Red Scare!

Algorithms and Data Structures Report

Costel Gutu, Vladislav Konjushenko, Julia Trznadel, Hana Dubovská, Oleksandr Adamov

November 26, 2025

1 Algorithms

We consider five problems defined on an input graph $G = (V, E)$ with special vertices s, t, and a distinguished subset $R \subseteq V$ ("red" vertices). Each input instance consists of a mix of directed and undirected edges. Two of the problems are NP-hard in general and therefore only partial solutions are provided: Many (NP-hard in general) and Alternate (we only solve undirected graphs; ?! otherwise)." For each problem, we implemented an algorithm consistent with the assignment specification:

- **A (Alternate):** Determine whether there exists an s–t path whose consecutive vertices alternate between red and non-red. Solve only on undirected graphs; if any directed edge exists we return "?!". On undirected graphs we run BFS from s over edges where exactly one endpoint is red, with a visited set to enforce simple paths; this finds an alternating s–t path if it exists.
- **F (Few):** Compute the minimum number of red vertices on any s–t path. Implemented using Dijkstra's algorithm where each vertex contributes cost 0 or 1.
- **M (Many):** Compute the maximum number of red vertices on an s–t path. In general this problem is NP-hard. Our solver returns "?!" for instances with undirected edges or directed cycles. For the remaining DAGs, dynamic programming over a topological order gives the correct value.
- **N (None):** Compute the length of the shortest s–t path avoiding all *internal* red vertices (i.e. s and t may be red, intermediates may not). Implemented using BFS on a restricted state space.
- **S (Some):** Determine whether there exists an s–t path containing at least one red vertex. Implemented via forward reachability from s and backward reachability to t on the reverse graph.

All algorithms were implemented in Python 3.13.

2 Results

We executed all algorithms over the dataset, but included in the table *only the instances with at least 500 vertices*, matching the requirement in the assignment description.

The table below reports, for each instance:

- the number of vertices n,
- **A**: whether an alternating s–t path exists; '?!' if the instance has any directed edge.
- **F**: minimum number of red vertices on any s–t path,
- **M**: maximum number of red vertices on an s–t path ("?!" for unsupported),
- **N**: length of the shortest red-avoiding s–t path,
- **S**: whether some s–t path uses at least one red vertex.

Both A and M can be '?!' when the instance is outside the supported class.

Instance	**n**	**A**	**F**	**M**	**N**	**S**
bht.txt	5757	false	0	?!	6	true
common-1-1000.txt	1000	false	-1	?!	-1	false
common-1-1500.txt	1500	false	-1	?!	-1	false
common-1-2000.txt	2000	false	-1	?!	-1	false
common-1-2500.txt	2500	false	1	?!	6	true
common-1-3000.txt	3000	false	1	?!	6	true
common-1-3500.txt	3500	false	1	?!	6	true
common-1-4000.txt	4000	false	1	?!	6	true
common-1-4500.txt	4500	true	1	?!	6	true
common-1-500.txt	500	false	-1	?!	-1	false
common-1-5000.txt	5000	true	1	?!	6	true
common-1-5757.txt	5757	true	1	?!	6	true
common-2-1000.txt	1000	true	1	?!	4	true
common-2-1500.txt	1500	true	1	?!	4	true
common-2-2000.txt	2000	true	1	?!	4	true
common-2-2500.txt	2500	true	1	?!	4	true
common-2-3000.txt	3000	true	1	?!	4	true
common-2-3500.txt	3500	true	1	?!	4	true
common-2-4000.txt	4000	true	1	?!	4	true
common-2-4500.txt	4500	true	1	?!	4	true
common-2-500.txt	500	true	1	?!	4	true
common-2-5000.txt	5000	true	1	?!	4	true
common-2-5757.txt	5757	true	1	?!	4	true
gnm-1000-1500-0.txt	1000	false	1	?!	-1	true

Instance	**n**	**A**	**F**	**M**	**N**	**S**
gnm-1000-1500-1.txt	1000	false	2	?!	-1	true
gnm-1000-2000-0.txt	1000	false	0	?!	7	true
gnm-1000-2000-1.txt	1000	false	2	?!	-1	true
gnm-2000-3000-0.txt	2000	false	0	?!	8	true
gnm-2000-3000-1.txt	2000	true	2	?!	-1	true
gnm-2000-4000-0.txt	2000	false	0	?!	6	true
gnm-2000-4000-1.txt	2000	false	0	?!	5	true
gnm-3000-4500-0.txt	3000	false	0	?!	10	true
gnm-3000-4500-1.txt	3000	false	2	?!	-1	true
gnm-3000-6000-0.txt	3000	false	0	?!	6	true
gnm-3000-6000-1.txt	3000	false	2	?!	6	true
gnm-4000-6000-0.txt	4000	false	0	?!	7	true
gnm-4000-6000-1.txt	4000	false	1	?!	15	true
gnm-4000-8000-0.txt	4000	false	0	?!	5	true
gnm-4000-8000-1.txt	4000	true	2	?!	6	true
gnm-5000-10000-0.txt	5000	false	2	?!	5	true
gnm-5000-10000-1.txt	5000	true	1	?!	5	true
gnm-5000-7500-0.txt	5000	false	-1	?!	-1	false
gnm-5000-7500-1.txt	5000	false	-1	?!	-1	false
grid-25-0.txt	625	true	0	?!	324	true
grid-25-1.txt	625	true	0	?!	123	true
grid-25-2.txt	625	true	5	?!	-1	true
grid-50-0.txt	2500	false	0	?!	1249	true
grid-50-1.txt	2500	false	0	?!	521	true
grid-50-2.txt	2500	false	11	?!	-1	true
increase-n500-1.txt	500	?!	2	16	1	true
increase-n500-2.txt	500	?!	1	17	1	true
increase-n500-3.txt	500	?!	1	16	1	true
rusty-1-2000.txt	2000	false	-1	?!	-1	false
rusty-1-2500.txt	2500	false	-1	?!	-1	false
rusty-1-3000.txt	3000	false	0	?!	14	true
rusty-1-3500.txt	3500	false	0	?!	14	true
rusty-1-4000.txt	4000	false	0	?!	13	true
rusty-1-4500.txt	4500	false	0	?!	7	true
rusty-1-5000.txt	5000	false	0	?!	7	true
rusty-1-5757.txt	5757	false	0	?!	7	true
rusty-2-2000.txt	2000	false	0	?!	5	true
rusty-2-2500.txt	2500	false	0	?!	4	true
rusty-2-3000.txt	3000	false	0	?!	4	true
rusty-2-3500.txt	3500	false	0	?!	4	true

Instance	n	A	F	M	N	S
rusty-2-4000.txt	4000	false	0	?!	4	true
rusty-2-4500.txt	4500	false	0	?!	4	true
rusty-2-5000.txt	5000	false	0	?!	4	true
rusty-2-5757.txt	5757	false	0	?!	4	true
smallworld-30-0.txt	900	false	0	?!	9	true
smallworld-30-1.txt	900	true	1	?!	11	true
smallworld-40-0.txt	1600	false	0	?!	8	true
smallworld-40-1.txt	1600	true	1	?!	13	true
smallworld-50-0.txt	2500	false	0	?!	3	true
smallworld-50-1.txt	2500	true	2	?!	-1	true
wall-n-100.txt	800	false	0	?!	1	true
wall-n-1000.txt	8000	false	0	?!	1	true
wall-n-10000.txt	80000	false	0	?!	1	true
wall-p-100.txt	602	false	0	?!	1	true
wall-p-1000.txt	6002	false	0	?!	1	true
wall-p-10000.txt	60002	false	0	?!	1	true
wall-z-100.txt	701	false	0	?!	1	true
wall-z-1000.txt	7001	false	0	?!	1	true
wall-z-10000.txt	70001	false	0	?!	1	true

3 Methods

3.1 Problem N: None

Problem. Given a graph $G = (V, E)$ with a distinguished set of red vertices $R \subseteq V$ and two vertices $s, t \in V$, the problem None asks for the length of a shortest path from s to t whose internal vertices are all non-red. The endpoints s and t are allowed to be red. If no such path exists, the answer is -1.

Algorithm. We first mark all red vertices as blocked, except possibly s and t:

$$B := R \setminus \{s, t\}.$$

We then run a standard breadth–first search from s on the graph obtained by deleting all vertices in B. More concretely, BFS maintains a queue of vertices together with their distance from s. When exploring a vertex v, the algorithm only considers neighbors $u \notin B$. The search stops when t is dequeued, at which point the stored distance is the length of a shortest s–t path with no blocked internal vertex. If t is never reached, the answer is -1.

Running time. Each vertex and edge is processed at most once by BFS. Deleting the blocked vertices conceptually can be implemented by simply skipping them during exploration. Thus the running time is $O(n + m)$.

3.2 Problem S: Some

Problem. The problem SOME asks whether there exists an s–t path that visits at least one red vertex.

Algorithm. We use two reachability computations:

- First, we perform BFS from s in the original adjacency structure and obtain the set Reach_s of vertices reachable from s.
- Second, we build the reverse adjacency lists. For every edge $u \to v$ we store u as a neighbor of v, and perform BFS from t in this reverse graph, obtaining the set Reach_t^* of vertices that can reach t in the original graph.

Then there exists an s–t path that passes through a red vertex r if and only if $r \in R \cap \mathrm{Reach}_s \cap \mathrm{Reach}_t^*$. The algorithm simply checks whether this intersection is non-empty and returns `true` if it is, and `false` otherwise.

Running time. Each BFS takes $O(n + m)$ time. The final scan over all red vertices is $O(n)$. Thus the total running time is $O(n + m)$.

3.3 Problem F: Few

Problem. The problem FEW asks for the minimum number of red vertices on any s–t path. If there is no s–t path at all, the answer is -1.

Algorithm. This problem can be seen as a shortest-path problem with non-negative *vertex* costs. We assign a cost of 1 to every red vertex and 0 to every non-red vertex. The cost of a path is the sum of the costs of all vertices on the path. In particular, if s is red then the path cost includes 1 for s.

We then run Dijkstra's algorithm on this vertex-weighted graph. In the implementation, the distance array $dist[v]$ stores the minimum cost of any known path from s to v. We initialize

$$dist[v] := \infty \quad \text{for all } v \in V,$$

and set $dist[s] := 1$ if $s \in R$ and 0 otherwise. When relaxing an edge (v, u), the tentative distance to u is

$$dist[v] + \big(1 \text{ if } u \in R \text{ else } 0\big).$$

Dijkstra's algorithm continues until the priority queue is empty. If $dist[t] < \infty$ at the end, this value is the answer; otherwise we return -1.

Running time. All edge relaxations have non-negative cost and we use a binary heap. Therefore the running time of Dijkstra's algorithm is $O((n + m) \log n)$.

3.4 Problem A: Alternate

Problem. The problem ALTERNATE asks whether there exists an s–t path whose consecutive vertices alternate between red and non-red. For every edge (v, u) on the path exactly one of v and u is red. We only solve this on undirected graphs; if any directed edge exists, we return '?!'.

Algorithm. We perform BFS from s, traversing an edge (v,u) only if exactly one of v,u is red. We keep a visited set, so each vertex is enqueued at most once. If t is reached we return true; otherwise false.

Running time. Each vertex is enqueued at most once, and each edge is inspected at most twice as once from each endpoint, with a constant-time check of the color condition. Thus the running time is $O(n + m)$.

3.5 Problem M: Many on DAGs

Problem. The problem MANY asks for the maximum number of red vertices that can appear on any s–t path. If there is no s–t path at all, the answer is -1. On general graphs this problem is NP-hard (see Section 4). Therefore we restrict our implementation to a well-defined class of input graphs: directed acyclic graphs with no undirected edges.

Algorithm. Given an input graph G, we first check whether it fits this restricted class:

1. If the instance contains any undirected edge (e.g. an edge specified as "$u - v$"), we do not attempt to solve MANY and instead mark the result as `?!` in the output table.
2. Otherwise we treat the given adjacency lists as a directed graph and perform a standard depth–first search to detect directed cycles. If a cycle is found, we again return `?!`.
3. If the graph is a directed acyclic graph, we compute a topological order of its vertices, for example by Kahn's algorithm.

On a DAG we solve MANY by dynamic programming in topological order. For each vertex v we maintain a value

$$dp[v] = \text{maximum number of red vertices on any path from } s \text{ to } v,$$

with the convention that $dp[v] = -\infty$ if v is not reachable from s. We initialize

$$dp[v] := -\infty \quad \text{for all } v \in V, \qquad dp[s] := \begin{cases} 1 & \text{if } s \in R, \\ 0 & \text{otherwise.} \end{cases}$$

Then we process the vertices in topological order. For each vertex v with $dp[v] > -\infty$ and each outgoing edge (v, u) we update

$$dp[u] := \max\big(dp[u],\ dp[v] + \big(1 \text{ if } u \in R \text{ else } 0\big)\big)\,.$$

After all vertices have been processed, if $dp[t] = -\infty$ there is no path from s to t and we return -1; otherwise we return $dp[t]$.

Running time. On graphs that pass the initial checks with no undirected edges and no directed cycles, computing the topological order and performing the dynamic program both take $O(n+m)$ time. The cycle detection via depth–first search is also $O(n + m)$. Thus the total running time for MANY on this restricted class is $O(n + m)$.

4 NP-hardness of Many

In this section we show that the decision version of Many is NP-hard on general graphs.

4.1 Problem definition

Consider the following decision version of Many:

Decision-MANY
Instance: A graph $G = (V, E)$, a set of red vertices $R \subseteq V$, two vertices $s, t \in V$, and an integer $k \geq 0$.
Question: Does there exist a simple s–t path in G that visits at least k red vertices?

Clearly, Decision-MANY is in NP: a certificate is a simple s–t path, and we can verify in polynomial time that it is simple and that it visits at least k red vertices.

4.2 Reduction from Hamiltonian s–t Path

Source problem. We reduce from the following known NP-complete problem:

Hamiltonian s–t Path
Instance: A graph $G = (V, E)$ and two distinct vertices $s, t \in V$.
Question: Does there exist a simple path from s to t that visits every vertex of G exactly once?

The Hamiltonian s–t Path problem is NP-complete even for undirected graphs.

Claim. Decision-MANY is NP-hard.

Proof. We will reduce from Hamiltonian s–t Path, which is known to be NP-complete. Let $G = (V, E)$ with distinguished vertices s, t be an instance of Hamiltonian s–t Path. Let $n = |V|$. From this instance we construct an instance of Decision-MANY as follows.

Construct a graph $G' = (V', E')$ by simply taking $G' = G$ (same vertices and edges). Set the set of red vertices to be all vertices:

$$R' := V'.$$

Keep the same distinguished vertices s', t':

$$s' := s, \qquad t' := t,$$

and set the threshold

$$k := n.$$

Thus the resulting Decision-MANY instance is (G', R', s', t', k).

The construction clearly runs in time polynomial in the size of the input graph, and the size of the constructed instance is $O(n + m)$.

We now prove correctness of the reduction by showing that G has a Hamiltonian s–t path if and only if the constructed Decision-MANY instance has an s'–t' path that visits at least k red vertices.

Suppose G has a Hamiltonian s–t path P. By definition, P is a simple path from s to t that visits every vertex of V exactly once. Since $R' = V'$, every vertex on P is red. The path P therefore visits exactly n red vertices. Because we set $k = n$, P is a valid witness for the Decision-MANY instance: it is a simple s'–t' path that visits at least k red vertices. Thus the Decision-MANY instance is a YES-instance.

Conversely, suppose the constructed Decision-MANY instance (G', R', s', t', k) is a YES-instance. Then there exists a simple s'–t' path P' in G' that visits at least k red vertices. Recall that $R' = V'$ and $k = n$. Because P' is simple, it cannot visit more than n distinct vertices. But P' visits at least $k = n$ red vertices, which implies that P' visits *exactly* n distinct vertices, that is, all vertices of V'. Thus P' is a simple path from s to t that visits every vertex of G exactly once. Therefore P' is a Hamiltonian s–t path in the original instance.

This shows that the Hamiltonian s–t Path instance is a YES-instance if and only if the constructed Decision-MANY instance is a YES-instance. Since Hamiltonian s–t Path is NP-complete, this proves that Decision-MANY is NP-hard. Combined with the observation that Decision-MANY is in NP, we conclude that Decision-MANY is NP-complete.

5 Resources

All code used in this report was written in Python 3.13.3. Key files:

- `src/red_scare.py` — all algorithms and the parser
- `src/run_all.py` — batch evaluation over all dataset files
- `results.txt` — raw results
- `results_filtered.txt` — results used in this report