Red Scare! Report

by Alice Cooper.

Results

The following table gives my results for all graphs of at least 500 vertices.

Instance name	n	A	F	M	N	S
rusty-5762	5,762	true	16	–	?	5
wall-p-10000	10,000					
⋮						

The columns are for the problems Alternate, Few, Many, None, and Some. The table entries either give the answer, or contain '?' for those cases where I was unable to find a solution within reasonable time. For those questions where there is a reason for my inability to find a good algorithm (because the problem is hard), I wrote '?!'.

For the complete table of all results, see the tab-separated text file `results.txt`.

Methods

For problem A, I solved each instance G by $\cdots$[1] The running time of this algorithm is $\cdot$, and my implementation spends $\cdots$ seconds on the instance $\cdots$ with $n = \cdots$.

I solved problem $\cdots$ for all $\cdots$[2] graphs using $\cdots$.

I was unable to solve problem $\cdots$ except for the $\cdots$ instances. This is because, in generality, this problem is $\cdots$. To see this, consider the following reduction from $\cdots$. Let ...

I was also unable to solve $\cdots$ for $\cdots$, but I don't know why.[3]

——————————WIP——————————

[1] Describe what you did. Use words like "building a inverse anti-tree without self-loops where each vertex in G is presented by a Strogatz–Wasserman shtump. I then performed a standard longest hash sorting using the algorithm of Bronf (Algorithm 5 in [1])." Be neat, brief, and precise.

[2] For instance, "planar, bipartite"

[3] Remove or expand as necessary.

Common notation and preprocessing

We denote the input graph by $G = (V, E)$ with $|V| = n$ and $|E| = m$. The graph may be directed or undirected; algorithms below handle both variants by respecting edge orientations. Let $s, t \in V$ be the source and sink vertices and $R \subseteq V$ be the set of *red* vertices. Unless stated otherwise, a path is a simple path (no repeated vertices). Complexity results assume the usual unit-cost RAM model.

For clarity we sometimes construct derived graphs G' (subgraphs or vertex-split graphs) and we report run time in terms of n and m.

Problem: NONE — shortest s-t path whose internal vertices avoid R

Algorithm. Create $G' = (V', E')$ from G by removing every red vertex $v \in R$ except keep s and t even if they are red:

$$V' = V \setminus (R \setminus \{s, t\}), \quad E' = E \cap (V' \times V').$$

Run BFS (breadth-first search) from s in G' to compute the distance $d(s, t)$ (unweighted edges). If t is unreachable return 'no path'.

Correctness. Any simple s-t path that has no internal red vertices is exactly an s-t path in G'. BFS computes shortest path length in unweighted graphs, so it returns the desired answer.

Complexity. Removing vertices is $O(n + m)$ (linear-time filtering). BFS runs in $O(n + m)$. Overall $O(n + m)$.

Problem: SOME — does an s-t path exist that contains at least one vertex in R?

Algorithm (preferred, simple reachability pair). 1. Run BFS (or DFS) from s in G to compute Reach_s, the set of vertices reachable from s. 2. Run BFS on the reverse graph G^{rev} from t to compute ReachTo_t, the set of vertices that can reach t. 3. If there exists $r \in R$ such that $r \in \text{Reach}_s \cap \text{ReachTo}_t$, then there exists a path srt and thus an s-t path containing a red vertex; otherwise no such path exists.

Correctness. If a simple s-t path contains some red vertex r, then necessarily $s \to r$ and $r \to t$. Conversely, if $s \to r$ and $r \to t$ then concatenating the two paths yields an s-t walk with r; cycles can be removed to obtain a simple path containing r. Hence the test is correct.

Complexity. Two BFS runs: $O(n + m)$ total.

Problem: FEW — minimize the number of red vertices on an s-t path

Model. Assign vertex costs $c(v) = 1$ if $v \in R$, else $c(v) = 0$. The goal is to find a simple s-t path minimizing $\sum_{v \text{ on path}} c(v)$. (If the assignment requires excluding endpoints from the count, subtract their costs as appropriate.)

Reduction to edge-weighted shortest path (vertex-split trick). For each vertex $v \in V$ create two nodes $v_{\text{in}}, v_{\text{out}}$ and add a directed edge $v_{\text{in}} \to v_{\text{out}}$ of weight $c(v)$. For every original edge $(u \to v)$ add a zero-weight edge $u_{\text{out}} \to v_{\text{in}}$. For undirected edges $\{u, v\}$ add

both directed copies. Finally run Dijkstra's algorithm (nonnegative weights) from s_{in} to t_{out}. The returned distance equals the minimum total vertex cost of an s-t path.

Optimization. Because costs are $\{0, 1\}$, a 0–1 BFS (deque) can replace Dijkstra for slightly better practical performance; however Dijkstra with a binary heap is simple and reliable.

Complexity. The transformed graph has $2n$ vertices and $m + n$ edges, so Dijkstra runs in $O((n + m) \log n)$ using a binary heap. 0–1 BFS runs in $O(n + m)$.

Problem: MANY — maximum number of red vertices on a simple s-t path (hard)

Problem statement. Find a simple s-t path that maximizes the number of red vertices visited.

Complexity / hardness theorem. **Theorem.** The decision version of MANY ("is there an s-t path that visits at least k red vertices?") is NP-complete. In particular, MANY is NP-hard via a reduction from the Hamiltonian s-t path problem.

Hardness proof (reduction). Let $H = (V_H, E_H)$ be an instance of Hamiltonian s-t path: decide whether there exists a simple path from s to t that visits every vertex in V_H exactly once. Construct an instance G for MANY by taking $G := H$ and setting $R := V_H$ (i.e., every vertex is red). Let $k := |V_H|$.

If H has a Hamiltonian s-t path P, then that path visits all $n := |V_H|$ red vertices, so G has an s-t path with at least k red vertices. Conversely, if G has an s-t simple path visiting at least $k = n$ red vertices, because there are only n vertices total, that path must visit every vertex exactly once and is therefore a Hamiltonian s-t path in H. Thus Hamiltonian s-t path reduces in polynomial time to MANY, proving NP-hardness.

Membership in NP is immediate for the decision version: a candidate path can be verified in polynomial time by checking simplicity, endpoints, and counting red vertices. Hence the decision version is NP-complete.

Practical plan (since general polytime algorithm unlikely). Because MANY is NP-hard, we adopt a mixed strategy:

- **Polynomial exact cases**: detect special graph classes and solve in polytime. In particular:

  - If G is a DAG, compute longest path by topological DP (linear time).
  - If G is a tree or has small treewidth, apply tree DP.

- **Exact exponential for small instances**: run a DFS enumerating simple paths with pruning and an upper-bound heuristic (count of remaining reachable red nodes). Include an enforced time limit per instance; abort when exceeded.
- **ILP / SAT approach**: for medium instances encode the problem into an integer linear program or SAT and run a solver (useful on many practical instances).
- **Heuristics**: greedy expansions, beam search, and randomized restarts for large instances where exact solving times out.

Record whether an instance was solved exactly or timed out; include solver/timeout metadata in the results table.

Problem: ALTERNATE — existence of an alternating-color s-t path

Definition. A path is alternating if consecutive vertices in the path have opposite colors (red/non-red). Equivalently, every edge of the path connects vertices of different color.

Algorithm. Construct the subgraph $G_{\text{alt}} = (V, E_{\text{alt}})$ where

$$E_{\text{alt}} = \{(u, v) \in E : \text{color}(u) \neq \text{color}(v)\}.$$

Run BFS from s in G_{alt} and test reachability of t.

Correctness. A path in G is alternating iff it uses only edges that connect vertices of different color, so such a path exists in G iff t is reachable from s in G_{alt}.

Complexity. Filtering edges is $O(m)$; BFS is $O(n + m)$. Overall $O(n + m)$.

Implementation and reporting conventions

- All polynomial algorithms are implemented to run in $O(n + m)$ or $O((n + m) \log n)$ as described above.
- For MANY, every run records: solver used, whether the instance is in a special class (DAG, treewidth bound), best found value, and runtime. If an exact solution is not found within the prescribed timeout, the instance is marked with ‘¿ in the results table and the solver logs are attached in the appendix.

- Empirical experiments include per-instance runtime, memory usage, and which algorithm/heuristic produced the result.

References

1. *APLgraphlib—A library for Basic Graph Algorithms in APL*, version 2.11, 2016, Iverson Project, `github.com/iverson/APLgraphlib`.[4]

2. A. Lovelace, *Algorithms and Data Structures in Pascal*, Addison–Wesley 1881.

[4] If you use references to code, books, or papers, be professional about it. Use whatever style you want, but be consistent.