Red Scare

Bertram Højer, Bjarke Christensen, Gustav Helms,
Viktor Due Pedersen, Karl Monrad Kieler & Oliver Jarvis

November 24, 2022

1 Results

We identified the three problems, to which there exists an universal solution, to be: *none, few, alternate*. The two remaining problems, *some and many*, was classified as NP-hard problems for general graphs, but solvable in polynomial time for *directed acyclic graphs (DAG)*. See sections below for details on implementation, running times and reductions. In our implementation the three P-problems successfully solved all test cases and the NP-problems could solve problems for all DAGs.

The results for all problems can be found in `results.csv`.

2 Data Structure

We implemented all graphs as directed graphs utilizing the networkx library NetworkX). When the graph was undirected we simply added directed edges in both directions. Furthermore, all graph search algorithms and graph checks where done with the implementations found in *networkx*.

3 Problems

3.1 None

Our algorithm implemented to solve the *none* problem first removes all edges that connect to red vertices. Then it uses a path searching algorithm to check whether there is exists an s-t path. The algorithm returns *true* if a path exists and *-1* if none exists. The running time of the algorithm is: $E + (V + E) = 2E + V$, i.e. one iteration through all edges and then a search algorithm e.g. BFS. Thus, the algorithm is upper bounded with: $O(E + V)$. The algorithm works on all graphs in polynomial time.

3.2 Some

For "Some" we have to return True/False if a path exists with at least one red node. Our solution deduces the answer from the output of our solution to "Many". Its output is the path with the most red nodes. If the output of many is not in set {-1, 0}, then there must exist a path between s, t and some red node. Whereby we can safely output true, and for the opposite case false. Utilizing the solution from "Many", therefore apply the same assumptions to "Some" as implemented in "Many".

3.3 Many

The *many* problem is a NP-hard problem (see reduction below), but can be solved in polynomial time for *directed acyclic graphs (DAG)*. Our DAG checks will also label two-cycles as graphs containing cycles. This is not ideal as the "Many" and "Some" problems can actually be solved for graphs that contain two-cycles, our implementation does not account for this fact. To solve the *many* problem on DAGs, we implemented a dynamic programming algorithm. The algorithm solves:

$$Opt(x) = \begin{cases} max(1 + Opt(p(x))) & \text{if } x \in R \\ max(Opt(p(x))) & \text{if } x \notin R \end{cases}$$

Where $p(x)$ is all nodes with edges going into x. We define the base case for s as:

$$Opt(s) = \begin{cases} 1 & \text{if } s \in R \\ 0 & \text{if } s \notin R \end{cases}$$

The function is initialized as $Opt(t)$ and is recursively calling itself until it reaches a lead node. If the leaf node is s, the value of Opt(s) is returned. If it reaches a leaf node which is not s then it returns $-\infty$ to penalize non-valid paths. If no paths exists the algorithm returns -1. The algorithm uses a memoization table. The running time of the algorithm is $O(|E|) \in O(|V|^2)$ as it has to fill out the memoization table of $length = |V|$.

REDUCTION: Show that we can solve any weighted longest path problem, by transforming it into an instance of the Many problem. We will make the argument that an instance of the *longest path* problem can be reduced to an instance of the Many problem; LP $\leq_p$ MP.

- Given an instance of LP_w with graph, $G(E, V, W)$, we wish to compute the length of the longest path.
- Create MP instance: G(V', E') Augmentation: Each weighted edge is transformed, to have the number of weight as intermediate red nodes

between the original start and end vertices of the edge and all weights are ignored. An edge from u to v with weight 2 would have two red 2nodes inserted between u and v.

- Solve: MP solver will output the path with the maximum number of red nodes.
- Deduce: The number of red nodes in the longest path in G' is equal to the length of the longest path in the original graph G.

So by showing that LP_w problems can be solved by reducing them into MP problems, we have shown that MP must be at least be as hard as LP_w, which is know to be NP-hard.

3.4 Few

Our implementation to the 'few' problem transforms the original graph G into G' by adding a weight of 1 to each of the edges going into a red node and a weight of 0 to all other nodes. We then use a path-finding algorithm (in this case Dijkstra's algorithm) to find the shortest path in the transformed graph. The function then returns the length of the path, which is equivalent to the amount of red nodes in the path. If no path exists it returns -1. The overall running time of the implementation is $|E|$ for the transformation of the graph and $(E + V) \cdot \log(v)$ for running Dijkstra's on the transformed graph. Thus the time-complexity of the implementation is $O((E + V) \cdot \log(v))$.

3.5 Alternate

Our implementation to solve the 'Alternate' problem starts by transforming the graph G into a graph G', where all edges between nodes of the same color are removed. Such that if the nodes in the edge (u, v) are both black or both red the edge is removed. We then run a path-finding algorithm (in this case breadth-first-search) to check whether there exists an s-t path in G'. We return *True* if there is a path and *False* if G' has s-t path. The transformation of the graph takes at most the number of edges operations, $|E|$. So the total time of the algorithm is $|E| + (E + V)$. So the running time of the implementation is $O(E + V)$.