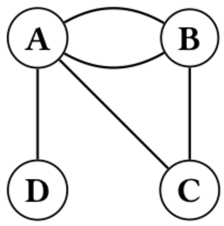
Exercise 10.1 *Eulerian tours in multigraphs* **(1 point).**

A *multigraph* $G = (V, E)$ is a graph which is permitted to have multiple copies of the same edge. That is, the edges E form a *multiset* (a set in which elements are allowed to occur multiple times). For example, the multigraph with $V = \{1, 2, 3, 4\}$ and $E = \{\{A, B\}, \{A, B\}, \{A, D\}, \{B, C\}, \{A, C\}\}$ is depicted below. To avoid confusion, the term *simple graph* is sometimes used to indicate that duplicate edges are not allowed.



(a) An Eulerian tour in a multigraph is a tour which visits every edge exactly once. If multiple copies of an edge exist, the tour should visit each of them exactly once. Given a multigraph $G = (V, E)$, describe an algorithm which constructs a *simple* graph $G' = (V', E')$ such that G has a Eulerian tour if and only if G' has a Eulerian tour. The new graph should satisfy $|V'| \leq |V| + |E|$, and $|E'| \leq 2 \cdot |E|$. The runtime of your algorithm should be at most $O(n + m)$. You are provided with the number of vertices n and an adjacency list of G (if there are multiple edges between $v, w \in V$, then w appears that many times in the list of neighbours of v).

We want an algorithm A sum that

$$G=(V,E) \xrightarrow{A} G'=(V',E')$$

where G is a multigraph and G' is a simple graph and A fulfills:

G has Eulerian tour iff

G' has Eulerian tour.

Furthermore, we demand $|E'| \leq 2 \cdot |E|$, $|V'| \leq |V| + |E|$

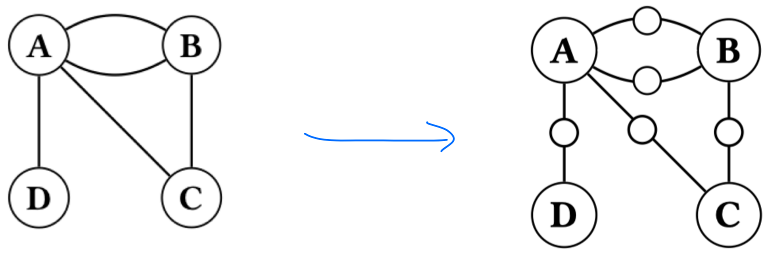
and A in $O(n+m)$.

Notice that the constraints of $|E'|$ and $|V'|$ tell us a lot about G'.

Let $E = \{e_1, \dots, e_{|E|}\}$, we write $e_k = \{v_k, w_k\}$.

For V' we first add all vertices from V and another vertex v'_k for every edge e_k, $1 \leq k \leq |E|$.

$$E' = \bigcup_{k=1}^{|E|} \{e_k^1, e_k^2\}, \text{ where } e_k^1 = \{v_k, v_k'\} \text{ and } e_k^2 = \{w_k, v_k'\}.$$



$G = (V, E)$ $\qquad$ $G' = (V', E')$

Clearly G' fulfills $|V'| \leq |V| + |E|$ and $|E'| \leq 2|E|$ and $O(n+m)$.

It remains to show G has Eulerian tour iff G' has Eulerian tour.

($\Rightarrow$) Assume that G has Eulerian tour $T = (e_{j_0}, e_{j_1}, \dots, e_{j_{|E|}})$.

By replacing each e_{j_k} with $\underbrace{e^1_{j_k}, e^2_{j_k}}$ order important? YES!

we obtain Eulerian tour in G'.

($\Leftarrow$) Assume that G' has Eulerian tour. In every Eulerian tour T' in G' e^1_k and e^2_k must appear $\underbrace{\text{directly adjacent}}$ in T'.

True or false? True! Order? Either order!

because they are the only edges connecting to v'_k. Then we obtain T by replacing e^1_k, e^2_k with e_k.

(b)* Let $G = (V, E)$ be a *simple* graph, and let $f : E \to \mathbb{N} \cup \{0\}$ be a function. A Eulerian f-tour of G is a tour which visits each edge $e \in E$ exactly $f(e)$ times. Describe an algorithm which constructs a simple graph $G' = (V', E')$ such that G has a Eulerian f-tour if and only if G' has a Eulerian tour. The new graph should satisfy $|V'| \leq |V| + \sum_{e \in E} f(e)$, and $|E'| \leq 2 \sum_{e \in E} f(e)$. The runtime of your algorithm should be at most $O(n + m + \sum_{e \in E} f(e))$.

Solution:

To construct G', first, we remove all edges e from G with $f(e) = 0$. Then, we construct a multigraph $H = (V, F)$, where F contains exactly $f(e)$ copies of each edge in G. Note that $|F| = \sum_{e \in E} f(e)$. Note also that, by definition, an Eulerian tour exists in H if and only if a Eulerian f-tour exists in G. Finally, we use part (a) to convert H into a simple graph $G' = (V', E')$, where we know that $|V'| \leq |V| + |F| = |V| + \sum_{e \in E} f(e)$ and $|E'| \leq 2 \cdot |F| = 2 \cdot \sum_{e \in E} f(e)$.

Exercise 10.4 *Strongly connected components* **(1 point)**.

Let $G = (V, E)$ be a directed graph with n vertices and m edges. Recall from Exercise 9.5 that two distinct vertices $v, w \in V$ are *strongly connected* if there exist both a directed path from v to w, and from w to v.

The vertices of G can be partitioned into disjoint subsets $V_1, V_2, \ldots, V_k \subseteq V$ with $V = V_1 \cup V_2 \cup \ldots \cup V_k$, such that any two distinct vertices $v, w \in V$ are strongly connected if and only if they are in the same subset V_ℓ, for some $1 \leq \ell \leq k$. The subsets V_ℓ are called the *strongly connected components* of G.

As in Exercise 9.5, you are provided with the number of vertices n, and the adjacency list Adj of G.

(a) Describe an algorithm that outputs the strongly connected components of G in time $O(n \cdot (n + m))$.

Hint: *Apply the algorithm of Exercise 9.5 several times. After each application, remove a vertex from G.*

The binary relation of being strongly connected is an equivalence relation (reflexive, symmetric, transitive). The equivalence classes form strongly connected components.

Solution:

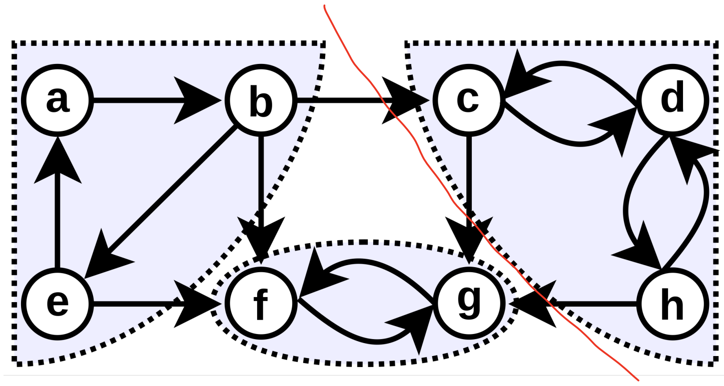
For each $v \in V$, create a list $L_v = [v]$. We iteratively apply the following procedure:

(i) Apply the algorithm of Exercise 9.5 to find two strongly connected vertices, say v, w in G. If no such vertices exist, stop and output L_v for each vertex v that is still in G.

(ii) Set $L_v \leftarrow L_v \cup L_w$.

(iii) For every in-neighbor x of w (except possibly v) add an edge (x, v) to G. For every out-neighbor y of w (except possibly v) add an edge (v, y) to G. Then remove w from G.

For the runtime of the algorithm, note that in each iteration, one vertex is removed from G, and so there can be at most n iterations. Each iteration can be executed in time $O(n + m)$, leading to total runtime $O(n \cdot (n + m))$.



Correctness:

(I) for any v in G, all vertices in L_v are strongly connected. (step (i), (ii))

(II) (iii) does not change strong connectivity.

(I) + (II) $\Longrightarrow$ we conclude after termination we have L_v of any remaining v of G contains the strongly connected component of v.

```
KOSARAJU(G, n)
    mark all v ∈ V as unvisited
    L ← ∅        // empty list

    for unvisited v ∈ V
        DFS_1(v, L)

    mark all v ∈ V as unvisited
    reverse L
    C ← ∅        // list of components
    create G^T = (V, E^T) from G

    for unvisited v ∈ L    // in order of L
        T ← ∅        // single component
        DFS_2(v, T)
        add T to C

    return C     // contains all components
```

```
DFS_1(v, L)
    mark v as visited
    for unvisited u ∈ N_G(v)   // neighbors of v in G
        DFS_1(u)
    add v to back of L

DFS_2(v, T)
    mark v as visited
    add v to component T
    for unvisited u ∈ N_{G^T}(v)   // neighbors of v in G^T
        DFS_2(u)
```