COMS20010 — Problem sheet 4

You don't have to finish the problem sheets before the class — focus on understanding the material the problem sheet is based on. If working on the problem sheet is the best way of doing that, for you, then that's what you should do, but don't be afraid to spend the time going back over the quiz and videos instead. (Then come back to the sheet when you need to revise for the exam!) I'll make solutions available shortly after the problem class. As with the Blackboard quizzes, question difficulty is denoted as follows:

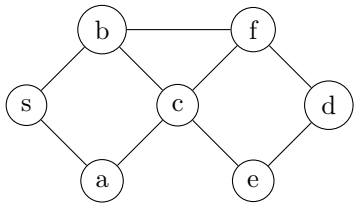
- ⋆ You'll need to understand facts from the lecture notes.
- ⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in unfamiliar situations.
- ⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes and also move a bit beyond them, e.g. by seeing how to modify an algorithm.
- ⋆⋆⋆⋆ You'll need to understand and apply facts and methods from the lecture notes in a way that requires significant creativity. You should expect to spend at least 5–10 minutes thinking about the question before you see how to answer it, and maybe far longer. Only 20% of marks in the exam will be from questions set at this level.
- ⋆⋆⋆⋆⋆ These questions are harder than anything that will appear on the exam, and are intended for the strongest students to test themselves. It's worth trying them if you're interested in doing an algorithms-based project next year — whether you manage them or not, if you enjoy thinking about them then it would be a good fit.

If you think there is an error in the sheet, please post to the unit Team — if you're right then it's much better for everyone to know about the problem, and if you're wrong then there are probably other people confused about the same thing.

This problem sheet covers week 4, focusing on graph representations, depth- and breadth-first search, and Dijkstra's algorithm.

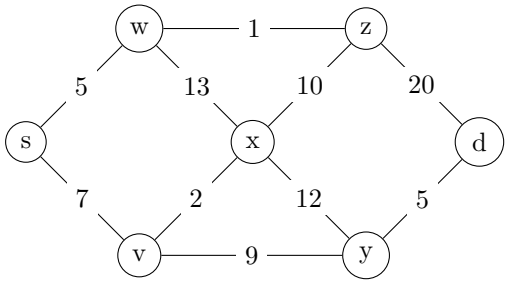
1. Let $G = (V, E)$ be a connected graph, let $x \in V$, and run depth-first search on G starting from x. Let T be the graph on V whose edges are the edges of G traversed by depth-first search — that is, edges $\{u, v\}$ such that at some point in the algorithm `helper`(v) is called from `helper`(u) while v is unexplored.
   (a) [⋆⋆⋆] Prove that T is a tree.
   (b) [⋆⋆⋆] Considering T to be rooted at x, prove that if $\{u, v\} \in E$, then either u is an ancestor of v in T or vice versa. (In other words, T is a DFS tree for G.)

2. We will begin by considering the following unweighted graph.



   (a) [⋆] Write down the adjacency list representation and adjacency matrix representation of this graph.
   (b) [⋆⋆] Run the BFS algorithm on this unweighted graph to find the distance from s to d. Show your working by showing the order the algorithm visits each vertex as well as the distances it finds to each vertex.

(c) [⋆⋆] Now run Dijkstra's algorithm on the following weighted graph to find the distance from s to d. Show your working by showing the order the algorithm visits each vertex as well as the distances it finds to each vertex.



(d) [⋆] Write down the adjacency list and adjacency matrix representations of this graph.

(e) [⋆] How is Dijksta's algorithm different or similar to the BFS algorithm?

3. [⋆⋆] John is trying to solve the single-source shortest path problem on a weighted graph with positive weights. He decides that Dijkstra's algorithm is too complicated, and that he should be able to get a "good enough" answer by ignoring the weights and using breadth-first search. Give an example to show that this approach could return a path which is a million times too long.

4. (a) [⋆⋆] Give an example in which breadth-first search as described in lectures, with inputs $G = (V, E)$ and $v \in V$, requires $\Omega(|V|^2)$ space.

   (b) [⋆⋆⋆] Suggest a way of reducing the space requirement to $O(|V|)$.

5. [⋆⋆⋆] Give an algorithm which, given a directed graph $G = (V, E)$ and two disjoint lists of vertices $X, Y \subseteq V$, returns a shortest path from any vertex in X to any vertex in Y in $O(|V| + |E|)$ time. (In other words, the path should start in X, end in Y, and be of length $\min\{d(x, y) \colon x \in X, y \in Y\}$.) You may assume that at least one such path exists. (**Hint:** The "nice" way of doing this involves using an algorithm you already know in a clever way, not coming up with a new one.)

6. [⋆⋆] Let $G = (V, E)$ be a connected graph, and suppose that T is **both** a DFS tree **and** a BFS tree for G. Prove that this implies G is a tree (so that $G = T$).

7. (a) [⋆⋆⋆] Give an algorithm which, given a connected undirected graph $G = (V, E)$ in adjacency list form, decides whether G contains a cycle with an odd number of edges in $O(|E|)$ time. (**Hint:** A BFS tree will be useful.)

   (b) [⋆⋆] How would you adapt your algorithm to work with arbitrary graphs G which may or may not be connected?

8. You are trying to plan a train journey within the UK as cheaply as possible. Unfortunately, ticket pricing in the UK does not make sense — often the cheapest way to go from point A to point B is to go via points C, D and E.

   (a) [⋆⋆] Explain how to use Dijkstra's algorithm to find the cheapest journey from A to B, given a list of possible journeys and prices. You may ignore arrival and departure times — if your shortest route is $A \to B \to C$, then on arrival at B you will simply wait for the next train from B to C.

   (b) [⋆⋆⋆] On top of the strange ticket pricing, several train companies are deeply unreliable and may leave you stranded at a station (or even on a track) for hours. You have had particularly bad experiences with one particular company, let's call them Sirius Rail, and wish to give them as little money as possible. Given a list of the journeys they run, adapt your answer to part a) to first minimise the amount of money you pay to Sirius Rail, then minimise the total cost of the route subject to that.

9. (a) [⋆⋆] Let G be a graph with vertex set $[n]$ and adjacency matrix A. Prove by induction that for all $t \geq 1$ and all $i, j \in [n]$, $(A^t)_{i,j}$ is the number of **walks** from i to j of length t.

(b) [$\star\star\star$] Give a recursive algorithm which, given an $n \times n$ matrix A and an integer $t \geq 0$, calculates A^t using $O(\log n)$ matrix multiplications.

(c) [$\star\star$] Using parts a) and b), give an algorithm which lists all pairs of vertices $i, j \in [n]$ which are distance at most k apart, using at most $O(n^3 \log k)$ arithmetic operations.

(d) [$\star$] Is the algorithm of part c) ever useful?

(e) [$\star\star\star\star$] By calculating A^k more efficiently, the running time for part c) can be cut to $O(n^\omega \log k)$ arithmetic operations, where $\omega \approx 2.373$. (Yes, ω is the standard name for this constant, and yes, this is a very stupid choice given that ω-notation exists.) Will this version of the algorithm ever be useful?

10. You are building a package management system, like apt-get for Linux or pip for Python. A basic problem such systems have to solve is resolving dependencies — a user wishes to install several packages at once, and some of these packages require other packages to be installed first. This question will walk you through a way of using DFS trees to build a system able to handle this without further input from the user.

    We can view the situation as a directed graph G, where the vertex set P is the set of all available packages. If there is an edge from x to y, this indicates that x *directly depends* on y, and that x cannot be installed unless y is present. *Indirect dependencies* are also possible — we might have (x, y), $(y, z) \in E(P)$, so that x cannot be installed without first installing z, but $(x, z) \notin E(P)$.

    (a) [$\star$] Explain briefly why it is reasonable to assume that G does not contain a (directed) cycle. We call such graphs *directed acyclic graphs* or *dags*.

    (b) [$\star\star\star\star$] Give an algorithm which, given a package x and a dag G in adjacency list form, outputs an **ordered list** of which packages must be installed in order to install x. All the direct and indirect dependencies of any given package should appear before it on the list, and your algorithm should run in $O(|E(G)|)$ time. This is called a *topological order*.