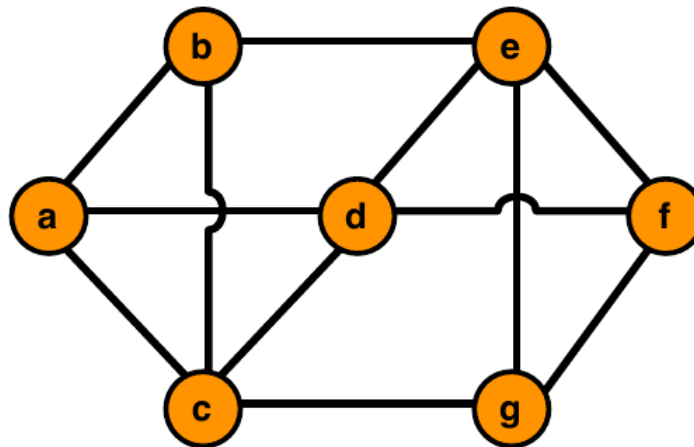


1. (30 points) Grog gives you the following unweighted graph and asks you to construct a weight function w on the edges, using positive integer weights only, such that the following conditions are true regarding minimum spanning trees (MST) and single-source shortest path trees (SSSP):

- 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.

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 two algorithms.



2. (25 points) Harry and Shadow think they have come up with a way to get rich by exploiting the ore market. Their idea is to exploit exchange rates in order to transform one unit of gold into more than one unit of gold, through a sequence of exchanges. For instance, suppose 1 unit of gold buys 0.82 silver units, 1 silver unit buys 129.7 ethereum units, 1 ethereum unit buys 12 doge units, and finally 1 doge unit buys 0.0008 gold units. By converting their loot, Harry and Shadow think they could start with 1 gold unit and buy $0.82 \times 129.7 \times 12 \times 0.0008 \approx 1.02$ gold units, thereby making a 2% profit! The problem is that those dwarves at Rocky Mountain Bank charge a transaction cost for each exchange.

Suppose that Harry and Shadow start with knowledge of n ores c_1, c_2, \dots, c_n and an $n \times n$ table R of exchange rates, such that one unit of ore c_i buys $R[i, j]$ units of ore 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 ores $\langle c_{i_1}, c_{i_2}, \dots, c_{i_k} \rangle$ such that $R[i_1, i_2] \times R[i_2, i_3] \times \dots \times R[i_{k-1}, i_k] \times R[i_k, i_1] > 1$. Each transaction, however, must pay Rocky Mountain Bank a fraction α of the total transaction value, e.g., $\alpha = 0.01$ for a 1% rate.

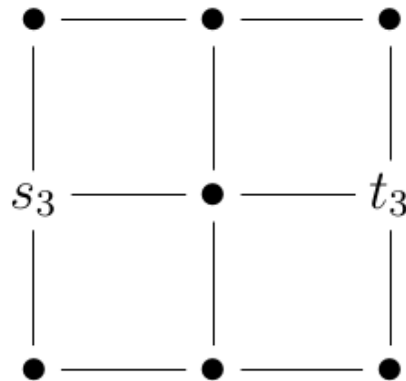
- (a) When given R and α , give an efficient algorithm that can determine if an arbitrage opportunity exists. Analyze the running time of your algorithm.
Thormund's hint: It is possible to solve this problem in $O(n^3)$.
- (b) For an arbitrary R , explain how varying α changes the set of arbitrage opportunities that exist and that your algorithm might identify.

3. (55 points) 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.

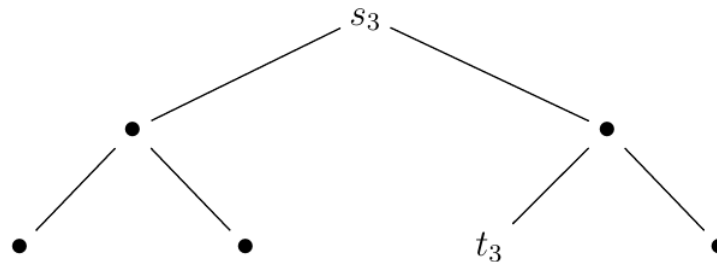
Throughout this problem, all graphs are unweighted, undirected, simple graphs.

- (a) 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.
- (b) Implement from scratch a function **BidirectionalBFS**(G, s, t) that takes in an 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.
- (c) 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 $n = 3, 4, 5, \dots, 20$.

- 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 $n = 3, 4, 5, \dots, 15$. 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 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 $n = 3, 4, 5, \dots, 20$.