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
Lucas Slot, Manuel Wiedmer, Hongjie Chen, Ding Jingqiu

20 November 2023

Algorithms & Data Structures — Exercise sheet 9 — HS 23

The solutions for this sheet are submitted at the beginning of the exercise class on 27 November 2023.

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 the $(V_i)_{1 \leq i \leq k}$ are pairwise disjoint.

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

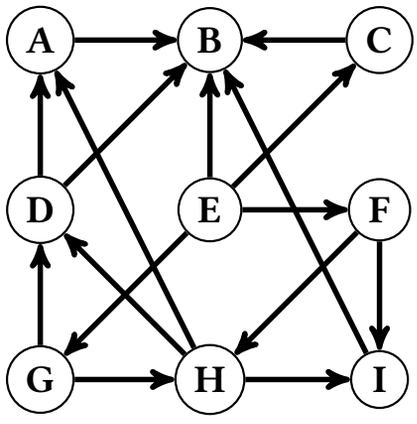
Exercise 9.2 *Short statements about graphs (cont'd)* **(1 point)**.

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 for every vertex $v \in V$ its in-degree $\deg_{\text{in}}(v)$ is even, then $|E|$ is even.

(b) For a longest directed path $P : v_0, \ldots, v_\ell$ in G, the endpoint has to be a sink.

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

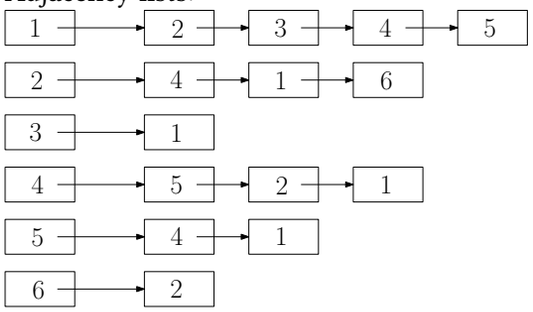


Exercise 9.3 *Data structures for graphs.*

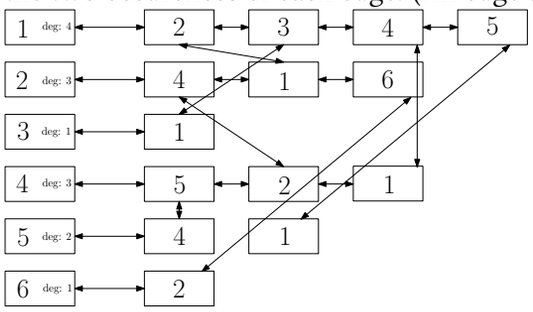
Consider three types of data structures for storing an undirected graph G with n vertices and m edges:

1) Adjacency matrix.

2) Adjacency lists:



3) Adjacency lists, and additionally we store the degree of each node, and there are pointers between the two occurences of each edge. (An edge appears in the adjacency list of each endpoint).



For each of the above data structures, what is the required memory (in Θ-Notation)?

Which runtime (worst case, in Θ-Notation) do we have for the following queries? Give your answer 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: Two adjacent vertices $u, v \in V$. Delete the edge $e = \{u, v\}$ 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 v and all incident edges from the graph.

For the last two queries, describe your algorithm.

Exercise 9.4 *Number of paths in DAGs* **(1 point)**.

Let $G = (V, E)$ be a directed graph without directed cycles[1] (i.e., a directed acyclic graph or short DAG). Assume that $V = \{v_1, \ldots, v_n\}$ (for $n = |V| \in \mathbb{N}$). Further assume that v_1 is a source and v_n is

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

a sink. The goal of this exercise is to find the number of paths from v_1 to v_n.

(a) Prove that there exists a topological sorting of G that has v_1 as first and v_n as last vertex.

Using part (a), we assume from now on that the sorting $v_1, v_2, \ldots, v_n$ of the vertices is a topological sorting. We can achieve this by renaming the vertices. Part (a) tells us then that we do not need to rename v_1 and v_n.

(b) Prove that for any directed v_1-v_n-path $P : v_1 = v_{i_0}, v_{i_1}, \ldots, v_{i_\ell} = v_n$ we have $i_0 < i_1 < \cdots < i_\ell$.

(c) 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 number of v_1-v_n paths 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?
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 $DP[i] =$ number of paths in G from v_i to v_n.*

(d)* What happens if the vertices v_1 and v_n are not a source respectively a sink? Can we still find the number of v_1-v_n paths using a similar approach as above?

Exercise 9.5 *Strongly connected vertices* **(1 point)**.

Let $G = (V, E)$ be a directed graph with n vertices and m edges. We say two distinct vertices $v, w \in V$ are *strongly connected* if there exists both a directed path from v to w, and from w to v.

Describe an algorithm which finds a pair $v, w \in V$ of strongly connected vertices in G, or decides that no such pair exists. The runtime of your algorithm should be at most $O(n + m)$. You are provided with the number of vertices n, and the adjacency list Adj of G.

Hint: *Use DFS as a subroutine.*

Exercise 9.2 *Short statements about graphs (cont'd)* **(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.

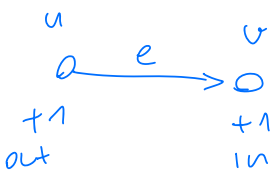
(a) If for every vertex $v \in V$ its in-degree $\deg_{\text{in}}(v)$ is even, then $|E|$ is even.

Solution:

This statement is true.
The following equality holds $\sum_{v \in V} \deg_{\text{in}}(v) = |E|$ since on both sides every edge is counted exactly once. Hence, if all terms on the left side are even, then also $|E|$ is even.

further we have $\sum_{v \in V} \deg_{in}(v) = \sum_{v \in V} \deg_{out}(v) = |E|$.

This is because every edge e adds $+1$ to out degree of one vertex and $+1$ to in degree of another vertex.



(b) For a longest directed path $P : v_0, \ldots, v_\ell$ in G, the endpoint has to be a sink.

Solution:

This statement is false.
Consider the graph with three vertices

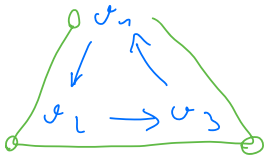
$$V = \{v_1, v_2, v_3\}$$

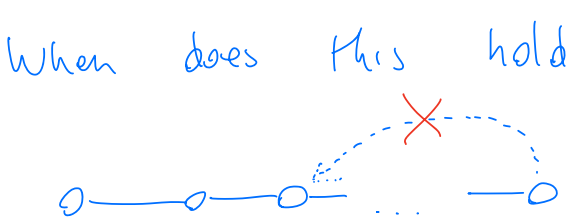
and the following edges

$$E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}.$$

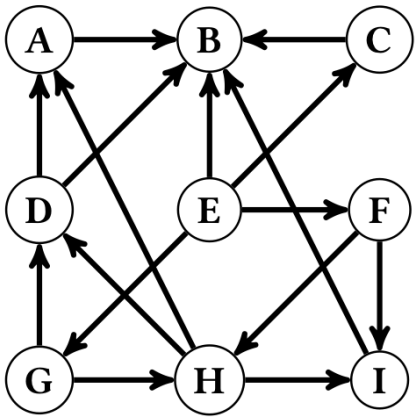
A longest directed path is for example v_1, v_2, v_3. However, v_3 is not a sink since it has the outgoing edge (v_3, v_1).

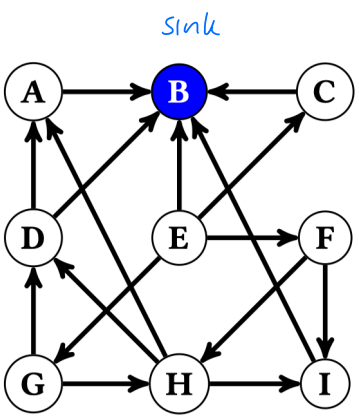
Remark: If the graph is assumed to have no directed cycles, i.e. we have a directed acyclic graph, then the statement holds as was used and proven in the lecture.

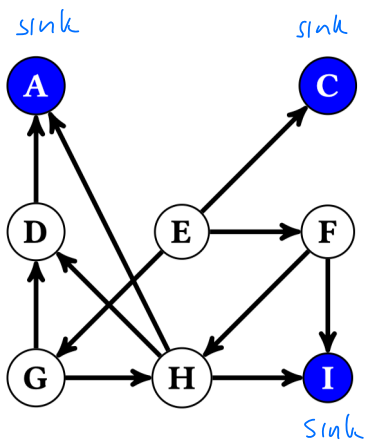


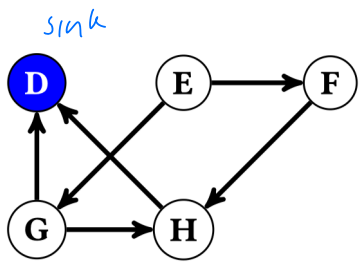


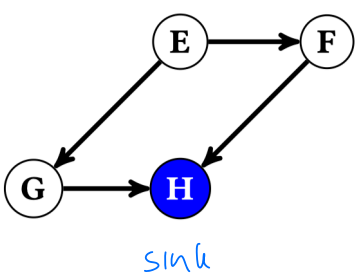
(c) The following graph has a topological sorting. If so, give a topological sorting; if not, prove why no topological sorting can exist.

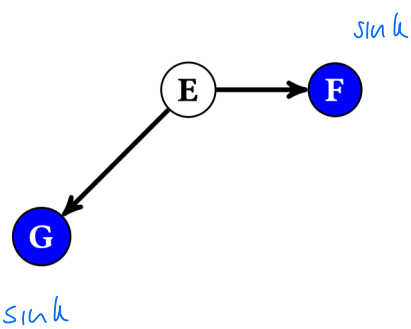












⟹ (E, F, G, H, D, A, C, I, B)

Exercise 9.4 *Number of paths in DAGs* **(1 point)**.

Let $G = (V, E)$ be a directed graph without directed cycles[1] (i.e., a directed acyclic graph or short DAG). Assume that $V = \{v_1, \ldots, v_n\}$ (for $n = |V| \in \mathbb{N}$). Further assume that v_1 is a source and v_n is a sink. The goal of this exercise is to find the number of paths from v_1 to v_n.

(a) Prove that there exists a topological sorting of G that has v_1 as first and v_n as last vertex.

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

Solution:

Define the graph G' as the graph G without v_1 and v_n (and all incident edges). This graph is still acyclic, so we know from the lecture that it has a topological sorting. Adding v_1 in the beginning of this sorting and v_n in the end, we get a topological sorting of G. This is indeed a topological sorting since all edges involving v_1 are of the form (v_1, v_i) for some i (v_1 is a source), all edges involving v_n are of the form (v_i, v_n) for some i (v_n is a sink) and we started with a topological sorting of G'.

Using part (a), we assume from now on that the sorting $v_1, v_2, \ldots, v_n$ of the vertices is a topological sorting. We can achieve this by renaming the vertices. Part (a) tells us then that we do not need to rename v_1 and v_n.

(b) Prove that for any directed v_1-v_n-path $P : v_1 = v_{i_0}, v_{i_1}, \ldots, v_{i_\ell} = v_n$ we have $i_0 < i_1 < \cdots < i_\ell$.

In a topological sorting of a graph, for any edge (v, w), we have that v comes before w in the sorting. Since $v_1, v_2, \ldots, v_n$ is a topological sorting of G we thus get that for any edge (v_i, v_j) we have $i < j$. In particular, if we have a directed v_1-v_n-path $P : v_1 = v_{i_0}, v_{i_1}, \ldots, v_{i_\ell} = v_n$, then $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots (v_{i_{\ell-1}}, v_{i_\ell})$ are edges of G and thus $i_0 < i_1 < \cdots < i_\ell$.

(c) 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 number of v_1-v_n paths 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, $Adj[u] := \{v \in V \mid (u, v) \in E\}$.

Hint: *Define the entry of the DP table as $DP[i] =$ number of paths in G from v_i to v_n.*

Solution:

1. *Dimensions of the DP table*: $DP[1 \ldots n]$
2. *Subproblems*: $DP[i]$ is the number of paths in G from v_i to v_n.
3. *Recursion*: We initialise $DP[n] = 1$. DP can then be computed recursively as follows for $i < n$

$$DP[i] = \sum_{j \in \{i+1,\ldots,n\}:\ (v_i, v_j) \in E} DP[j].$$

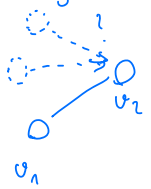
follows from (b)

The reason why this holds is that every path from v_i to v_n is of the following form: We first have an edge (v_i, v_j) and then a path from v_j to v_n (which might be of length 0). By part (b), we have that $j > i$, which is the reason why we only consider $j \in \{i + 1, \ldots, n\}$. Thus, to get the number of paths from v_i to v_n we can sum the number of v_j-v_n paths for all out-neighbors v_j of v_i (which satisfy $j > i$ since $v_1, v_2, \ldots, v_n$ is a topological sorting). Note that if we have the edge (v_i, v_j), then no v_j-v_n path contains v_i as the graph G is acyclic. Hence, combining an edge (v_i, v_j) with a path from v_j to v_n gives always a path (and not just a walk). This shows that our recurrence relation is correct.

reverse topo.-order

4. *Calculation order*: We can compute the entries by order of decreasing i. Then, for computing $DP[i]$ all entries $DP[j]$ we need have already been computed.
5. *Extracting the solution*: The solution can be found in $DP[1]$, by definition of the DP table.
6. *Running time*: Computing the ith entry of DP uses time $O(\deg_{\text{out}}(v_i) + 1)$ (the "1" is for accessing the first element of the list). Summing this over all vertices, we get that the total running time is $O(|V| + |E|)$ as wanted (using that $\sum_{i=1}^{n} \deg_{\text{out}}(v_i) = |E|$).

Why would topo.-order not work?

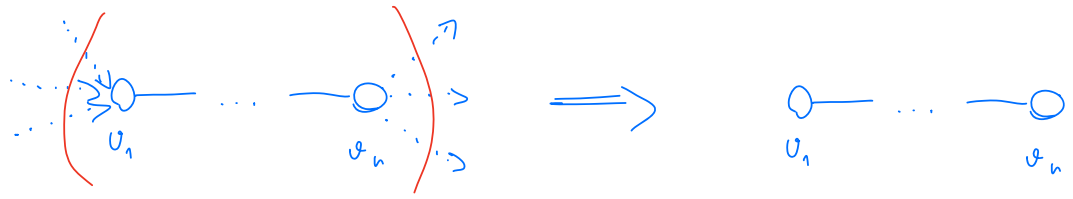


Remark: Note that we could have also defined the following DP table: $DP'[i] =$ number of paths in G from v_1 to v_i. One can initialise $DP'[1] = 1$ and find a similar recurrence relation as above for $DP'[i]$, where we sum over all in-neighbors of v_i. Using this, one can find the number of v_1-v_n paths by computing all entries in increasing order and returning $DP'[n]$. The problem with this approach is that finding all in-neighbors needs time $O(|V| + |E|)$ since we need to go over all adjacency lists as we are only given a list with the out-neighbors for every vertex. Thus, the run time of this solution would be $O(|V| \cdot (|V| + |E|))$ instead of the wanted $O(|V| + |E|)$.

(d)* What happens if the vertices v_1 and v_n are not a source respectively a sink? Can we still find the number of v_1-v_n paths using a similar approach as above?

Solution:

We note that any v_1-v_n path will not use an incoming edge at v_1 nor an outgoing edge at v_n (otherwise v_1 respectively v_n would occur twice which cannot happen in a path). Hence, given any directed acylic graph G, we can delete all incoming edges at v_1 and all outgoing edges at v_n. In this new graph G' v_1 is a source and v_n is a sink. Also, G' is still acyclic and the number of v_1-v_n paths in G' is equal to the number of v_1-v_n paths in G. Hence, to compute the number of v_1-v_n paths in G, we can equivalently compute the number of v_1-v_n paths in G'. This can be done using parts (a)–(c). Note that deleting all incoming edges at v_1 and all outgoing edges at v_n can be done in time $O(|V|+|E|)$, so the overall running time of the algorithm is still $O(|V|+|E|)$. Hence, we can find the number of v_1-v_n in any directed acyclic graph G in time $O(|V|+|E|)$.



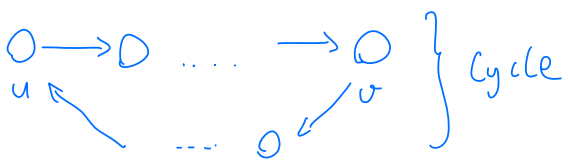
paths doesn't change, since we cannot use the incoming edges to get out of v_1 and using them after going out of v_1 would mean taking v_1 twice (contradiction to path). Similar reasoning for v_n.

Exercise 9.5 *Strongly connected vertices* **(1 point).**

Let $G = (V, E)$ be a directed graph with n vertices and m edges. We say two distinct vertices $v, w \in V$ are *strongly connected* if there exists both a directed path from v to w, and from w to v.

Describe an algorithm which finds a pair $v, w \in V$ of strongly connected vertices in G, or decides that no such pair exists. The runtime of your algorithm should be at most $O(n + m)$. You are provided with the number of vertices n, and the adjacency list Adj of G.

Hint: *Use DFS as a subroutine.*



Solution:

We use the following algorithm.

Algorithm 1

```
Input: integer n. Adjacency list Adj[1...n].

Let status[1...n] be a global array, with all entries initialized to UNVISITED.

function visit(u)
    status[u] ← VISITING
    for each v in Adj[u] do                    ▷ Iterate over all neighbours v.
        if status[v] = VISITING then           ▷ There is a directed cycle containing u and v.
            Output (u, v) and terminate
        if status[v] = UNVISITED then
            visit(v)
    status[u] ← VISITED.
for u = 1, 2, ..., n do
    if status[u] = UNVISITED then
        visit[u]
Output "no strongly connected vertices exist"
```

The algorithm above uses DFS to determine if there is a directed cycle in G. As we traverse the graph at most once, its runtime is at most $O(n + m)$.

Note that at any point during the algorithm there is a directed path from any vertex v with $status[v] =$ VISITING to the current vertex u. Therefore, if u has a neighbour v with $status[v] =$ VISITING, there must be a directed cycle containing both u and v. But that means u and v are strongly connected.

The algorithm only terminates if a directed cycle is found, or when all vertices have status VISITED. In the latter case, no directed cycle exists in the graph.