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 $G_1 = (V_1, E_1), \dots, G_k = (V_k, E_k)$ 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.

**Solution:**

We first show that disjoint unions of complete graphs are transitive ($\Leftarrow$), and then that any transitive graph is a disjoint union of complete graphs ($\Rightarrow$).

$\Leftarrow$: Let $G = (V, E)$ be a disjoint union of complete graphs $G_1 = (V_1, E_1), \dots, G_k = (V_k, E_k)$. Let $\{u, v\}, \{v, w\} \in E$. Since $G$ is a disjoint union, there exists $i \in \{1..k\}$ such that $v \in V_i$, $\{u, v\} \in E_i$, and $\{v, w\} \in E_i$. Since $E_i \subseteq V_i \times V_i$, we get $u \in V_i$ and $w \in V_i$. From the assumption that $G_i$ is complete, we finally get $\{u, w\} \in E_i \subseteq E$.

$\Rightarrow$: Let $G = (V, E)$ be a transitive graph. We can decompose $G$ into its connected components $G_1 = (V_1, E_1), \dots, G_k = (V_k, E_k)$. Clearly, $G$ is the disjoint union of its connected components. Let us now show that each connected component is complete. Let $i \in \{1..k\}$. Consider $u, v \in V_i$ with $u \neq v$. As $u$ and $v$ are in the same connected component of $G$, there exists a path $u = w_1 \to w_2 \to \cdots \to w_p = v$ from $u$ to $v$ in $G_i$.

We will now show by induction on $j \in \{2, \dots, p\}$: $P(j)$: "the edge $\{u, w_j\}$ is in $E_i$."

**Base case:** $j = 2$. As $u = w_1, \dots, w_p$ forms a path in $G_i$, we have $\{u, w_2\} = \{w_1, w_2\} \in E_i$, which immediately yields $P(2)$.

**Induction step:** Let $j \in \{2, \dots, p - 1\}$ such that $P(j)$ holds. This means that we have an edge $\{u, w_j\} \in E_i$. Now, as $w_1, \dots, w_p$ is a path in $G_i$, we also have an edge $\{w_j, w_{j+1}\} \in E_i$. Using the transitive property of $G$, we obtain $\{u, w_{j+1}\} \in E$, and since $G$ is a disjoint union, we also have $\{u, w_{j+1}\} \in E_i$. This shows $P(j + 1)$.

**Conclusion:** $P(j)$ holds for all $j \in \{2, \dots, p\}$.

For $j = p$, we obtain $P(p)$: "the edge $\{u, w_p\}$ is in $E_i$". As $w_p = v$, we get $\{u, v\} \in E_i$. Hence $G_i$ is complete, which concludes the proof.

**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.

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

**Solution:**

This statement is false.
Consider the graph with three vertices

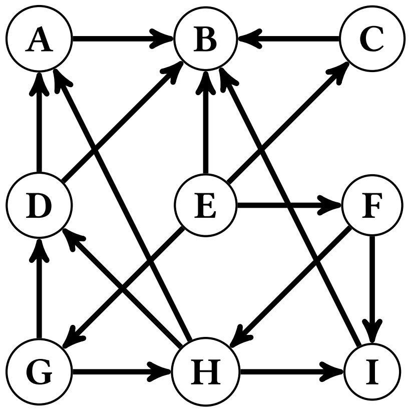
$$V = \{v_1, v_2, v_3\}$$

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.*

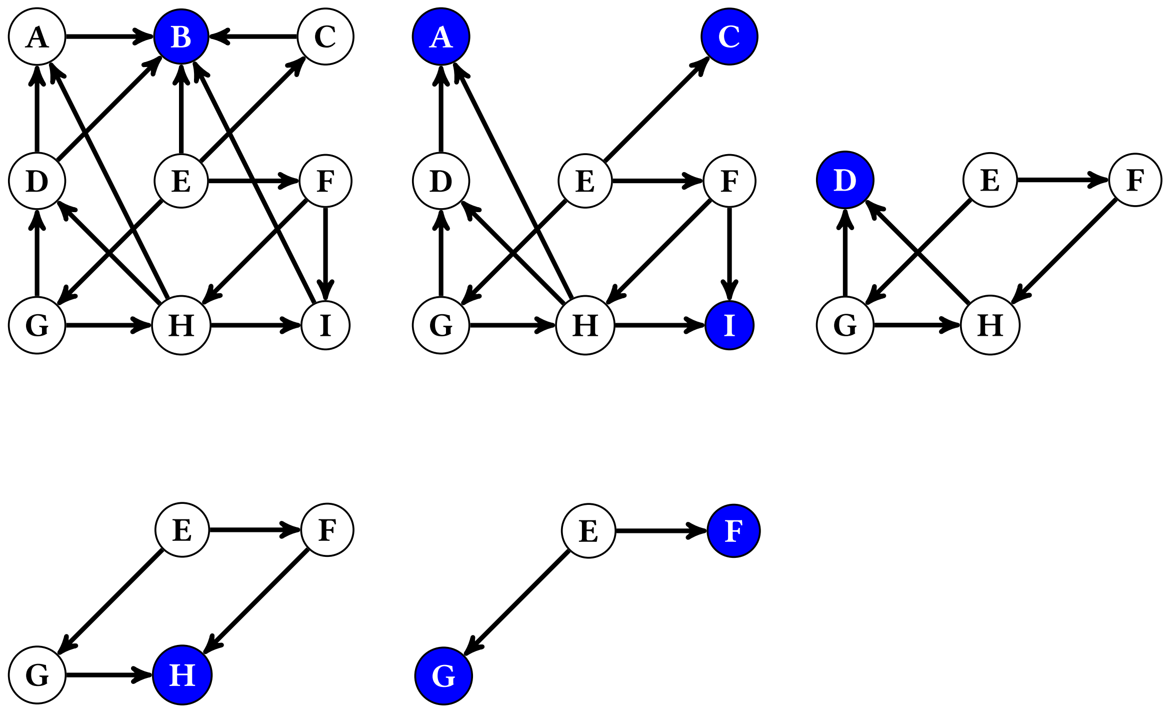
(c) The following graph has a topological sorting. If so, give a topological sorting; if not, prove why no topological sorting can exist.



**Solution:**

This statement is true.
We construct a topological sorting with the strategy that was discussed in the lecture, that is, we find a sink $v$ in the graph, remove $v$ (and all incident edges) from the graph and iterate. Since the

graph is small, we find a sink "by hand", but we could as well do it with the algorithm that was discussed in the lecture. All graphs that occur in the process are described below, where the current sinks are marked in blue. Note that there are different possible topological sortings, the one described here is just an example.
In the first graph the vertex "B" is a sink, so "B" appears last in our topological sorting.
Now, there are several choices, all the vertices "A", "C" and "I" are sinks. We can add all these three vertices into our sorting, so a possible current end of the sorting is (A,C,I,B).
After removing these vertices, the only sink is "D", leading to the current end of the sorting (D,A,C,I,B).
In the next graph, only "H" is a sink, leading to (H,D,A,C,I,B).
In the last graph, both "F" and "G" are sinks. After removing these two vertices only the vertex "E" remains, so we get for example the following topological sorting (E,F,G,H,D,A,C,I,B). One can verify in the original graph that this is indeed a topological sorting. This is however not strictly necessary since we know from the lecture that the algorithm we executed is correct.



**Guidelines for correction:**

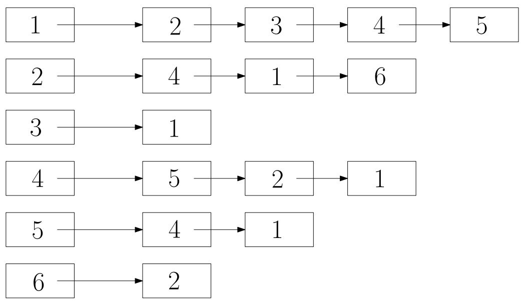
For awarding the bonus points, each subexercise should be split into two parts, namely one part is giving the correct answer and the other part is giving a correct proof or counterexample. If at least 3 parts are solved correctly, $1/2$ points should be awarded. If all 6 parts are solved correctly, $1$ point should be awarded.

**Exercise 9.3** *Data structures for graphs.*

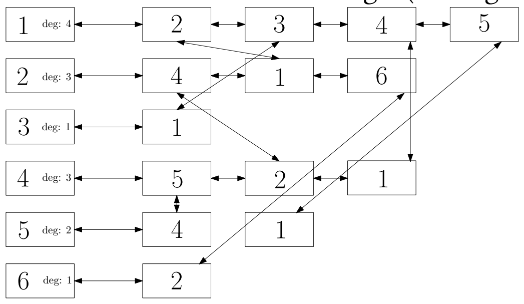
Consider three types of data structures for storing an undirected graph $G$ with $n$ vertices and $m$ edges:

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)?

**Solution:**

$\Theta(n^2)$ for adjacency matrix, $\Theta(n+m)$ for adjacency list and improved adjacency list.

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 $\deg(v)$.

**Solution:**

$\Theta(n)$ in adjacency matrix, $\Theta(1+\deg(v))$ in adjacency list, $\Theta(1)$ in improved adjacency list.

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

**Solution:**

$\Theta(n)$ in adjacency matrix, $\Theta(1)$ in adjacency list and in improved adjacency list.

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

**Solution:**

$\Theta(1)$ in adjacency matrix, $\Theta(1+\min\{\deg(v), \deg(u)\})$ in adjacency list and in improved adjacency list.

(d) Input: Two adjacent vertices $u, v \in V$. Delete the edge $e = \{u, v\}$ from the graph.

**Solution:**

$\Theta(1)$ in adjacency matrix, $\Theta(\deg(v) + \deg(u))$ in adjacency list and $\Theta(\min\{\deg(v), \deg(u)\})$ in improved adjacency list.

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

**Solution:**

$\Theta(n)$ in the adjacency matrix ($\Theta(n)$ for finding a neighbor and $\Theta(1)$ for the edge deletion).

$\Theta(1+\max_{w:\{u,w\}\in E} \deg(w))$ for the adjacency list ($\Theta(1)$ for finding a neighbor and $\Theta(\max_{w:\{u,w\}\in E} \deg(w))$ for the edge deletion).

$\Theta(1)$ for the improved adjacency list ($\Theta(1)$ for finding a neighbor and $\Theta(1)$ for the edge deletion).

(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.

**Solution:**

$\Theta(1)$ in adjacency matrix, $\Theta(1+\min\{\deg(v), \deg(u)\})$ in adjacency list and in improved adjacency list.

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

**Solution:**

$\Theta(n^2)$ in adjacency matrix, $\Theta(n+m)$ in adjacency list and $\Theta(n)$ in improved adjacency list.

For the last two queries, describe your algorithm.

**Solution:**

Query (f): We check whether the edge $\{u, v\}$ does not exist. In adjacency matrix this information is directly stored in the $u$-$v$-entry. For adjacency lists we iterate over the neighbors of $u$ and the neighbors of $v$ in parallel and stop either when one of the lists is traversed or when we find $v$ among the neighbors of $u$ or when we find $u$ among the neighbors of $v$. If we didn't find this edge, we add it: in the adjacency matrix we just fill two entries with ones, in the adjacency lists we add nodes to two lists that correspond to $u$ and $v$. In the improved adjacency lists, we also need to set pointers between those two nodes, and we need to increase the degree for $u$ and $v$ by one.

Query (g): In the adjacency matrix we copy the complete matrix, but leave out the row and column that correspond to $v$. This takes time $\Theta(n^2)$. There is an alternative solution if we are allowed to *rename* vertices: In this case we can just rename the vertex $n$ as $v$, and copy the $n$-th row and column into the $v$-th row and column. Then the $(n-1) \times (n-1)$ submatrix of the first $n-1$ rows and columns will be the new adjacency matrix. Then the runtime is $\Theta(n)$. Whether it is allowed to rename vertices depends on the context. For example, this is not possible if other programs use the same graph.

In the adjacency lists we remove $v$ from every list of neighbors of every vertex (it takes time $\Theta(n+m)$) and then we remove a list that corresponds to $v$ from the array of lists (it takes time $\Theta(n)$). In the improved adjacency lists we iterate over the neighbors of $v$ and for every neighbor $u$ we remove $v$ from the list of neighbors of $u$ (notice that for each $u$ we can do it in $\Theta(1)$ since we have a pointer between two occurences of $\{u, v\}$) and decrease $\deg(u)$ by one. Then we remove the list that corresponds to $v$ from the array of lists (it takes time $\Theta(n)$).

## 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 $V = \{v_1, \dots, v_n\}$ (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 $v_n$.

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

**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 $v_1, v_2, \dots, v_n$ 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 $P : v_1 = v_{i_0}, v_{i_1}, \dots, v_{i_\ell} = v_n$ we have $i_0 < i_1 < \dots < i_\ell$.

**Solution:**

In a topological sorting of a graph, for any edge $(v, w)$, we have that $v$ comes before $w$ in the sorting. Since $v_1, v_2, \dots, v_n$ 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 $P : v_1 = v_{i_0}, v_{i_1}, \dots, v_{i_\ell} = v_n$, then $(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \dots (v_{i_{\ell-1}}, v_{i_\ell})$ are edges of $G$ and thus $i_0 < i_1 < \dots < i_\ell$.

(c) Describe a bottom-up dynamic programming algorithm that, given a graph $G$ with the property that $v_1, \dots, v_n$ 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* $v_n$.

**Solution:**

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

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

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 $j \in \{i+1, \dots, n\}$. 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 $v_1, v_2, \dots, v_n$ 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.

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 $i$th 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 $\sum_{i=1}^{n} \deg_{\text{out}}(v_i) = |E|$).

*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 initialize $DP'[1] = 1$ 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 $DP'[n]$. When implementing this recursion as the one above (i.e. when computing $DP'[i]$ we sum over all in-neighbors), the run time of this solution would be $O(|V| \cdot (|V| + |E|))$ instead of the wanted $O(|V| + |E|)$. The reason for this is that finding all in-neighbors of a given vertex 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. However, it is also possible to implement this recursion in linear time.*
*Instead of computing the sum for $DP'[i]$ when considering $i$, we can do the following: Whenever we are at position $j$, we add $DP'[j]$ to $DP'[i]$ for all out-neighbors $i$ of $j$. The values $DP'[j]$ are initialized as 0 (except $DP'[1]$, which is still set to 1). Then once we arrive at $j$, $DP'[j]$ is correct. One can show this inductively since all in-neighbors of $j$ come before $j$. Hence, when we arrive at $DP'[n]$, the value in $DP'[n]$ is also the number of paths from $v_1$ to $v_n$.*
*Another option is to compute all the in-neighbours in advance and save them in adjacency lists, which can be done in time $O(|V| + |E|)$. Then the update step for $DP'[i]$ can be done by summing over the in-neighbours as described above.*

(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 $O(|V| + |E|)$.

**Guidelines for correction:**

$1/2$ point should be awarded for correctly solving parts (a) and (b). $1/2$ point should be awarded for correctly solving part (c).

**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 $\mathrm{Adj}$ of $G$.

***Hint:*** *Use DFS as a subroutine.*

**Solution:**

**Solution using the edge classification of DFS:**
We want to use depth-first search and the edge-classification we get from it. We use the following algorithm: We run DFS and if we encounter a back edge, we return the endpoints of it. If we never encounter a back edge, then we output that the graph $G$ has no pair of strongly connected vertices.

The runtime of the execution of DFS is $O(n + m)$. Also, identifying back edges does not increase the runtime. One option to identify the back edges is to go over all edges after the DFS and check the pre- and post-numbers of the endpoints (recall that $(u, v)$ being a back edge is defined as $\mathrm{pre}(v) < \mathrm{pre}(u) < \mathrm{post}(u) < \mathrm{post}(v)$), which takes time $O(n + m)$. Alternatively, we can do it directly during the DFS: When processing the neighbors of $u$, an edge $(u, v)$ is a back edge exactly when $v$ is already marked but has not yet been assigned a post-number, i.e. "visit($v$)" is not yet finished. Checking this does not increase the asymptotic runtime of the DFS. Thus, the runtime of the proposed algorithm is indeed $O(n + m)$.

Regarding correctness, note that if we find a back edge $(u, v)$, then $\{u, v\}$ is strongly connected since there is a path from $v$ to $u$ using tree edges ($u$ is processed during "visit($v$)") and we have the edge $(u, v)$, which gives a path from $u$ to $v$. Also note that if we do not find a back edge, then we know from the lecture that the graph has a topological ordering. But the existence of a topological ordering implies that there cannot be a pair of strongly connected vertices. Indeed, let $u, v \in V$. Assume that $u$ comes before $v$ in the ordering (otherwise replace the roles of $u$ and $v$ in following argument). All edges in $G$ go from vertices that come earlier in the ordering to vertices that come later. Thus, we cannot have a path from $v$ to $u$ since such a path would need to have at least one edge that goes from some vertex to a vertex that comes earlier in the ordering. Hence, in both cases (we find a back edge or we do not), the algorithm is correct, which finishes the proof of correctness for the proposed algorithm.

**Direct solution giving an explicit algorithm:**
We use the following algorithm.

**Algorithm 1**

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

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

function $visit(u)$
    $status[u] \leftarrow$ 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
            $visit(v)$
    $status[u] \leftarrow$ VISITED.
for $u = 1, 2, \ldots, n$ 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.

**Guidelines for correction:**

- 1/2 point should be awarded for the correct idea (i.e., finding a directed cycle using DFS, and using the ‘back edge’ to extract a strongly connected pair).
- 1/2 point should be awarded for a correct algorithm and explanation of correctness/runtime.