Prof. Pietro S. Oliveto

CS217 Data Structures and Algorithm Analysis
Fall term 2023

Solutions for Exercise Sheet 13

Handout: December 12th — Deadline: December 19th, 4pm

Question 13.1 (0.5 marks)

Consider a directed graph $G(V, E)$.

1. With an adjacency list representation, how long does it take to compute the in-degree of a vertex $v \in V$?
2. With an adjacency list representation, how long does it take to compute the in-degrees of all vertices?
3. With an adjacency matrix representation, how long does it take to compute the in-degree of a vertex $v \in V$?
4. With an adjacency matrix representation, how long does it take to compute the in-degree of a vertex $v \in V$?

Solution:

1. We have to scan through the adjacency lists of each vertex and see how many times vertex v is pointed to. Thus the runtime is $O(V + E)$.
2. We need to do the same as above except that this time we need to keep V counters, one per node. The runtime is still $O(V + E)$.
3. We have to scan the column (i, v) in the matrix, so the runtime is $O(V)$.
4. We have to scan the whole matrix column by column, thus the runtime is $O(V^2)$.

Question 13.2 (0.5 marks)

A mayor of a city decides to monitor every road in the city with 360° video surveillance cameras. Imagine the road network as an undirected graph where edges represent roads and vertices represent junctions. When a video camera is placed on a junction, it can monitor all incident roads.

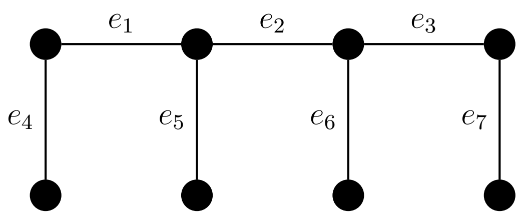
In graph terms, an edge is called *monitored* if there is a camera on at least one of its vertices. The goal is to identify on which vertices to put cameras in order to monitor every edge with a minimum number of cameras.

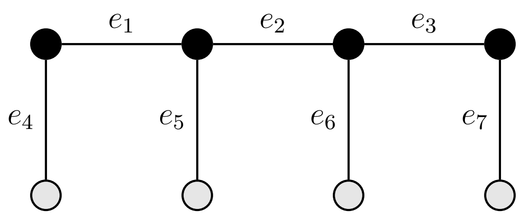
The mayor decides to use the following strategy: *While there is an unmonitored edge, put video cameras on* ***both*** *of its vertices.*

How good is this strategy? Does it always produce an optimal solution? Does it come close? Justify your answer.

Can you think of a greedy strategy for this problem?

Solution: This problem is well known under the name Minimum Vertex Cover (or just Vertex Cover). The mayor's strategy does not always find a global optimum, see the following examples with selected vertices drawn in black. The choice of which unmonitored edge to pick can be important. The left-hand side solution has picked the vertical edges e_4, e_5, e_6, e_7 and thus selected all vertices. The right-hand side solution has picked e_1 and e_3 and has managed to find an optimal solution.





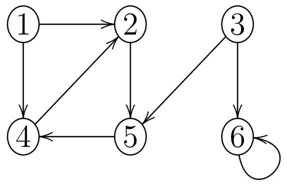
When scaling these graphs horizontally in the obvious way, we get differences between $n/2$ selected vertices for the optimal solution and n selected vertices for a worst-case solution.

This is not a coincidence: one can show that on every graph, the algorithm always produces a solution with at most twice as many selected vertices as the optimal solution. Such a solution is called a *2-approximation*. No algorithm performing much better in the worst case is known for the problem. The above examples hence illustrate a best-case and a worst-case solution resulting from the algorithm (depending on the order of edges).

The goal is to select the smallest set of nodes that covers all the edges. Hence a greedy strategy for the problem would iteratively select the node with the most *currently uncovered* adjacent edges (and consequently remove this node and its adjacent edges from the graph for the next iterative step). This choice is greedy because, at every step, it selects the node that minimises the number of remaining uncovered edges for the next step. Hence, it is the locally optimal choice. The performance of the greedy strategy can be shown to be arbitrarily bad by constructing a worst-case instance for the purpose.

Question 13.3 (0.25 marks)

Perform a breadth-first search on the following graph with vertex 3 as source. Show the d and π values of each node.



Solution.

	d	π
1	∞	NIL
2	3	4
3	0	NIL
4	2	5
5	1	3
6	1	3

Question 13.4 (0.25 marks)

State what happens if BFS uses a single bit to store the colour of each vertex (0 for white and 1 for gray) and thus the last line of the algorithm is removed.

Solution. The algorithm never specifically checks whether nodes are black and does not enqueue nodes that are gray. Thus, the behaviour of the algorithm remains unchanged and a bit can be used to flag whether each node is white (i.e., 0) or gray (i.e., 1). The book uses black nodes for pedagogical reasons, to illustrate when the processing of nodes is completed.

Question 13.5 (0.25 marks)

What is the running time of BFS if an adjacency matrix representation is used instead of an adjacency list?

Solution. It would take $O(V^2)$ to identify the neighbours of all nodes to decide which to enqueue in the last **for** loop. Thus the runtime would be $O(V + V^2) = O(V^2)$.

Question 13.6 (1 mark) Implement BFS(G, s) for a given undirected graph $G(V, E)$ and the procedure PRINT-PATH(G, s, v). The input will be:

- first line: N M (the number of vertices and edges).
- second line: the source node s.
- M lines each containing a pair $v_i v_j$ meaning there is an edge between these two nodes.
- the final two lines contain one node each v_y and v_z to be given input to PRINT-PATH(G, s, v).

You have to first build the adjacency list representing the graph with the required attributes (colour, .d, .π). And then run the two algorithms on the graph G.

The output will be the two paths from s to v_y and from s to v_z.

You can use library functions for Enqueue and Dequeue if you prefer.