Red Scare! Report

By *[Insert Name Here]* consisting of *Eisuke Okuda, Anders Havbro Hjulmand, Andreas Frederik Flensted Olsen, Veron Hoxha, Constantin-Bogdan Craciun.*

Results

The following table displays our results for a selection of the graphs. For the complete table of all results, see the tab-separated text file `results.txt`.

Instance name	n	m	A	F	M	N	S
G-ex	8	9	True	0	?!	3	True
common-1-1000	1000	559	False	-1	?!	-1	False
grid-50-0	2500	7301	False	0	?!	1249	True
increase-n500-3	500	61751	True	1	17	1	True
rusty-2-5000	5000	123432	False	0	?!	4	True
ski-level20-3	254	353	False	0	4	-1	True
smallworld-50-1	2500	6930	True	2	?!	-1	True
wall-z-10000	70001	80000	False	0	?!	1	?!

The columns are for the problems Alternate (A), Few (F), Many (M), None (N), and Some (S), with $n = |V|$ and $m = |E|$. For those questions where there is a reason for our inability to find a good algorithm (because the problem is hard), we wrote '?!'.

Methods

We used the library `networkx` in Python to face all problems. We solved problem **"None"**, **"Few"** and **"Alternate"** for all types of graphs. On the other hand for problem **"Many"** we solved it only for DAG's graph class, while for problem **"Some"** we solved it for all graphs except the `wall-z-`*`x`* types.

For the problem **"None"**, we constructed a graph G_{None} where all the red nodes were removed. We then used the function `shortest_path_length` from `networkx` that uses Breadth-First-Search for graphs whose edge-weights are all 1. The running time of this algorithm is polynomial with upper bound $O(n + m)$.

For the problem **"Few"** we constructed a graph G_{Few} for undirected graphs where all edges were replaced by two directed edges. Then for all edges with end-points in a red node, we add a weight w

where $w = m$. For all edges with end-point in a black node, we add a weight $w = 1$. Then we use `networkx` implementation of Dijkstra's algorithm to find the `shortest_path` from $s \to t$. We then return the number of red nodes in the shortest path. The running time of this algorithm is polynomial with upper bound $O(n + m)$.

For directed acyclic graphs (DAGs), we first perform topological sorting on the graphs, and then using dynamic programming with the following recursion formula, where *p(i)* refers to any node that has an outgoing edge to node *i*:

$$\text{OPT}(i) = \min\left(\text{OPT}(p(i)) + \mathbb{I}(i \text{ is red})\right)$$

The upper bound running time is $O(n^2)$ which is polynomial.

For the problem **"Alternate"**, we constructed a graph $G_{\text{Alternate}}$ where we only added an edge between nodes of different colors. We then used the function `has_path` from `networkx` which uses Depth-First-Search as the certifying algorithm. The upper bound running time is $O(n + m)$, which is polynomial.

The problem **"Many"** is NP-Hard. In order to solve **"Many"**, the algorithm must find all s-t paths in graph G. The problem arises when the size of the graph increases (both the number of nodes and edges), also accounting that the length of the paths can differ. We argue that $LongestPath(LP) \leq_p Many$. To reduce the longest path problem to the **"Many"**, we need to construct G' from G in which all nodes are red. In this case, to solve **"Many"** we need to find the longest path from $s \to t$ in G' which is NP-Hard.

For **"Many"**, we were only able to handle DAGs. We first perform topological sorting on the graphs, and then using dynamic programming with the following recursion, where *p(i)* refers to any node that has an outgoing edge to node *i*:

$$\text{OPT}(i) = \max\left(\text{OPT}(p(i)) + \mathbb{I}(i \text{ is red})\right)$$

The upper bound running time is $O(n^2)$ which is polynomial.

We suspect that the problem **"Some"** is NP-Hard. We were not able to handle all instances, but we were able to handle DAGs and some undirected graphs as well. To solve **"Some"** for DAGs we simply returned true if the output of **"Many"** was ≥ 1. To solve **"Some"** for *some* undirected graphs, we used *network-flow*. By connecting a source with capacity 2 to a red node and connecting a sink to s and t with capacity 1, we know that there is an s-t path that uses at least one red node if the maximum flow to the sink is 2.

However, there are some undirected graphs where network flow will give an answer that is a false positive. Such an instance is shown in Figure 1. Here the network flow will give the answer 2, but actually, the answer is False, because we are not allowed to visit node 10 twice. We thus do not run network flow on graphs whose name starts with *wall-z-x*.

In addition to *wall-z-x*, it is likely that there exist more undirected graphs where network flow will give an incorrect answer. However we do not know how to identify these graphs in polynomial time (we check if there exists a hamiltonian path from s to t, and only run network flow if that exists, but that is an NP-hard problem). Thus, the results for **"Some"** for undirected graphs may not be correct. They are only correct if there exists a Hamiltonian path from s to t that includes a red node.

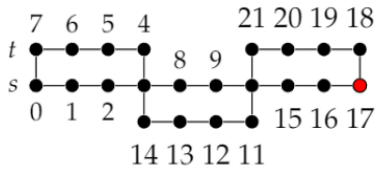

Figure 1: An example where network flow algorithm will fail for the problem "Some". Three bricks with overlap 0 wall-z-3

References

1. Hagberg, Aric and Swart, Pieter and S Chult, Daniel, *Exploring network structure, dynamics, and function using NetworkX*, Los Alamos National Lab.