Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Departement of Computer Science
Johannes Lengler, David Steurer
Kasper Lindberg, Lucas Slot, Hongjie Chen, Manuel Wiedmer

18 November 2024

Algorithms & Data Structures — Exercise sheet 9 — HS 24

The solutions for this sheet are submitted on Moodle until 24 November 2024, 23:59.

Exercises that are marked by * are challenge exercises. They do not count towards bonus points.

You can use results from previous parts without solving those parts.

Exercise 9.1 *Transitive graphs.*

Let $G = (V, E)$ be an undirected graph. We say that G is

- **transitive** when for any two edges $\{u, v\}$ and $\{v, w\}$ in E, the edge $\{u, w\}$ is also in E;
- **complete** when its set of edges is $\{\{u, v\} \mid u, v \in V, u \neq v\}$;
- the **disjoint union** of $G_1 = (V_1, E_1), \ldots, G_k = (V_k, E_k)$ iff $V = V_1 \cup \cdots \cup V_k$, $E = E_1 \cup \cdots \cup E_k$, and $V_1, \ldots, V_k$ are pairwise disjoint.

Show that an undirected graph G is transitive if and only if it is a disjoint union of complete graphs.

Exercise 9.2 *Short statements about directed graphs* **(1 point)**.

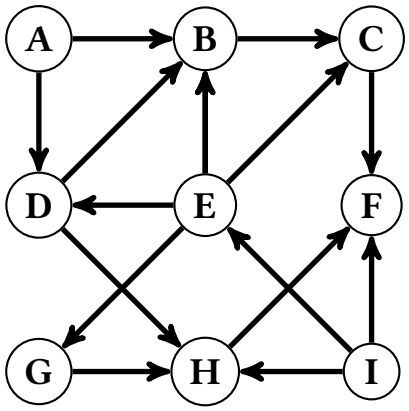
In the following, let $G = (V, E)$ be a directed graph. For each of the following statements, decide whether the statement is true or false. If the statement is true, provide a proof; if it is false, provide a counterexample.

(a) If G has no sources, it must have a directed cycle[1] (a source is a vertex with in-degree 0).

(b) If both the in-degree and out-degree of each vertex in G are even, then G contains a directed Eulerian walk (i.e., a directed walk which uses each edge exactly once).

(c) The following graph has a topological sorting. If so, give a topological sorting; if not, prove why no topological sorting can exist.

[1] A directed cycle is a closed directed walk of length at least 2 for which all vertices are pairwise distinct except the endpoints.



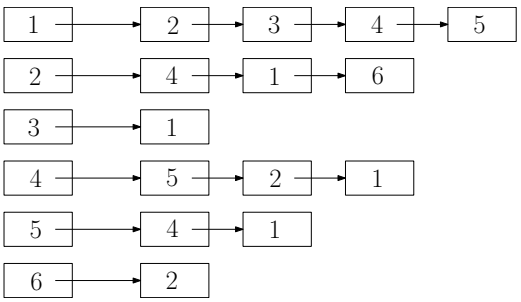
Exercise 9.3 *Data structures for graphs.*

Consider the following three types of data structures for storing an undirected graph $G = (V, E)$ with $V = [n]$ and $|E| = m$: (the following three instances of data structures are for a graph with $n = 6$ and $m = 7$)

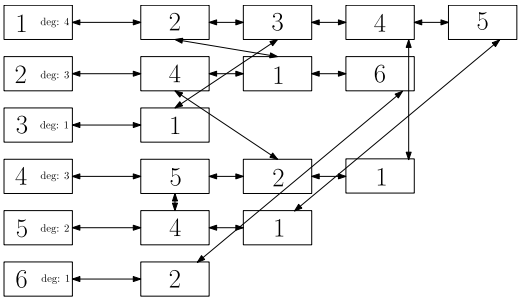
1) Adjacency matrix:

0	1	1	1	1	0
1	0	0	1	0	1
1	0	0	0	0	0
1	1	0	0	1	0
1	0	0	1	0	0
0	1	0	0	0	0

2) Adjacency lists: For each vertex v, we store its neighbors in a singly linked list $\mathrm{Adj}[v]$. Given a vertex $v \in V$, we can access the head of $\mathrm{Adj}[v]$ in constant time.



3) Improved adjacency lists: For each vertex, we store its neighbors in a doubly linked list $\mathrm{Adj}[v]$. Given a vertex $v \in V$, we can access the head of $\mathrm{Adj}[v]$ in constant time. We also store the degree of each vertex $v \in V$ in the head of $\mathrm{Adj}[v]$. Additionally, for each edge $\{u, v\}$, there are pointers between the element corresponding to v in $\mathrm{Adj}[u]$ and the element corresponding to u in $\mathrm{Adj}[v]$.



Question: For each of the above data structures, what is the required memory in Θ-notation?

Question: Which worst-case runtime do we have for the following queries? Give your answer in Θ-notation depending on n, m, and/or $\deg(u)$ and $\deg(v)$ (if applicable).

(a) Input: A vertex $v \in V$. Find $\deg(v)$.

(b) Input: A vertex $v \in V$. Find a neighbor of v (if a neighbor exists).

(c) Input: Two vertices $u, v \in V$. Decide whether u and v are adjacent.

(d) Input: An edge $\{u, v\} \in E$. Delete the edge $\{u, v\}$ from the graph.

(e) Input: A vertex $u \in V$. Find a neighbor $v \in V$ of u and delete the edge $\{u, v\}$ from the graph.

(f) Input: Two vertices $u, v \in V$ with $u \neq v$. Insert an edge $\{u, v\}$ into the graph if it does not exist yet. Otherwise do nothing.

(g) Input: A vertex $v \in V$. Delete all edges incident to v from the graph.

Question: For the last two queries, describe your algorithm.

Exercise 9.4 *Longest path in a DAG* **(1 point)**.

Let $G = (V, E)$ be a directed graph without directed cycles (i.e., a directed acyclic graph or short DAG). Assume that $V = \{v_1, \ldots, v_n\}$ (for $n = |V| \in \mathbb{N}$) and that the sorting $v_1, v_2, \ldots, v_n$ of the vertices is a topological sorting. The goal of this exercise is to find the longest path in G.

(a) Let P be a path in G. Prove that if $P = (v_{i_1}, v_{i_2}, \ldots, v_{i_k})$, then $i_1 < i_2 < \cdots < i_k$.

(b) Describe a bottom-up dynamic programming algorithm that, given a graph G with the property that $v_1, \ldots, v_n$ is a topological sorting, returns the length of the longest path in G in $O(|V| + |E|)$ time. You can assume that the graph is provided to you as a pair (n, Adj) of the integer $n = |V|$ and the adjacency lists Adj. Your algorithm can access $Adj[u]$, which is a list of vertices to which u has a direct edge, in constant time. Formally, $Adj[u] := \{v \in V \mid (u, v) \in E\}$.

In your solution, address the following aspects:

1. *Dimensions of the DP table*: What are the dimensions of the DP table?
2. *Subproblems*: What is the meaning of each entry?
3. *Recursion*: How can an entry of the table be computed from previous entries? Justify why your recurrence relation is correct. Specify the base cases of the recursion, i.e., the cases that do not depend on others.
4. *Calculation order*: In which order can entries be computed so that values needed for each entry have been determined in previous steps? Describe the calculation order in pseudocode.

5. *Extracting the solution*: How can the solution be extracted once the table has been filled?

6. *Running time*: What is the running time of your solution?

Hint: *Define the entry of the DP table as $DP[i]$ = length of longest path in G ending at vertex v_i.*

Exercise 9.5 *DFS does not solve closed Eulerian walk* **(1 point)**.

Consider the following algorithm which takes as input an Eulerian graph G with n vertices V and an edge set E and outputs a walk:

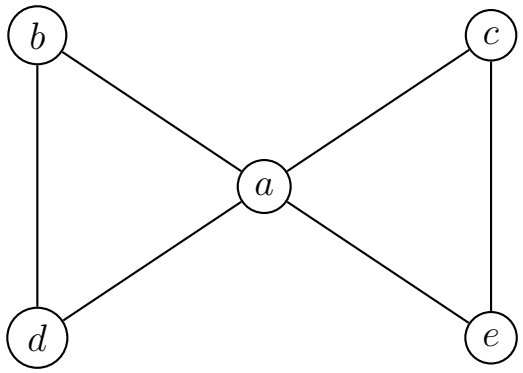
Algorithm 1 'Supposed closed Eulerian walk'-finding algorithm

```
Run DFS on G starting from a vertex v_1 ∈ V                    ▷ First step
(v_1, v_2, ..., v_n) ← DFS pre-order

u ← v_1                                                        ▷ Second step
P ← (v_1)
while u has at least one neighbor do
    w ← neighbor of u which has the largest index in the DFS pre-order
    Append w to P
    Remove {u, w} from E
    u ← w
return P
```

Remark. Given the DFS pre-order $(v_1, v_2, \ldots, v_n)$, we say v_i has larger index in this pre-order than v_j if $i > j$.

(a) In this part we will see an example of when the algorithm does in fact produce a closed Eulerian walk. Consider the following graph



Execute the algorithm on this graph using a as the starting vertex and lexicographic ordering for the DFS subroutine. This means that if we start at vertex b for example, then the DFS subroutine prioritizes going to a first.

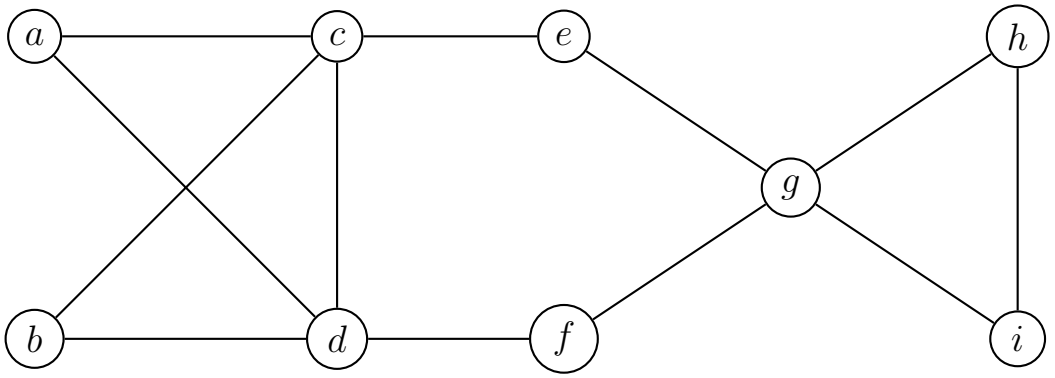
Show the pre-order generated by the DFS, the final DFS tree and the walk outputted by the algorithm.

Confirm that this walk is indeed a closed Eulerian walk.

(b) However, this algorithm does not always produce a closed Eulerian walk. Consider the graph below. Find a starting vertex and a pre-order of the vertices of the graph below such that running the second

step of the algorithm with that pre-order produces a walk which is not a closed Eulerian walk.

Note that this pre-order can be generated by any possible DFS tree, meaning the lexicographic ordering of the vertices should not be taken into account. The labeling of the vertices are purely to help with writing out the answers.



Show the starting vertex, the pre-order you chose, the DFS tree which would generate this pre-order and the walk outputted by the algorithm.