Problem 8.1. Topological sorting and connected components

Consider the following graph $G = (V, E)$.

1. Provide a subset $E' \subset E$ of edges that one can remove such that the resulting graph $G' = (V, E \setminus E')$ can be sorted topologically. The set $E'$ should be as small as possible. Motivate your choice.

2. Provide a topological sorting of the graph $G'$.

3. For $n \in \{1, 2, 3, 4\}$, construct a graph with $n$ vertices that can be sorted topologically that has the maximum number of edges. Based on this, provide a conjecture on the maximum number of edges a topologically sortable graph on $n$ vertices may have, and prove this conjecture.

4. What is the minimum number of edges in an undirected graph with $n$ vertices and $k$ connected components? Describe how components with a minimum number of edges look like.

Submission link: https://codeboard.ethz.ch/daex08t01

Solution of Problem 8.1.

1. The graph contains two 3-element cycles (on \{B, C, D\} and \{B, E, D\}). We need to remove an edge from each of these to make the graph sortable. The edge (B, D) is contained in both of them, and indeed after removing the edge, the resulting graph can be sorted topologically.

   \[ E' = \{(B, D)\}. \]

2. A, B, C, E, D

3. No loops and no parallel edges may occur.

<table>
<thead>
<tr>
<th>$n$</th>
<th>number of edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
</tbody>
</table>

Since for any two vertices there can at most be one edge, the total number of edges is at most $\binom{n}{2}$. This bound is tight as for any complete undirected graph (which has $\binom{n}{2}$ edges), we can direct the edges such that they respect an arbitrary order of the vertices. A similar but inductive argument is provided with the topological sort algorithm from the lectures where iteratively nodes without in-edges are removed from the graph.
4. Two vertices are in the same connected component if and only if there is a path between these two vertices. If the number of edges in a connected component is minimal, there must not be any cycles, as otherwise one could remove an edge from the cycle and the component would remain connected. Hence, a connected component in an undirected graph is cycle-free and connected, that is, a tree. A tree on \( s \) vertices has exactly \( s - 1 \) edges. If a graph on \( n \) vertices has \( k \) connected components with vertex sets \( V_1, \ldots, V_k \), then the graph has at least

\[
\sum_{i=1}^{k} (|V_i| - 1) = \left( \sum_{i=1}^{k} |V_i| \right) - k = n - k.
\]

**Problem 8.2. Depth-first-search and breadth-first-search**

Both depth-first-search (DFS) and breadth-first-search (BFS) admit a certain degree of freedom: one may choose the order in which the neighbors of a vertex are considered. For the graph shown below, let us assume that both DFS and BFS visit the neighbors of a vertex in alphabetical order.

1. Give the DFS and BFS ordering of the graph (i.e., the sequence in which the vertices are visited by DFS and BFS, respectively), starting from vertex \( A \).

2. Is there a starting vertex in this graph from which the DFS ordering is the same as the BFS ordering?

3. For an arbitrary integer \( n \), describe an undirected connected graph with \( n \) vertices for which the DFS ordering matches the BFS ordering.

4. Which is the asymptotic running time of DFS and BFS if the graph is provided as an adjacency matrix and not as an adjacency list? Justify your answer.

**Submission link:** [https://codeboard.ethz.ch/daex08t02](https://codeboard.ethz.ch/daex08t02)

**Solution of Problem 8.2.**

1. DFS: \( A, B, C, D, E, F, H, G \)
   
   BFS: \( A, B, F, C, H, D, G, E \)

2. No, the orders are different for all starting vertices.

3. One example is a star graph, that is, a graph with a center vertex \( c \), and for every other vertex \( v \) there is exactly one edge \((c, v)\). If the starting vertex is \( c \), then clearly BFS and DFS produce the same order. Also, if the starting vertex is not \( c \), then the second vertex that is visited in both DFS and BFS is \( c \); then, all other vertices are also visited in the same order.
4. Both DFS and BFS consider all neighbors of a vertex. If the graph is given with adjacency lists, the number of steps for each vertex is of the order of its neighbors. In total, both searches need \( O(|V| + |E|) \), which is in \( O(|E|) \) for connected graphs.

With adjacency matrices, we basically need to check for every vertex pair whether there is an edge between them. (For each vertex, we need to inspect every entry in its row.) Thus, the total number of steps is in \( \Omega(|V|^2) \).

If the graph is “sparse” (for example if \( |E| \in O(|V|) \)), then the asymptotic running time is worse when using an adjacency matrix.

**Problem 8.3. Programming exercise: Closeness centrality**

This exercise is about calculating the *closeness centrality* of vertices in an undirected graph \( G = (V, E) \). This is a measure of how far a vertex is from other vertices in a graph. For a vertex \( v \), the closeness centrality \( C(v) \) of \( v \) is given by

\[
C(v) = \sum_{u \in V \setminus \{v\}} d(v, u),
\]

where \( d(v, u) \) is the length of the shortest path between \( v \) and \( u \). We define \( d(v, u) = 0 \) if there is no path between \( v \) and \( u \). If \( C(v) \) is small, the vertex \( v \) is considered “central” in its connected components. (Note that the closeness centrality is usually defined to be the inverse of the above sum.)

In this exercise, we analyze a graph that stems from collaborations on scientific papers. The vertices of the graph are the co-authors of the mathematician Paul Erdős. (There are 511 such co-authors.) There is an edge between two vertices if the corresponding authors have jointly published a paper. See [https://oakland.edu/enp/thedata/](https://oakland.edu/enp/thedata/).

We provide a framework that reads the data from this file and transforms it into an adjacency matrix. For each vertex (indexed from 0 to 510), the name of the co-author is provided in a vector. Your task is to compute the closeness centrality of each co-author. Print the values in the order in which the co-authors appear in the input file (i.e., in order of their indices).

Implement the Floyd-Warshall algorithm to obtain the lengths of the shortest path between any two vertices, and then use the above definition to compute and output the closeness centrality.

**Submission link:** [https://codeboard.ethz.ch/daex08t03](https://codeboard.ethz.ch/daex08t03)

**Solution of Problem 8.3.**

```c++
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include <cassert>

using namespace std;

template<typename Matrix>
void allPairsShortestPaths(unsigned n, Matrix& m)
{
    for(unsigned k = 0; k < n; ++k)
    {
        for(unsigned i = 0; i < n; ++i)
        {
            for(unsigned j = i + 1; j < n; ++j)
            {
                if(k == i || k == j)
                    continue;
```
```c++
if (m[i][k] == 0 || m[k][j] == 0)
    continue;
if (m[i][j] == 0 || m[i][k] + m[k][j] < m[i][j])
    m[i][j] = m[j][i] = m[i][k] + m[k][j];
}
}
}
}

int main()
{
    unsigned n = 0;
    ifstream input("../input.txt");
    char buffer[41];
    buffer[40] = '\0';

    for(unsigned i = 0; i < 21; ++i)
    {
        input.ignore(numeric_limits<streamsize>::max(), '\n');
        input >> n;
        cout << "vertices: " << n << endl;
    }

    vector<vector<unsigned>> adjacencies(n, vector<unsigned>(n, 0));
    vector<string> names(n);

    for(unsigned i = 0; i < n; ++i)
    {
        unsigned index, degree, dummy;
        input >> index >> degree >> dummy;
        assert(i + 1 == index);
        while(input.peek() == ' ')
            input.get();
        input.read(buffer, 40);
        names[i] = buffer;
        for(unsigned j = 0; j < degree; ++j)
        {
            unsigned neighbor;
            input >> neighbor;
            adjacencies[i][neighbor - 1] = 1;
            adjacencies[neighbor - 1][i] = 1;
        }
    }

    allPairsShortestPaths(n, adjacencies);

    for(unsigned i = 0; i < n; ++i)
    {
        cout << names[i] << ": ";
        unsigned centrality = 0;
        for(unsigned j = 0; j < n; ++j)
        {
            if(j == i)
                continue;
            centrality += adjacencies[i][j];
        }
        cout << centrality << endl;
    }
```
return 0;
}