Week 9 Tutorial Sheet

(To be completed during the Week 9 tutorial class)

Objectives: The tutorials, in general, give practice in problem solving, in analysis of algorithms and data structures, and in mathematics and logic useful in the above.

Instructions to the class: Aim to attempt these questions before the tutorial! It will probably not be possible to cover all questions unless the class has prepared them in advance. There are marks allocated towards active participation during the class. You **must** attempt the problems under **Assessed Preparation** section **before** your tutorial class and give your worked out solutions to your tutor at the start of the class – this is a hurdle and failing to attempt these problems before your tutorial will result in 0 mark for that class even if you actively participate in the class.

Instructions to Tutors:

1. The purpose of the tutorials is not to solve the practical exercises!
2. The purpose is to check answers, and to discuss particular sticking points, not to simply make answers available.

Supplementary problems: The supplementary problems provide additional practice for you to complete after your tutorial class, or as pre-exam revision. Problems that are marked as **(Advanced)** difficulty are beyond the difficulty that you would be expected to complete in the exam, but are nonetheless useful practice problems as they will teach you skills and concepts that you can apply to other problems.

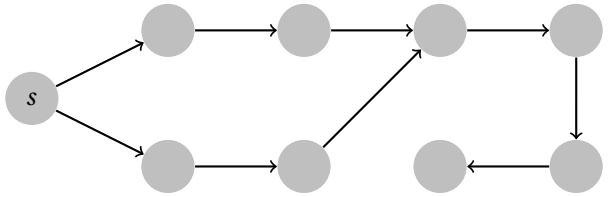
Assessed Preparation

Problem 1. Implement a directed graph class in Python. Your graph should use an adjacency list representation and should support the following operations:

- Initialise the graph with n vertices, where n is a given parameter
- Add a directed, weighted edge between the vertices u and v, with weight w

Focus on making your code easy to use and easy to understand. Avoid using Python lists as obfuscated data structures, e.g. please don't store edges like $[v_1, v_2, w]$. It is recommended that you make an `Edge` class to store edges for improved readability. Writing $e.u$, $e.v$, and $e.w$ is much nicer to read than $e[0], e[1], e[2]$, etc, particularly once you start nesting them!

Problem 2. Label the vertices of the following graph in the order that they might be visited by a depth-first search, and by a breadth-first search, from the source s.



Problem 3. Consider an undirected connected graph G. In plain words, describe how can you use depth-first search to find whether the graph G has a cycle or not.

Tutorial Problems

Problem 4. Write psuedocode for an algorithm that given a directed graph and a source vertex, returns a list of all of the vertices reachable in the graph from that source vertex. Your algorithm should run in $O(V+E)$ time.

Problem 5. Devise an algorithm for determining whether a given undirected graph is two-colourable. A graph is two-colourable if each vertex can be assigned a colour, black or white, such that no two adjacent vertices are the same colour. Your algorithm should run in $O(V+E)$ time. Write psuedocode for your algorithm

Problem 6. This problem is about cycle finding as discussed in Section 12.3 of the unit notes.

(a) Explain using an example why the algorithm given for finding cycles in an undirected graph does not work when applied to a directed graph

(b) Describe an algorithm based on depth-first search that determines whether a given directed graph contains any cycles. Your algorithm should run in $O(V+E)$ time. Write psuedocode for your algorithm

Problem 7. Recall from lectures that breadth-first search can be used to find single-source shortest paths on unweighted graphs, or equivalently, graphs where all edges have weight one. Consider the similar problem where instead of only having weight one, edges are now allowed to have weight zero or one. We call this the *zero-one* shortest path problem. Write psuedocode for an algorithm for solving this problem. Your solution should run in $O(V+E)$ time (this means that you can not use Dijkstra's algorithm!) [Hint: Combine ideas from breadth-first search and Dijkstra's algorithm]

Problem 8. Describe an algorithm for finding the **shortest** cycle in an unweighted, directed graph. A shortest cycle is a cycle with the minimum possible number of edges. You may need more than linear time to solve this one. Write psuedocode for your algorithm.

Supplementary Problems

Problem 9. Write psuedocode for a non-recursive implementation of depth-first search that uses a stack instead of recursion.

Problem 10. Write psuedocode for an algorithm that counts the number of connected components in an undirected graph that are cycles. A cycle is a non-empty sequence of edges $(u_1,u_2),(u_2,u_3),...(u_k,u_1)$ such that no vertex or edge is repeated.

Problem 11. In this question we consider a variant of the single-source shortest path problem, the *multi-source shortest path* problem. In this problem, we are given an unweighted graph and a set of many source vertices. We wish to find for every vertex v in the graph, the minimum distance to any one of the source vertices. Formally, given the sources $s_1,s_2,...s_k$, we wish to find for every vertex v

$$\mathrm{d}[v] = \min_{1\le i\le k} \mathrm{dist}(v,s_i).$$

Describe how to solve this problem using a modification to breadth-first search. Your algorithm should run in $O(V+E)$ time.

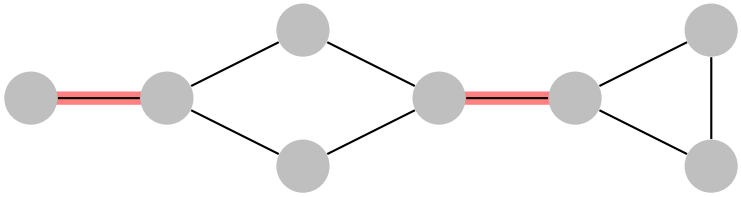
Problem 12. Recall the definition of two-colourability from Problem 5. Describe an algorithm for counting the number of valid two colourings of a given undirected graph.

Problem 13. Argue that the algorithm given in the course notes for detecting whether an undirected graph contains a cycle actually runs in $O(V)$ time, not $O(V+E)$ time, i.e. its complexity is independent of $|E|$.

Problem 14. (Advanced) Given a directed graph G in adjacency matrix form, determine whether it contains a "*universal sink*" vertex. A universal sink is a vertex v with in-degree $|V|-1$ and out-degree 0, ie. a vertex such that

every other vertex in the graph has an edge to v, but v has no edge to any other vertex. **Your algorithm should run in $O(V)$ time.** Note that this means that you can not read the entire adjacency matrix and meet the required complexity.

Problem 15. (Advanced) This problem is about determining another interesting property of a graph G, namely its *bridges*. A bridge is an edge that if removed from the graph would increase the number of connected components in the graph. For example, in the following graph, the bridges are highlighted in red.



(a) Prove that an edge is a bridge if and only if it does not lie on any simple cycle in G.

(b) Suppose that we number the vertices of G in the order that they are visited by depth-first search. Denote this quantity by `dfs_ord`$[v]$ for each vertex v. Define the following quantity `low_link`$[v]$ for each vertex v such that

$$\texttt{low\_link}[v] = \min \begin{cases} \texttt{dfs\_ord}[v], \\ \texttt{dfs\_ord}[u] : \text{for any vertex } u \text{ reachable from } v \text{ via unused edges after visiting } v \end{cases}$$

Explain how the quantity `low_link` can be computed in $O(V+E)$ time

(c) Explain how the quantities `dfs_num` and `low_link` can be used to determine which edges are bridges

(d) Write psuedocode that uses the above facts to implement an algorithm based on depth-first search for determining the bridges of a given undirected graph in $O(V+E)$ time