Red Scare! Report

by Andreas Tietgen (anti), Albert Rise Nielsen (albn) and Amalie Bøgild Sørensen (abso).

Results

The following table gives our 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 we were unable to find a solution within reasonable time. For those questions where there is a reason for our inability to find a good algorithm (because the problem is hard), we wrote '?!'.

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

Methods

For problem A, we've used a modified version of the BFS algorihtm to find an alternating path from s to t. Depending on the color of the start vertex the algorithm queues edges that leads to a different color than the current and discards edges that leads to a vertex with the same color. In the case where the queue of alternating vertices are empty wihtout finding an alternating path to the end vertex, the algorithm will return 'false'. However, in the case where it finds the end vertex with a different color than the current visited vertex, the algorithm will return 'true' Looking at the running time of this algorithm it is O(V+E), the same as Bredth-First Search.

For problem F we've solved each instance with Dijkstra's algorithm to find the path containing the fewest amount of red vertices. We've achieved this by assigning each edge entering a red vertex a weight of 1. Every other edge is assigned a vertex of 0. Thereby the path with the smallest cost to get from s to t must be the one that enters the fewest red vertices. Since Dijkstra only allows simple paths, the cost of the path must also be the total amount of red vertices that the path enters. Therefore the cost to get to t is the total red vertices the path enters. In case there is no path from s to t the cost to get to t

will be *sys.maxsize*.

For problem M, we first check that the graph is directed and contains no cycles. This is because the problem can't be solved if there is a cycle and for undirected graphs. Thus the problem is NP-hard. On the other hand, if the graph is directed and with no cycles we compute a topological sorting starting from the start node. That is, any nodes pointing to the start node will not be part of the topological sorting. We do this with DFS by including a boolean "is_back" tag in the stack we pop from. If the tag on the current node is true we know that all nodes that can be reached from the current node have been visited, and thus we can add the node to our topological sorting. The result will thus be a reverse topological sorting, such that we can pop from the sorting in order to obtain the first node in the topological sorting. When returning from the topological sorting, we also check whether the target node is in visited, since then we know that there exists a path from start to target. If this is not the case we can safely return -1. If there is a path going from start to end vertex, we can run the longest chain algorithm in order to find the path containing the most red vertices: We pop from the topological sorting and define 'a' as 1 if this node is red, otherwise 0. We then visit all neighbours and update their entry in the dist array: if the distance to our current node + a is bigger than what is currently stored, we set the dist to exactly the distance to our current node + a. In this way we make sure that we get the chain with the most red vertices. Furthermore, we use memoization as we save the current computed dist to all vertices and reuse this in other compuations. At last, we return the dist stored for the target node, which is the maximum number of red vertices on any path from start to target. The runtime of M is O(V+E) where V are the vertices and E are the edges.

For problem N, we solved it by doing a BFS while avoiding to queue any red vertices unless it is the target node. BFS is guaranteed to find the shortest path which is what we are trying to achieve here. The runtime of BFS is O(V*E) where V are the number of vertices and E is the number of edges.

For problem S, we used the answer from problem M. Since problem M returns the maximum number of red nodes on a path from s to t we know that if this is > 0, we have to return True and otherwise False for problem S. Thus, the Some problem reduces to the Many problem. For that reason the runtime is the same as for problem M. Moreover, since Many is NP-hard and Some reduces to Many then Some is also NP-hard and can't be solved for undirected graphs and graphs with cycles.