, with `S=0, T=6`:

```
0→1  0→2  1→3  2→3  3→4  3→5  4→6  5→6
```

Every path funnels through node `3`, so the minimum **vertex** cut is `1` (expose hand `3`). But no single *edge* cut works — the narrowest edge cuts are pairs like `{0→1, 0→2}`, `{1→3, 2→3}`, `{3→4, 3→5}` or `{4→6, 5→6}`, so the minimum **edge** cut is `2`. Both values were confirmed with an independent Edmonds–Karp edge max-flow. Solving the edge version here would answer `2` where the correct answer is `1`.

3. The reduction: node splitting

The standard trick for converting vertex capacities into edge capacities is **node splitting** (vertex splitting / node bisection).

Replace every original node `v` by two nodes:

- `v_in` = `2v`
- `v_out` = `2v + 1`

and add edges:

Edge	Capacity	Meaning
`v_in → v_out`		

Edge	Capacity	Meaning
	`1` for internal `v`	hand `v` can carry the signal once; cutting this edge = exposing hand `v`
`v_in → v_out`	`∞` for `v ∈ {S, T}`	the coordinator and district cannot be exposed
`u_out → v_in` for each original edge `u → v`	`∞`	links themselves are not the resource being spent

Then run max-flow from `S_out` to `T_in`.

Why this is exactly right:

- **Every path through the split graph must traverse some `v_in → v_out` edge for each original node it visits.** So a unit of flow saturating capacity `1` at each internal node means each internal hand is used by at most one flow path — precisely internal vertex-disjointness.
- **Only the internal `v_in → v_out` edges have finite capacity.** Therefore every minimum cut consists solely of those edges, and each such edge corresponds one-to-one with "expose hand `v`". A min cut of size `k` is literally a set of `k` hands to expose.
- **`S` and `T` get `∞` capacity**, encoding "can't be touched directly", so no min cut ever selects them.
- By max-flow/min-cut, `maxflow(S_out → T_in) =` min cut of split graph `=` min number of internal hands to expose. Combined with Menger, this is also the number of internally disjoint paths.

Choosing the source as `S_out` and the sink as `T_in` is deliberate: it skips `S`'s and `T`'s own splitting edges, so their `∞` capacity never even matters for correctness — but keeping them `∞` makes the construction uniform and safe.

Edge cases

- **A direct edge `S → T`.** Then no set of *internal* hands can ever sever the chain — the flow is unbounded, since that path traverses no finite-capacity edge. The task is impossible; the solution detects this and prints `-1`.
- **`S == T`.** Degenerate/impossible for the same reason; also `-1`.
- **`T` unreachable from `S`.** Max-flow is `0`, which is the correct answer — zero hands need exposing.

Since every genuine `S → T` path passes through at least one internal node with capacity `1`, every augmenting path has bottleneck exactly `1`, so the flow value is always an integer bounded by `N - 2`.

4. Algorithm and complexity

Dinic's algorithm on the split graph. Because all bottleneck capacities are unit, Dinic behaves like the unit-capacity case: each augmentation pushes exactly `1`, and the number of phases is `O(√V)`, giving roughly `O(E√V)` — comfortably fast, and in practice the value of the flow is small (bounded by `N - 2` and typically by the graph's narrowest layer).

Implementation notes:

- Adjacency stored as flat `to` / `cap` arrays with per-node index lists; the reverse edge of edge `i` is `i ^ 1` (edges added in pairs).
- `itp[]` is the current-arc / iterator optimisation — without it, blocking-flow construction degrades badly.
- `level[]` from BFS restricts DFS to the level graph.
- The DFS is recursive, so the solution raises the recursion limit and runs `main` on a thread with a 64 MB stack. This avoids a `RecursionError` on long path chains in large hidden tests.

5. Solution

```python
import sys, threading
from collections import deque

def main():
    data = sys.stdin.buffer.read().split()
    if not data:
        return
    n = int(data[0]); e = int(data[1]); s = int(data[2]); t = int(data[3])
    pos = 4

    # split every node: v_in = 2*v, v_out = 2*v+1
    N = 2 * n
    INF = 1 << 60
    to = []
    cap = []
    head = [[] for _ in range(N)]

    def add(u, v, c):
        head[u].append(len(to)); to.append(v); cap.append(c)
        head[v].append(len(to)); to.append(u); cap.append(0)

    direct = False
    for _ in range(e):
        u = int(data[pos]); v = int(data[pos + 1]); pos += 2
        if u == s and v == t:
            direct = True
        add(2 * u + 1, 2 * v, INF)

    for v in range(n):
        # coordinator S and district T cannot be exposed → infinite capacity
        add(2 * v, 2 * v + 1, INF if (v == s or v == t) else 1)

    if s == t or direct:
        print(-1)
        return

    src, snk = 2 * s + 1, 2 * t
    level = [-1] * N
    itp = [0] * N

    def bfs():
        for i in range(N):
            level[i] = -1
        level[src] = 0
        q = deque([src])
        while q:
            u = q.popleft()
            for i in head[u]:
```

```
                if cap[i] > 0 and level[to[i]] < 0:
                    level[to[i]] = level[u] + 1
                    q.append(to[i])
        return level[snk] >= 0

    def dfs(u, f):
        if u == snk:
            return f
        h = head[u]
        while itp[u] < len(h):
            i = h[itp[u]]
            v = to[i]
            if cap[i] > 0 and level[v] == level[u] + 1:
                d = dfs(v, f if f < cap[i] else cap[i])
                if d > 0:
                    cap[i] -= d
                    cap[i ^ 1] += d
                    return d
            itp[u] += 1
        return 0

    flow = 0
    while bfs():
        for i in range(N):
            itp[i] = 0
        while True:
            f = dfs(src, INF)
            if f == 0:
                break
            flow += f
    print(flow)

sys.setrecursionlimit(1 << 20)
threading.stack_size(1 << 26)
th = threading.Thread(target=main)
th.start()
th.join()
```

6. Verification before submitting

Two checks were run locally rather than burning submission attempts on guesses.

Sample:

```
$ printf '7 8 0 6\n0 1\n0 2\n1 3\n2 4\n3 6\n4 6\n2 3\n1 4\n' | python3 sol.py
2
```

Randomised differential test against brute force. 400 random small graphs (`3 ≤ N ≤ 7`, up to 10 edges), with the answer independently computed by enumerating *every* subset of internal nodes in increasing size order and BFS-checking reachability:

```
import itertools, random, subprocess
from collections import deque

def brute(n, edges, s, t):
    inner = [v for v in range(n) if v != s and v != t]
    def reach(rem):
        seen = {s}; q = deque([s]); adj = {}
        for u, v in edges:
            if u in rem or v in rem: continue
            adj.setdefault(u, []).append(v)
        while q:
            u = q.popleft()
            if u == t: return True
            for v in adj.get(u, []):
                if v not in seen:
                    seen.add(v); q.append(v)
        return t in seen
    for k in range(len(inner) + 1):
        for c in itertools.combinations(inner, k):
            if not reach(set(c)): return k
    return -1

random.seed(1)
bad = 0
for trial in range(400):
    n = random.randint(3, 7)
    s, t = random.sample(range(n), 2)
    poss = [(u, v) for u in range(n) for v in range(n)
            if u != v and not (u == s and v == t) and v != s and u != t]
    edges = random.sample(poss, random.randint(0, min(len(poss), 10)))
    inp = f"{n} {len(edges)} {s} {t}\n" + "".join(f"{u} {v}\n" for u, v in edges)
    got = subprocess.run(['python3', 'sol.py'], input=inp,
                         capture_output=True, text=True).stdout.strip()
    exp = str(brute(n, edges, s, t))
    if got != exp:
        bad += 1; print("MISMATCH", inp, "got", got, "exp", exp)
print("done, mismatches:", bad)
```

Result: `done, mismatches: 0` — 400/400 agreement.

A performance smoke test (`N = 20000`, a wide layered graph with ~2400 edges and an answer of 200) finished in 0.03 s, so pure-Python Dinic is not a bottleneck here.

7. Submission

The web front end posts the editor contents to `/run` as JSON. Submitting directly:

```
import json, urllib.request
code = open('sol.py').read()
req = urllib.request.Request(
    'http://154.57.164.76:31482/run',
    data=json.dumps({'code': code, 'language': 'python'}).encode(),
    headers={'Content-Type': 'application/json'})
print(urllib.request.urlopen(req, timeout=120).read().decode())
```

Response:

```
{"challengeCompleted":true,"flag":"HTB{0n3_c00rd1n4t3d_m0m3nt}","result":{}}
```

8. Flag

```
HTB{0n3_c00rd1n4t3d_m0m3nt}
```

Takeaways

- **"Minimum things to remove to disconnect A from B" is min-cut.** If the things are *edges*, it is max-flow directly; if they are *vertices*, split nodes first.
- **Node splitting is the general recipe for vertex capacities**, and it converts a vertex cut into an edge cut whose members map one-to-one onto vertices.
- **Protected endpoints** (`S`, `T` untouchable) are expressed by giving their split edges infinite capacity — no extra logic needed.

- **Verify with brute force on small random inputs.** The sample here satisfies both the edge-cut and vertex-cut readings, so passing the sample proves very little; the differential test is what gave real confidence.