Red Scare

# In general

We used the following data structure for alle the problems:  
**Graph data structure**: A modified version of algs4 directed graph, merged with its SymbolGraph. <https://algs4.cs.princeton.edu/code/>  
We use a directed graph because it can handle both directed and undirected graphs. For undirected graphs, whenever vertices A and B are connected, we add an edge from A to B and an edge from B to A.  
Most of the algorithms we run use the fact that the vertices are indexed from 0 through n-1, with n being number of vertices. But since the vertices in red scare can have strings as names, we use a modified symbol graph. In our graph there is a map from vertex names to indices and its inverse from indices to names.  
To handle red vertices, we have an additional map that maps from a vertex to a boolean indicating whether the given vertex is red or not.  
The graph keeps track of inputs, such as count of vertices and edges. It has an adjacency list for each vertex to keep track of edges.

Total space: O(V + E), where V = vertices and E = edges.  
  
The input is only parsed once per file, and each method then receives a copy of the graph, so different exercises won’t affect the same graph.

Each exercise also has its own test method, which is simply known answers manually run by the group on some of the sample cases, which helped us debugging through solving the exercises.  
Some of the exercises also contains cross-exercise checks, such as for the Many problem. We know that If there is a path from s to t (found in None), and also no such path with red vertices (found in Some), we know the longest path with red vertices is 0 (answer for Many)!

# Results

**Format:**  
1st column: Filename + (count of vertices, count of edges, isDirectedOrUndirected)  
2nd column: Answer for None. Positive integer if reachable, and -1 if a simple path from S to T doesn’t exist.  
3rd column: Answer for Some. Either True, False or Timeout, whereas timeout denotes that we stopped checking for an answer after 2 seconds of computation time on that specific file.  
4th column: Answer for Many. -1 denotes no path from S to T. Any other integer denotes the number of red vertices from S to T. “HARD” denotes, the graph was not acyclic, and thus couldn’t be solved in polynomial time.  
5th column: Answer for Few. An integer denoting fewest number of red vertices from S to T. -1 means no simple path from S to T exists.  
6th column: Answer for Alternate. Either true or false, if an color-alternating simple path from S to T exists.

A screen shot of a computer

Description automatically generated  
A screen shot of a computer program

Description automatically generated  
A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated  
A screenshot of a computer screen

Description automatically generated  
A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated  
A black screen with white and red dots

Description automatically generated

# None

**Problem in short:** Shortest path in unweighted graph.  
There are three obvious approaches, which are either:  
1. When parsing the input, skip all red vertices  
2. After parsing, remove all red vertices from the constructed graph  
3. Modify the path finding algorithm to avoid red vertices

**Implementation details:**  
We went with option 2.  
We added the method ‘removeAllRed’ to our data structure, which simply removes all red vertices and their connected edges, except if it is the starting og ending vertex, and the dependent edges from/to the vertices, by going trough the graph in linear time.  
After that, we simply used BFS (also from the algs4 library) from vertices s to t. Since the graph is unweighted, BFS returns the shortest path if there is one.

**Works for:** All provided graphs.

**Running time:** O(V+E)

# Some

**Problem in short:** Path finding, with a condition. The simple path from S to T must contain at least 1 red vertex. This problem is NP-Hard.

**Reduction:**

**Implementation details:**Modified DFS from vertices s to t.  
DFS is preferred to BFS, as it tries one path at a time. During traversal, we keep track of visited nodes, to prevent DFS from revisiting a vertex, which would break its simple path, as it requires no revisited vertexes. If it couldn’t find a path from S to T with a red vertex, it will clear the visited tracker, making all vertexes available again, and will try a new path. As such, it might end up exploring all possible simple paths, checking if a red vertex is part of any of the paths. Normally DFS “marks” visited vertexes to avoid this. But this may exclude possible paths containing red vertices (see illustration), which is why we changed this part of the implementation.  
Therefore this algorithm doesn’t run in polynomial time, and as such a timeout has been implemented, which stops it from running after 2 seconds of attempting to find a valid simple path.

**Works for:** All (though only terminates within 2 seconds when n, m < XX)

**Running time:**  
It basically tries to do all permutations of the vertices (except start and end are always the same). Such as if we have vertices 1,2,3,4,5 which are all connected, being the worst case. Assuming s=1 and t=5 we try the following simple paths:  
1,2,3,4,5  
1,2,4,3,5  
1,3,2,4,5  
1,3,4,2,5  
1,4,2,3,5  
1,4,3,2,5

Therefore the running time is O((n-2)!)

# Many

**Problem in short/Reduction:** Longest path problem <https://en.wikipedia.org/wiki/Longest_path_problem>

Where longest here refers to number of red vertices, preferring as many as possible.

**Implementation details:**

1. If directed graph:  
Topological sort inspired by Algs4 library and by Kahns Algorithm implemented by geeksforgeeks. <https://www.geeksforgeeks.org/topological-sorting-indegree-based-solution/>  
If there is a cycle in the graph the algorithm will detect this.

If there is a cycle the problem is NP hard and we return HARD. Otherwise we do the following:

Traverse the topological order of the vertices O(V), and foreach edge O(E), relaxes the edge, which is basically checking if a longer path has been found, by counting the weight of black edges as 0 and red edges as 1, and if so we update our maxRedVerticesToV [x] array. Black edges refer to edges going into black vertices and red edges are edges going into red vertices.

2. If undirected acyclic graph:

Run BFS. If at some point we reach a marked vertex it means there is a cycle and the problem is np hard and we return HARD. Otherwise there can only be at most one path from s to t. BFS will find the path if there is one. Then we count the red vertices on the path and return the count. Otherwise return -1.

**Running time:**If acyclic graph:   
Topological order: O(V+E)  
Traversal: O(V+E):  
Therefore in total: O(V+E)  
If not acyclic graph:   
O(V + E)

**Works for:** directed and undirected acyclic graphs

# Few

**Problem in short:** Shortest Path in a Binary Weight Graph. Edges to black vertices have weight 0 and edges to red vertices have weight 1.

**Implementation details:**  
We have a 0-1 BFS. It is BFS but instead of a normal queue it uses a simple priority queue that takes all 0’s before 1’s. This is implemented as a dequeue that adds 0’s to the front and 1’s to the back. This ensures that the algorithm prefers taking fewest amount of red vertices.

**Works for:** All

**Running time:** O(V+E)

# Alternate

**Problem in short:** Path finding with restriction.

**Implementation details:**  
Modified BFS from algs4 library.  
During traversal it checks color constraints on edges, by checking current vertex color of ‘v’, then color of neighbour ‘w’ (in constant time). Only if they alternate in color (color(v) != color(w)) do we proceed to handle the edge and enqueue the vertex w.

**Works for:** All graphs

**Running time:** O(V+E)