Red Scare! Report

by Group G.

Results

Instance name	n	A	F	M	N	S
G-ex	8	True	0	2	3	?!
P3	3	True	1	1	-1	True
bht	5,757	False	0	?!	6	?!
common-1-100	100	False	-1	-1	-1	False
common-1-1000	1,000	False	-1	-1	-1	False
common-1-1500	1,500	False	-1	-1	-1	False
common-1-20	20	False	-1	-1	-1	False
common-1-2000	2,000	False	-1	-1	-1	False
common-1-250	250	False	-1	-1	-1	False
common-1-2500	2,500	False	1	?!	6	True
⋮						

The columns are for the problems Alternate, Few, Many, None, and Some. 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

None

For the None problem, we solved each instance G by first handling the trivial cases where the graph's vertices s and t are directly connected via an edge. This already constitutes a valid shortest path of length 1 and satisfies the constraint of having no internal red vertices. For all other cases, we run breadth-first search algorithm (BFS) in order to find the shortest path between s and t while avoiding any internal red vertices, avoiding already seen nodes to exclude walks through G with repeating vertices, and storing measured distances from s to the currently explored nodes.

BFS is guaranteed to find a path with minimal edge count between nodes, so once the t vertex is found, we know that all red vertices were avoided because of the constraint and we return the measured distance as the answer. If BFS cannot reach t from s, given the constraint, we output -1, which means that there is no such path from s

to t that avoids internal red nodes.

The running time of this solution is $O(V+E)$, where V is the number of vertices and E is the number of edges, which is linear in the size of the input graph. Our implementation processes all instances in a single pass and there is no need for any structural assumptions about the instances.

Many

In the problem Many we are given a graph G with vertices s and t, and must find the maximum number of red vertices on any simple s–t path. This is a longest simple path problem with $0/1$ vertex weights.

Since the problem is difficult on general graphs, the solver selects algorithms based on the structure of the input. We first check reachability, if t is not reachable from s, the answer is -1, decided in $O(n+m)$ time by BFS. Here, $n = |V(G)|$ is the number of vertices and $m = |E(G)|$ is the number of edges in the graph.

DAGs. If the graph is acyclic (detected by Kahn's algorithm in $O(n+m)$), we compute an exact solution by dynamic programming on a topological order. This runs in linear time.

Undirected trees. If the graph is undirected with $m = n-1$, it forms a tree. The unique s–t path is recovered by BFS and its red vertices counted. This also takes $O(n+m)$ time.

General graphs. Otherwise we run a DFS over simple paths with pruning: reachability checks, an upper bound on remaining red vertices, and a global node-expansion limit that keeps the running time polynomial. If the budget is exceeded, the instance is marked unsolved(?!), if the search finishes in time, the value is exact.

Universality for Many

The problem is NP-hard on general directed graphs, since a longest simple path instance reduces to Many by colouring every vertex red. Therefore no polynomial-time algorithm is expected for arbitrary inputs. Our solver is polynomial on two classes:

- directed acyclic graphs, recognised and solved in $O(n+m)$;
- undirected trees, recognised in $O(n+m)$ and solved by reconstructing the unique s–t path.

For all other graphs the DFS-with-pruning procedure is used with a polynomial budget, giving exact answers when it finishes and marking the instance as unsolved(?!) otherwise.

Some

For the Some problem, we solved each instance G by first checking reachability from s to t using breadth-first search algorithm (BFS) from s, so that if t is unreachable, the answer is False. Then we handled trivial cases: if either s or t is a red vertex, or all vertices in G are red, then any path contains a red vertex and we return True. Also, if the set of red vertices is empty we returned False.
We also solved the Some problem for all DAG instances, detected using Kahn's algorithm for cycle detection in directed graphs. For these instances, we computed all the vertices reachable from s using BFS, and the vertices reachable from t using a reversed-BFS. If a red vertex lies in the intersection, we return True. We were also able to solve the Some problem for all tree instances. Trees were detected by checking that the graph has exactly V-1 edges, and that is connected. For these, we used BFS to find the unique path from s to t, and then we checked for red vertices in the path. For the remaining instances, we were unable to solve the problem because, in general, the Some problem is NP-hard.
The running time of the algorithm is $O(V + E)$ for DAG and tree instances, since BFS is used to explore reachable vertices or extract the unique path. Trivial cases are handled in $O(V)$ time. For the remaining instances, any algorithm would require exponential time in the worst case, because the problem is NP-hard. The problem can be reduced to the vertex-disjoint paths problem on directed graphs, which is known to be NP-complete.

Few

The Few problem asks: given a graph with start vertex s, target vertex t, and a specified set of red vertices R, what is the minimum number of red vertices that appear on any simple path from s to t, or -1 if no such path exists. Red vertices may include s and t, and they are counted if they lie on the chosen path.

To solve this, the method models the task as a shortest-path problem where the "length" of a path is the number of red vertices it uses. Each vertex on a path contributes cost 1 if it is red and 0 otherwise, so among all s–t paths the goal is to minimize this total cost. Because these costs are non-negative, a Dijkstra-style algorithm can be used: starting from s, it propagates through the graph and always extends the currently cheapest (fewest red vertices) partial path, and the value obtained when t is first reached is exactly the minimum possible number of red vertices on any s–t path.

Alternate

For the problem ALTERNATE, we are tasked with identifying graphs that have a path from the start node to the end node with only alternating vertices between red and non-red color. We are to return 'true' in case there is such a path and 'false' otherwise.

To solve this problem we have first built a subgraph, that only contains edges where the two vertices are differently colored. Then to find a path with alternating colors we must find at least one path, as there are only alternating colored edges left in the graphs. Then we implemented a standard breadth-first search algorithm (BFS). We have managed to find a solution for all graphs. If a path is found between s and t it means there is an alternating path, otherwise such a path does not exist.

Finding the subgraph takes $O(E)$, however the graph size is often drastically reduced when removing non alternating edges, making the shortest path more efficient. After finding the subgraph the algorithm runs in linear time, $O(V + E)$, for all graphs.