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.

The solutions are intended to help you understand how to solve the exercises and are thus more detailed than what would be expected at the exam. All parts that contain explanation that you would not need to include in an exam are in grey.

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 $V = V_1 \cup \dots \cup V_k$, $E = E_1 \cup \dots \cup E_k$, and $V_1, \dots, V_k$ are pairwise disjoint.

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

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 $i \in [k]$ such that $v \in V_i$, $\{u, v\} \in E_i$, and $\{v, w\} \in E_i$. Then 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 $i \in [k]$. 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 $(w_1, w_2, \dots, w_p)$ from u to v in G_i, where $w_1 = u$ and $w_p = v$.

We will now show that $\{w_1, w_j\} \in E_i$ for $j = 2, 3, \dots, p$ by induction on j.

Base case: $j = 2$. As $(w_1, w_2, \dots, w_p)$ forms a path in G_i, we have $\{w_1, w_2\} \in E_i$ by the definition of paths.

Induction step: Assume $\{w_1, w_j\} \in E_i$ for some $j \in \{2, 3, \dots, p-1\}$. As $(w_1, w_2, \dots, w_p)$ forms a path in G_i, we have $\{w_j, w_{j+1}\} \in E_i$ by the definition of paths. As both $\{w_1, w_j\} \in E_i$ and

$\{w_j, w_{j+1}\} \in E_i$ and $E_i \subseteq E$, using the transitive property of G, we obtain $\{w_1, w_{j+1}\} \in E$. Since G is a disjoint union, we also have $\{w_1, w_{j+1}\} \in E_i$.

Conclusion: $\{w_1, w_j\} \in E_i$ for $j = 2, 3, \ldots, p$. This implies $\{w_1, w_p\} \in E_i$, i.e. $\{u, v\} \in E_i$.

Therefore, for every pair of distinct vertices $u, v \in V_i$, we can show $\{u, v\} \in E_i$, which means $G_i = (V_i, E_i)$ is complete.

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).

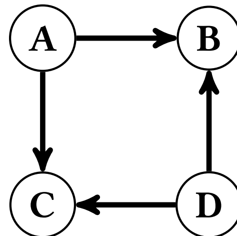
Solution:

This statement is true.
Consider the graph $\overleftarrow{G}$ obtained from G by flipping all edges. That is, $\overleftarrow{G} = (V, \overleftarrow{E})$, with $\overleftarrow{E} := \{(w, v) : (v, w) \in E\}$. Note that in-degrees in $\overleftarrow{G}$ correspond to out-degrees in G and vice versa. Therefore, $\overleftarrow{G}$ has no sinks. From the lecture, we thus know that $\overleftarrow{G}$ must have a directed cycle $(v_1, v_2, \ldots, v_k, v_1)$. But then the inverse $(v_1, v_k, \ldots, v_2, v_1)$ of this cycle is a directed cycle in G.

(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).

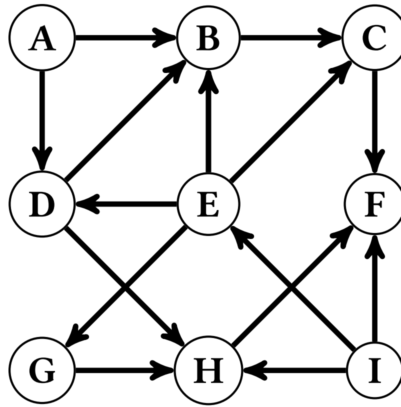
Solution:

This is false.
For a counterexample, consider the following directed graph:



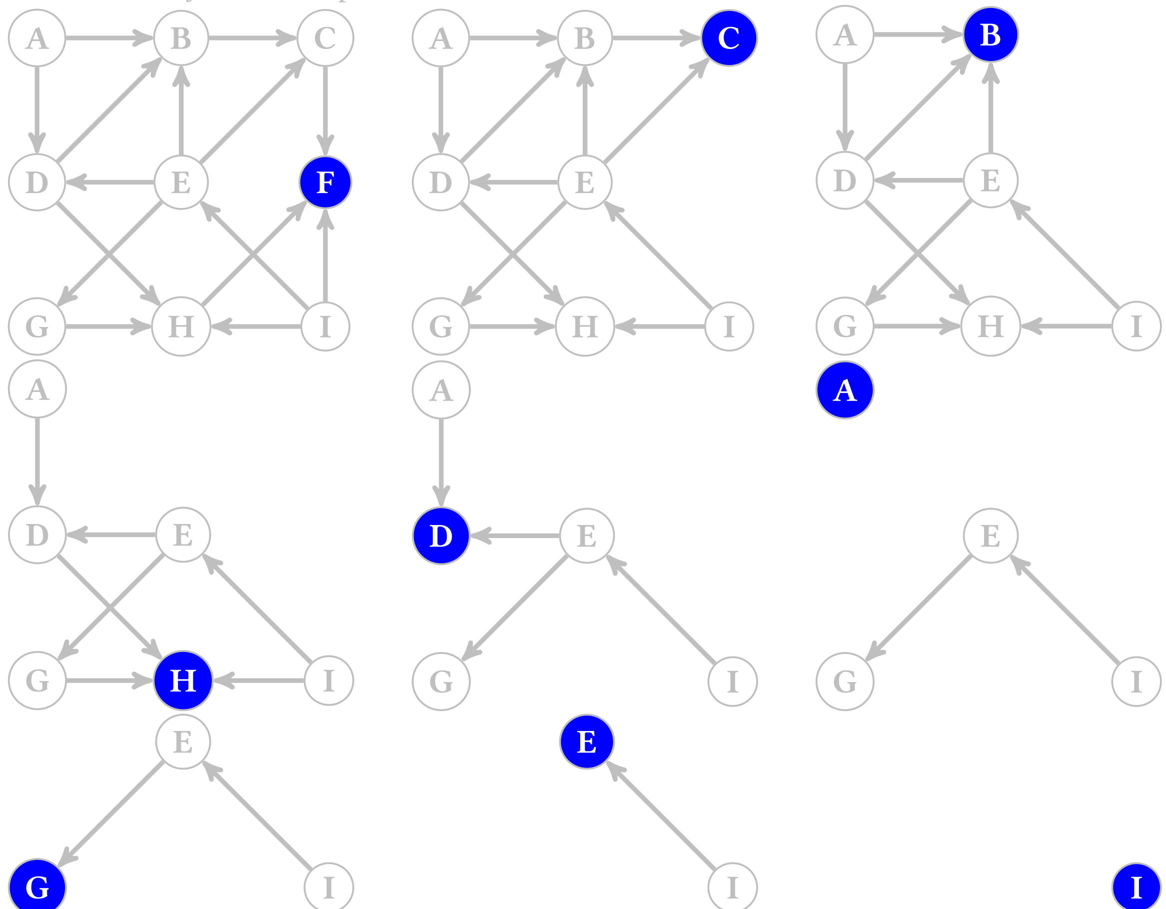
(c) The following graph has a topological sorting. If so, give a topological sorting; if not, prove why no topological sorting can exist.

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



Solution:

This statement is true. $I, E, G, A, D, H, B, C, F$ is a topological sorting of G.
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.



So, $I, E, G, A, D, H, B, C, F$ is a topological sorting of G.

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.

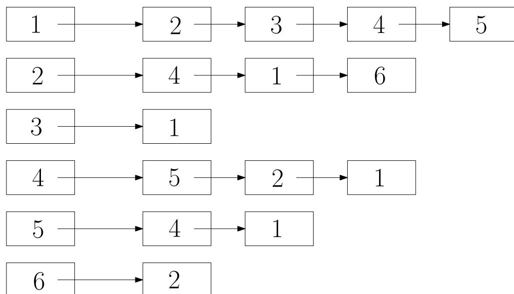
Exercise 9.3 *Data structures for graphs.*

Consider 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 $m = 7$)

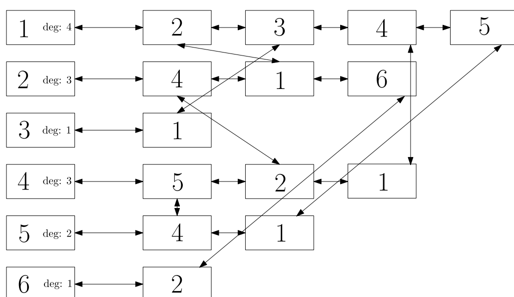
1) Adjacency matrix:

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?

Solution:

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

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 $\deg(v)$.

Solution:

$\Theta(n)$ in adjacency matrix, $\Theta(1+\deg(v))$ in adjacency list, $\Theta(1)$ in improved adjacency list. The reason why we write $\Theta(1+\deg(v))$ instead of $\Theta(\deg(v))$ is that when $\deg(v)=0$, we still need $\Theta(1)$ time.

(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. For (improved) adjacency list, we can search u in $\mathrm{Adj}[v]$ and v in $\mathrm{Adj}[u]$ simultaneously.

(d) Input: An edge $\{u, v\} \in E$. Delete the edge $\{u, v\}$ 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. Say $\deg(u) \leq \deg(v)$. For improved adjacency list, we can locate v in $\mathrm{Adj}[u]$ in $\Theta(\deg(u))$ time. Then we can use the pointer to delete u in $\mathrm{Adj}[v]$ in $\Theta(1)$ time.

(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 all edges incident to v from the graph.

Solution:

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

Question: 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 vertices to two lists that correspond to u and v. In the improved adjacency lists, we also need to set pointers between those two vertices, and we need to increase the degree for u and v by one.

Query (g): In the adjacency matrix, we fill every entry in v-th row and column with zero. In the adjacency lists we remove v from every list of its neighbors, and then we remove all elements in the list $\text{Adj}[v]$ except the head. In the improved adjacency lists we iterate over the neighbors of v and for every neighbor u we remove v from $\text{Adj}[u]$ (notice that for each u we can do it in $\Theta(1)$ time since we have a pointer between two occurrences of $\{u, v\}$) and decrease $\deg(u)$ by one. Then we remove all elements in the list $\text{Adj}[v]$ except the head and set $\deg(v)$ to zero.

What if we also want to delete the vertex v itself? The answer will depend on the details of how we want to manage the space/memory. For example, in the adjacency matrix we can 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.

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 $i_1 < i_2 < \cdots < i_k$.

Solution:

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

(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.*

Solution:

1. *Dimensions of the DP table*: $DP[1 \dots n]$
2. *Subproblems*: $DP[i]$ is the length of the longest path ending at v_i.
3. *Recursion*: We initialize $DP[1] = 0$. DP can then be computed recursively as follows for $i > 1$

$$DP[i] = \begin{cases} 0 & \text{if } v_i \text{ is a source} \\ 1 + \max_{j \in \{1,\dots,i-1\}\colon (v_j,v_i) \in E} DP[j] & \text{otherwise} \end{cases}$$

   To see why this holds, first notice that if v_i is a source, then only the trivial path $P = (v_i)$ ends at v_i. Hence, the longest path ending at a source has length 0. Otherwise, v_i has in-degree at least one.

   Let $k = DP[i]$. Let $P = (v_{j_0}, \dots, v_{j_k})$ be a longest path ending at v_i (i.e. $v_{j_k} = v_i$). By part (a), we have that $j_0 < \dots < j_k$ meaning that the j_{k-1} is already computed in the DP array so we know that what the length of the longest path ending at $v_{j_{k-1}}$ is. Removing v_{j_k} from the end of P gives us a path $(v_{j_0}, \dots, v_{j_{k-1}})$ ending at $v_{j_{k-1}}$. We claim $(v_{j_0}, \dots, v_{j_{k-1}})$ is a longest path ending at $v_{j_{k-1}}$ and $DP[j_{k-1}] = \max_{j \in \{1,\dots,i-1\}\colon (v_j,v_i) \in E} DP[j]$. The reason is, if our claim were false, then adding back v_i to this path would contradict the fact that P is a longest path ending at v_i. Thus $DP[j_{k-1}] = k - 1$ and then

$$1 + \max_{j \in \{1,\dots,i-1\}\colon (v_j,v_i) \in E} DP[j] = 1 + DP[j_{k-1}] = 1 + (k-1) = k\,.$$

   Therefore $DP[i] = 1 + \max_{j \in \{1,\dots,i-1\}\colon (v_j,v_i) \in E} DP[j]$.
4. *Calculation order*: For $i = 1, 2, \dots, n$, compute $DP[i]$.
5. *Extracting the solution*: The solution can be found as $\max_{i \in [n]} DP[i]$. The longest path in the graph must end at some vertex.
6. *Running time*: Computing the ith entry of DP uses time $O(\deg_{\text{in}}(v_i) + 1)$, where the 1 comes from the extra operations. To compute $DP[i]$, we need to access $DP[j]$ for all in-neighbors v_j of v_i. Since the given adjacency list consists of out-neighbors for each vertex, before filling in the DP table we need to compute a reversed adjacency list which consists of in-neighbors for each vertex. This process takes $O(|V| + |E|)$ time. Summing this over all vertices, we get that the total running time is $O(|V| + |E|)$ as wanted (using that $\sum_{i=1}^{n} \deg_{\text{in}}(v_i) = |E|$).

There is an alternative solution where we define $DP[i]$ to be the length of the longest path starting (rather than ending) at v_i. Then to compute $DP[i]$ we only need to access the out-neighbors (rather than in-neighbors) of v_i. So with this definition of the DP entries, we do not need to compute a reversed adjacency list in the beginning.

Guidelines for correction:

If (b) is incorrect award 0 points. If either (a) is correct and (b) is partially correct, or (a) is incorrect and (b) is correct award $1/2$ point. Award 1 point if both parts are correct.

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

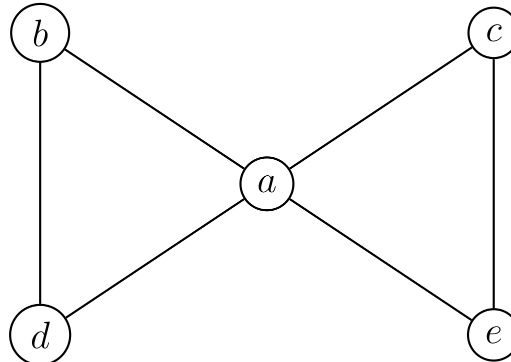
---

Run DFS on G starting from a vertex $v_1 \in V$ ▷ First step
$(v_1, v_2, \ldots, v_n) \leftarrow$ DFS pre-order

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



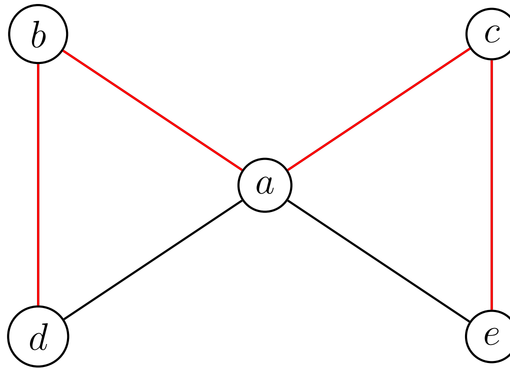
Execute the algorithm on this graph using a as the starting vertex and lexicographic ordering for the DFS subroutine. This means that if we start at vertex b for example, then the DFS subroutine prioritizes going to a first.

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.

Solution:

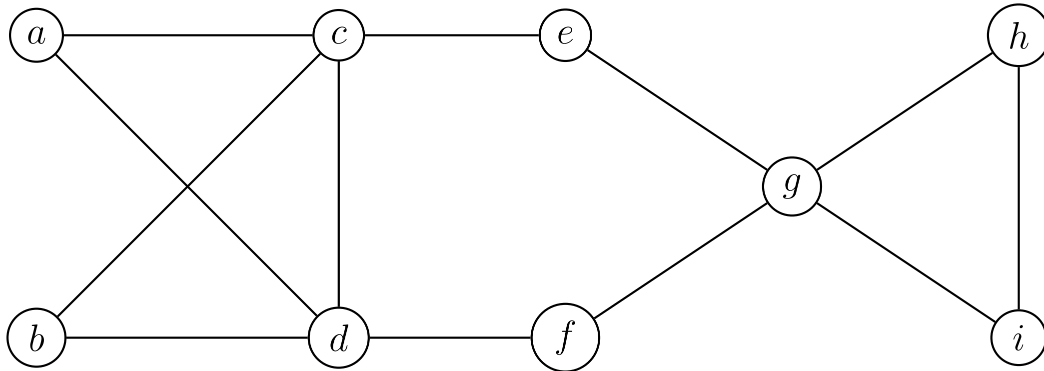
The DFS tree will look like



the pre-order is (a, b, d, c, e) and the walk is (a, e, c, a, d, b, a).

(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.

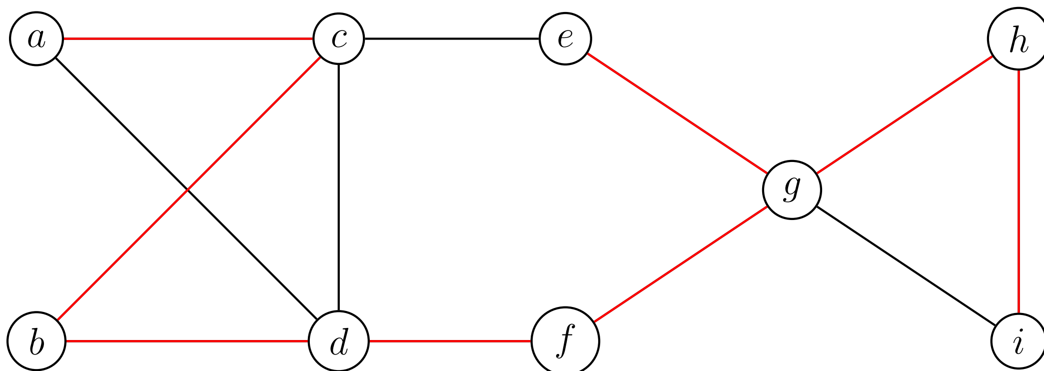
Note that this pre-order can be generated by any possible DFS tree, meaning the lexicographic ordering of the vertices should not be taken into account. The labeling of the vertices are purely to help with writing out the answers.



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.

Solution:

One solution is to start at a, with pre-order $(a, c, b, d, f, g, h, i, e)$, the DFS tree will look like



and the walk is $(a, d, f, g, e, c, d, b, c, a)$, missing out on edges (g, h), (g, i) and (h, i) thus making it not a closed Eulerian walk.

Guidelines for correction:

(a) is worth $1/2$ point. (b) is worth $1/2$ point.