1. (10 pts) Let $G = (V, E)$ be a graph with an edge-weight function w, and let the tree $T \subseteq E$ be a minimum spanning tree on G. Now, suppose that we modify G slightly by decreasing the weight of exactly one of the edges in $(x, y) \in T$ in order to produce a new graph G'. Here, you will prove that the original tree T is still a minimum spanning tree for the modified graph G'.
   To get started, let k be a positive number and define the weight function w' as

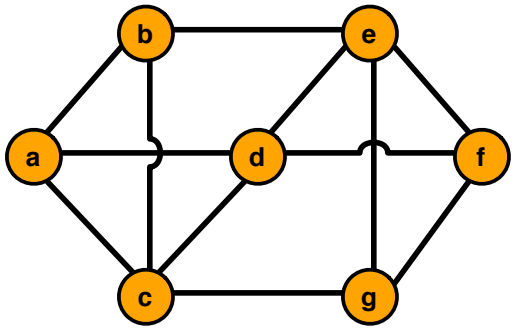
$$w'(u,v) = \begin{cases} w(u,v) & \text{if } (u,v) \neq (x,y) \\ w(x,y) - k & \text{if } (u,v) = (x,y) \end{cases} .$$

   Now, prove that the tree T is a minimum spanning tree for G', whose edge weights are given by w'.

2. (20 pts) Professor Snape gives you the following unweighted graph and asks you to construct a weight function w on the edges, using positive integer weights only, such that the following conditions are true regarding minimum spanning trees and single-source shortest path trees:

   - The MST is distinct from any of the seven SSSP trees.
   - The order in which Jarník/Prim's algorithm adds the safe edges is different from the order in which Kruskal's algorithm adds them.
   - Borůvka's algorithm takes at least two rounds to construct the MST.

   Justify your solution by (i) giving the edges weights, (ii) showing the corresponding MST and all the SSSP trees, and (iii) giving the order in which edges are added by each of the three algorithms. (For Borůvka's algorithm, be sure to denote which edges are added simultaneously in a single round.)



3. (10 pts extra credit) Crabbe and Goyle think they have come up with a way to get rich by playing the foreign exchange markets in the wizarding world. Their idea is to exploit these exchange rates in order to transform one unit of British wizarding money into more than one unit of British wizarding money, through a sequence of money exchanges. For instance, suppose 1 British wizarding penny buys 0.82 French wizarding pennies, 1 French wizarding penny buys 129.7 Russian wizarding pennies, and finally 1 Russian wizarding penny buys 0.0008 British wizarding pennies. By converting these coins, Crabbe and Goyle think they could start with 1 British wizarding penny and buy $0.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02$ British wizarding pennies, thereby making a 2% profit! The problem is that those goblins at Gringots charge a transaction cost for each exchange.

   Suppose that Crabbe and Goyle start with knowledge of n wizard monies $c_1, c_2, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of wizard money c_i buys $R[i, j]$ units of wizard money c_j. A traditional *arbitrage opportunity* is thus a cycle in the induced graph such that the product of the edge weights is greater than unity. That is, a sequence of currencies $\langle c_{i_1}, c_{i_2}, \ldots, c_{i_k} \rangle$ such that $R[i_1, i_2] \times R[i_2, i_3] \times \cdots \times R[i_{k-1}, i_k] \times R[i_k, i_1] > 1$. Each transaction, however, must pay Gringots a fraction α of the total transaction value, e.g., $\alpha = 0.01$ for a 1% rate.

   (a) When given R and α, give an efficient algorithm that can determine if an arbitrage opportunity exists. Analyze the running time of your algorithm.

   Hermione's hint: It is possible to solve this problem in $O(n^3)$. Recall that Bellman-Ford can be used to detect negative-weight cycles in a graph.

   (b) For an arbitrary R, explain how varying α changes the set of arbitrage opportunities that exist and that your algorithm might identify.

4. (40 pts) Bidirectional breadth-first search is a variant of standard BFS for finding a shortest path between two vertices $s, t \in V(G)$. The idea is to run *two* breadth-first searches simultaneously, one starting from s and one starting from t, and stop when they "meet in the middle" (that is, whenever a vertex is encountered by both searches). "Simultaneously" here doesn't assume you have multiple processors at your disposal; it's enough to alternate iterations of the searches: one iteration of the loop for the BFS that started at s and one iteration of the loop for the BFS that started at t.

   As we'll see, although the worst-case running time of BFS and Bidirectional BFS are asymptotically the same, in practice Bidirectional BFS often performs significantly better.

   Throughout this problem, all graphs are unweighted, undirected, simple graphs.

(a) Give examples to show that, in the worst case, the asymptotic running time of bidirectional BFS is the same as that of ordinary BFS. Note that because we are asking for asymptotic running time, you actually need to provide an infinite family of examples (G_n, s_n, t_n) such that $s_n, t_n \in V(G_n)$, the asymptotic running time of BFS and bidirectional BFS are the same on inputs (G_n, s_n, t_n), and $|V(G_n)| \to \infty$ as $n \to \infty$.

(b) Recall that in ordinary BFS we used a `state` array (see Lecture Notes 8) to keep track of which nodes had been visited before. In bidirectional BFS we'll need *two* `state` arrays, one for the BFS from s and one for the BFS from t. Why? Give an example to show what can go wrong if there's only one `state` array. In particular, give a graph G and two vertices s, t such that some run of a bidirectional BFS says there is no path from s to t when in fact there is one.

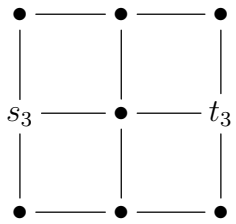
(c) Implement from scratch a function `BFS(G,s,t)` that performs an ordinary BFS in the (unweighted, directed) graph G to find a shortest path from s to t. Assume the graph is given as an adjacency list; for the list of neighbors of each vertex, you may use any data structure you like (including those provided in standard language libraries). Have your function return a pair (d, k), where d is the distance from s to t (-1 if there is no s to t path), and k is the number of nodes popped off the queue during the entire run of the algorithm.

(d) Implement from scratch a function `BidirectionalBFS(G,s,t)` that takes in a(n unweighted, directed) graph G, and two of its vertices s, t, and performs a bidirectional BFS. As with the previous function, this function should return a pair (d, k) where d is the distance from s to t (-1 if there is no path from s to t) and k is the number of vertices popped off of both queues during the entire run of the algorithm.

(e) For each of the following families of graphs G_n, write code to execute `BFS` and `BidirectionalBFS` on these graphs, and produce the following output:

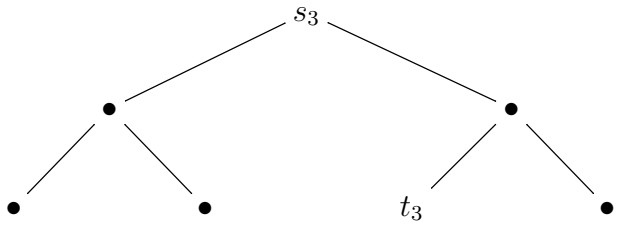
- In text, the pairs (n, d_1, k_1, d_2, k_2) where n is the index of the graph, (d_1, k_1) is the output of `BFS` and (d_2, k_2) is the output of `BidirectionalBFS`.
- a plot with n on the x-axis, k on the y-axis, and with two line charts, one for the values of k_1 and one for the values of k_2:

i. Grids. G_n is an $n \times n$ grid, where each vertex is connected to its neighbors in the four cardinal directions (N,S,E,W). Vertices on the boundary of the grid will only have 3 neighbors, and corners will only have 2 neighbors. Let s_n be the midpoint of one edge of the grid, and t_n the midpoint of the opposite edge. For example, for $n = 3$ we have:



(When n is even s_n and t_n can be either "midpoint," since there are two.) Produce output for $n = 3, 4, 5, \ldots, 20$.

ii. Trees. G_n is a complete binary tree of depth n. s_n is the root and t_n is any leaf. Produce output for $n = 3, 4, 5, \ldots, 15$. For example, for $n = 3$ we have:



iii. Random graphs. G_n is a graph on n vertices constructed as follows. For each pair of of vertices (i, j), get a random boolean value; if it is `true`, include the edge (i, j), otherwise do not. Let s_n be vertex 1 and t_n be vertex 2 (food for thought: why does it not matter, on average, which vertices we take s, t to be?) For each n, produce 50 such random graphs and report just the average values of (d_1, k_1, d_2, k_2) over those 50 trials. Produce this output for $n = 3, 4, 5, \ldots, 20$.