Exercise 1

Consider a directed multigraph with a set of vertices $V = \{A, B, C, D, E, F\}$ and a set of edges $E = \{0, 1, 2, 3, 4, 5, 6\}$. The functions src and tgt are given by following table:

e	src	tgt
0	A	B
1	B	B
2	A	C
3	C	E
4	E	D
5	D	C
6	C	D

a) Consider node C. What are its immediate predecessors and its immediate successors? What is its indegree and outdegree?

b) Is node E reachable from node A?

c) Draw a subgraph of this graph.

d) List all simple cycles of this graph.

e) Is the graph strongly connected?

f) List the strongly-connected components of this graph.

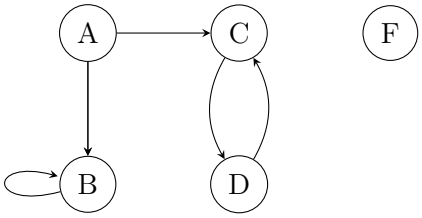
g) Write down the adjacency matrix of this graph.

Solution:

a) Immediate predecessors: A, D. Immediate successors: E, D. Indegree: 2. Outdegree: 2

b) Yes, via the path $A \to C \to E$.

c) For example, without node E:



d) Since we have no parallel edges, we can simply write paths as a list of vertices without any ambiguity. The simple cycles of this graph, written first as lists of edges and then as lists of vertices are:

- (1) or (B, B)
- $(5, 6)$ or (D, C, D)
- $(6, 5)$ or (C, D, C)
- $(3, 4, 5)$ or (C, E, D, C)

- $(4, 5, 3)$ or (E, D, C, E)
- $(5, 3, 4)$ or (D, C, E, D)

Note that e.g. $(3, 4, 5)$ and $(4, 5, 3)$ are different paths since the start at different locations. Thus, technically, we really do have to list both of them.

e) For this graph to be strongly connected, all nodes must be reachable from all other nodes. This is not the case in this graph for a few reasons: node F is not reachable from any other nodes, nor does it reach any other nodes, node A is not reachable from any other nodes, B is only reachable from A (and itself – a node is always reachable from itself).

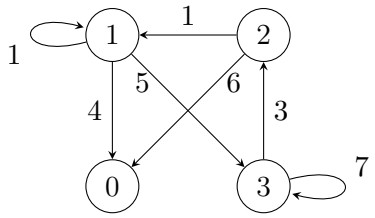
f) The strongly-connected components are: $\{B\}, \{A\}, \{F\}, \{C, D, E\}$

g) Beginning with A for the first row and column and descending alphabetically, we get the following adjacency matrix:

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Exercise 2

Consider the following weighted digraph:



a) Use Floyd's algorithm to determine the shortest paths of the above graph.

b) What is the shorted path between nodes 1 and 2? Between 0 and 3?

c) Replace the weight 5 between nodes 1 and 3 with -3, and run Floyd's algorithm again. What do you observe?

d) Replace the weight 5 between nodes 1 and 3 with -5, and run Floyd's algorithm again. What do you observe?

e) Under what conditions does the algorithm produce correct results if negative edge weights are present?

f) Can you think of a way to modify Floyd's algorithm so that it always produces sensible results even for negative edge weights?

Solution:

a) Floyd's algorithm finds the total weight of the shortest path between nodes on a graph, represented as a matrix. Its iterations on this graph are as follows:

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ 4 & 0 & \infty & 5 \\ 6 & 1 & 0 & \infty \\ \infty & \infty & 3 & 0 \end{pmatrix}$$

→ after iterating through $r = 0$ (no change):

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ 4 & 0 & \infty & 5 \\ 6 & 1 & 0 & \infty \\ \infty & \infty & 3 & 0 \end{pmatrix}$$

→ after iterating through $r = 1$:

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ 4 & 0 & \infty & 5 \\ 5 & 1 & 0 & 6 \\ \infty & \infty & 3 & 0 \end{pmatrix}$$

→ after iterating through $r = 2$:

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ 4 & 0 & \infty & 5 \\ 5 & 1 & 0 & 6 \\ 8 & 4 & 3 & 0 \end{pmatrix}$$

→ after iterating through $r = 3$:

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ 4 & 0 & 8 & 5 \\ 5 & 1 & 0 & 6 \\ 8 & 4 & 3 & 0 \end{pmatrix}$$

b) The shortest path between 1 and 2 is $(1, 3, 2)$ with a length of 8. There is no path between 0 and 3.

c) We get the following matrix:

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ 4 & 0 & 0 & -3 \\ 5 & 1 & 0 & -2 \\ 8 & 4 & 3 & 0 \end{pmatrix}$$

Nothing changed except that

- The shortest path between 1 and 3 is still $(1, 3)$ but now has length -3.
- The shortest path between 1 and 2 is still $(1, 3, 2)$ but now has length 0.
- The shortest path between 2 and 3 is now $(1, 3)$ and has length -2.

The results are correct.

d) We get the following matrix:

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ 3 & -1 & -2 & -6 \\ 4 & 0 & -1 & -5 \\ 7 & 3 & 2 & -2 \end{pmatrix}$$

This is now no longer the matrix of minimal distances because whenever we start at Node 1, 2, or 3, we can repeat the cycle $1 \to 2 \to 3 \to 1$ as often as we want to make the path as short as we want. Thus, the matrix of node distances should perhaps rather look like this:

$$\begin{pmatrix} 0 & \infty & \infty & \infty \\ -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty \\ -\infty & -\infty & -\infty & -\infty \end{pmatrix}$$

Here, a $-\infty$ in row i and column j means that there exist arbitrarily short paths from i to j.

e) The problem in d) is that there exist cycles of negative length. As long as this is *not* the case, the algorithm still works: shortest paths can be assumed to be simple (in the sense that if we have a shortest path that *isn't* simple, we can find another shortest path of the same length that is) and all the proofs from the lecture still work.

However, if there *is* a cycle of negative length, then some pairs of nodes in the graph do not even have a shortest path between them and a lot of the assumptions of Floyd's algorithm break down. It particular, the algorithm works upon the basis that there is no need to visit a node more than once.

f) If there are negative cycles, they can be detected by checking for negative entries on the diagonal of the result matrix (let's call it M) after running the algorithm.

To make a meaningful result, we can iterate through all nodes k that lie on a negative cycle (i.e. with $M_{kk} < 0$) and set $M_{ij} := -\infty$ for any nodes i, j such that k is reachable from i and j is reachable from k, i.e. where $M_{ik} \neq \infty$ and $M_{kj} \neq \infty$.

When implemented carefully, this can be done in $O(n^2)$ time.

Exercise 3

A digraph $G = (V, E)$ is called an *arborescence* if it has a node $v_0 \in V$ (called the *root*) such that for every node $v \in V$ there exists exactly one path from v_0 to v.

a) Draw a few arborescences.

b) Show that arborescences are acyclic.

c) What do the strongly-connected components of an arborescence look like?

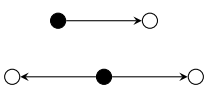
d) Show that the root of an arborescence has indegree 0.

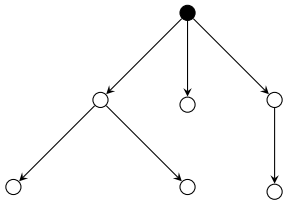
e) Show that every non-root node of an arborescence has indegree 1.

f) Prove that in an arborescence, $|V| = |E| + 1$. **Hint:** find a bijection between E and $V \setminus \{v_0\}$!

Solution:

a) The simplest arborescence is $(\{u\}, \emptyset)$, i.e. just a single node with no edges. Three more are shown below. In each one, the root is highlighted in black. The nodes are not given names in the drawings since the names do not matter for whether or not the graph is an arborescence.





We can see that arborescences have a tree-like branching structure: if we follow any of the edges from the root, we arrive at a 'child node' of the root, and the nodes reachable from that child node again form an arborescence. All these 'sub-arborescences' that 'branch out' from the root are disjoint.

b) Suppose we had a cycle $p = (v, \ldots, v)$. By definition, there exists exactly one path q from v_0 to v. But qp is also a path from v_0 to v and $qp \neq q$ because cycles are non-empty. This is a contradiction.

c) Because arborescences are acyclic, reachability is a partial order (in particular: antisymmetric). Thus each node is its own strongly-connected component.

d) If v_0 had non-zero indegree, it would have an immediate predecessor v, i.e. $v \to v_0$. By definition of an arborescence, there exists a path from v_0 to v. But then appending edge $v \to v_0$ would give us a cycle.

e) First of all, every non-root node w must have at least one incoming edge because there is (clearly non-empty) path from v_0 to w, and the last edge of such a path must have w as a target.

Next, let $u \to w$ and $v \to w$ be two different edges leading to w. There is a path p from v_0 to u and a path q from v_0 to v. Appending the edge $u \to w$ to p and the edge $v \to w$ to q, we get two different paths from v_0 to w. This is a contradiction.

f) Following the hint, we define the following bijection:

$$f : E \to V \setminus \{v_0\},\ f(e) = \mathrm{tgt}(e)$$

f is well-defined because there is no edge whose target is v_0. It is a bijection because every node in $V \setminus \{v_0\}$ has indegree 1, there is exactly one edge that has it as a target.

It then follows that the domain and codomain of f have the same cardinality, i.e. $|E| = |V \setminus \{v_0\}| = |V| - 1$.

Bonus exercise

Implement Floyd's algorithm for a directed *graph* (not multigraph) $G = (V, E)$ with non-negative integer weights w in your favourite programming language. You may assume that $V = \{1, \ldots, n\}$.

How do you represent the graph and the distances? How do you represent the result of the algorithm? Test your algorithm on the graph from Exercise 2 and use it to check your results.

Feel free to use the Haskell template `Floyd.hs` from the materials section on OLAT, which already defines one possible representation of graphs, distances, and distance matrices. You can however also easily implement something like this in Java, C, or C++.

Solution: An excerpt of the Haskell solution is printed below. The `.hs` file can also be found in the solutions section on OLAT. A Java solution is also provided.

```
minimum' :: [Int] -> Dist
minimum' xs = if null xs then Infinity else Finite (minimum xs)
```

```
floyd :: WeightedGraph -> DistMatrix
floyd graph = go 1 initMatrix
  where n = nNodes graph

        initDist i j
          | i == j    = 0
          | otherwise = minimum' [w | (u, w, v) <- edges graph, u == i, v == j]
        initMatrix = mkMatrix n initDist

        go r matrix
          | r > n     = matrix
          | otherwise =
              go (r + 1) $
                mkMatrix n (\i j ->
                  min (matrix ! (i, j)) (matrix ! (i, r) + matrix ! (r, j)))
```