Red Scare! Report

by Maria Techt, Daniel Hansen, Dennis Nguyen, Thor Olesen, Nicoline Scheel

Results

The table yields results for all graphs of at least 500 vertices.

Instance name	*n*	A	F	M	N	S
rusty-5757	5757	false	0	–	10	?
wall-n-1000	1000	false	0	? 1	?	
wall-z-1000	1000	false	0	? 1	?	
wall-p-1000	1000	false	0	? 1	?	
smallworld-40-1	1600	true	2	?	13	?
ski-illustration	36	false	0	?	8	?
increase-n500-3	500	true	1	?	1	?
grid-50-2	2500	false	25	?	-1	?
gnm-5000-10000-1	5000	true	8	?	5	?
G-ex	8	true	0	?	3	?
dodecahedron	20	false	1	?	1	?
common-2-5757	5757	true	2	?	4	?
bht	5757	false	0	?	6	?
⋮						

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-separted text file `results.txt`.

Methods

For the None problem we used a simple breadth first search, which checks whether a vertex is red before it includes it in any path. If it is red, it will not be included in a path. The running time for this solution is therefore that of breadth first search, which is O(N+M), where N = vertices and M = edges. We use disjoint paths to solve the Some problem with Network Flow. According to Fortune, Hopcroft and Wyllie (1980), the disjoint paths problem is NP-hard for directed graphs, so this can only be solved for undirected graphs.

We added two new nodes in the graph to represent a new source s′ and a new sink t′. We added a new edge between the s′ and the old source s and one between s′ and the old sink t. We took the first red node in the list r1 and created an edge, e, between t′ and r1.

We gave all edges in the graph a capacity of 1, except for the edge e, which gets a capacity of 2. We then used Ford Fulkerson to find the maximum flow. If the maximum flow is 2, there are two disjoint paths between s and r1 and t and r1, therefore, a path from s->r1->t, so the result would be true. If the flow is not 2, we take the next red node r2 and try again, which continues until we are out of red nodes. As the running time for Ford Fulkerson is O(f′*(m+n)) where f′ is a measure of the maximum flow, it can be reduced to O(2*(m+n))=O(m+n) in this case. However, the runtime of our algorithm will be O(r*(m+n)), where r is the number of red nodes, because we run Ford Fulkerson at most r times.

The Hamiltonian Path problem from a source s to a sink t is NP-hard. Therefore, by reducing this problem to an instance of Many, Many can be proved NP-hard as well. Given a graph G with vertices V, we want to find out if there exists a Hamiltonian path from s to t. We color all the vertices V in G red and solve the problem of Many on G. If Many results in |V| (the number of vertices), we know that there exists a Hamiltonian path from s to t. As this proves Many to be NP-hard, Many cannot provide a result for all instances of graphs in polynomial time. However, using Bellman-Ford, Many can provide a result for directed acyclic graphs (DAGs) in polynomial time.

Many is implemented for DAGs in polynomial time (O(V*E)) by using Bellman-Ford on a weighted graph based on the input graph. This is done by first converting the DAG to a weighted DAG, where each edge which is directed towards a red vertex has weight -1 and every other edge in the DAG has weight 0. The Bellman-Ford algorithm is then run on this weighted graph to find the shortest path. Due to the described weights, the shortest path will be the path with the most red vertices on, as the shortest path will be the one following edges that go to red vertices.

When solving the problem Few, we use a priority queue to pop edges with weight 0 as to avoid red vertices and favor others - thus, the vertices connected to a black node are weight 0, and the red ones are weight 1. We then run Dijkstra on this, which then favors paths with the least red vertices. The running time is the same as for Dijkstra's algorithm, that is, O(log N2̂), where N is the number of vertices.

When solving the problem alternate, we used a non-directed graph and applied breadth-first-serach (BFS) on it. For each node we select the adjacent node where it differs from the color of the current node. If a path exist between S and T where each neighbouring nodes alternates in black and blue, the system would return true, otherwise false. The running time for alternate is O(V + E)

Final Notes

The result when solving the instance of the "few" problem on the data file "data/common-2-1000.txt" outputs the following result:

- data/common-2-1000.txt
- Find Some: true
- ERR: undirected graph! Find Many:-1
- Find Few: 2
- Find Alternate: true
- Find None: 4

However, it does not make sense that the "few" instance returns 2 when the "none" instance returns 4. Namely, few returns the count of red vertices the path with fewest red vertices. Further, None returns the length of a path without red vertices. However, "few" returns the path with fewest red vertices containing 2. This contradicts the "none" case that returns 4, meaning there should be a path in the graph without red vertices of length 4.

Problems: arguably, "few" should be 0, since there is an s-t path of length 4 according to "none". Also, we run out of memory, when outputing the results for all data files to 'output.txt' using 'script.sh'

References

1. GITTA (2006): Dijkstra Algorithm: Short terms and Pseudocode, accessed last 16-11-2017 at: http://www.gitta.info/Accessibiliti/en/html/Dijkstra$_learningObject1.html$

2. Bhojasia, Manish (2011-17): Samfoundry - *Java Program to Use the Bellman-Ford Algorithm*, accessed last 10-11-2017 at: http://www.sanfoundry.com/java-program-use-bellman-ford-algorithm-find-shortest-path-between-two-vertices-assuming-that-negative-size-edges-exist-graph/