Task: graph algorithms

Implement the following graphing algorithms in C++ using the code contained in the graph.h file, add accordingly:

- breadth-first search
- Depth-first search including topological sorting
- Determination of strongly connected components
- Determination of minimum frameworks according to Prim
- Determination of shortest routes to Bellman-Ford and Dijkstra

A graph can be an object g of any type G in principle, which has at least the following member functions must possess:

• g.vertices() returns an object of any container type (eg std::list) that contains the nodes of the graph g (in any order).

Such a container type must have at least one parameterless member function size, which specifies the number of returns elements of the container (so g.vertices().size() returns the number of vertices of the graph g). It must also be possible to go through the elements of a container using "range-based for loops".

(so that the variable v in the statement for (V v : g.vertices()) ..... all vertices of the graph g passes through if they have type V).

All C++ Standard Library containers meet these requirements.

• For a node u of the graph g, g.successors(u) returns an object of such a container type, which contains the descendants of node u (in any order). (If u is not a node of the graph g the behavior of g.successors(u) may be undefined.)

• g.transpose() returns the transposed graph of g as a new object independent of g (i.e. the Graph g is not changed).

The type of the transposed graph can be different from the type G of the graph g, but must also have the have the member functions just described.

A weighted graph can also be an object of in principle any type, apart from those mentioned above Member functions has another member function weight such that g.weight(u, v) for a vertex u of the graph g and a successor v of u returns the weight of the edge from u to v as a double value.

(If u is not a vertex of the graph g or v is not a successor of u, then the behavior of g.weight(u, v) must be undefined.)

The default classes Graph and WeightedGraph meet the above requirements and can be used to generate test graphs as follows:

• For any node type V, in principle, an object of the type Graph<V> with the adjacency list representation of a graph can be initialized as a so-called initializer list in curly brackets: Each element This initializer list is a pair (also in curly brackets) consisting of a node of the type V and another initializer list with the descendants of this node, for example:

```
Graph<string> g = {          // Graph g with nodes of type string.
{ "A", { "B", "C" } },       // Node A has children B and C.
{ "B", { } },                // Node B has no descendants.
{ "C", { "C" } }             // Node C has itself as its successor.
};
```

For this graph g, g.vertices() returns a container with the elements "A", "B" and "C", g.successors("A") a container with the elements "B" and "C", g.successors("B") an empty one Container and g.successors("C") a container with element "C".

• Analogously, an object of the type WeightedGraph<V> can be initialized with the adjacency list representation of a graph extended by edge weights, in which the “inner” initializer lists instead of successors turn contain pairs in curly brackets, each consisting of a successor and the associated edge weight with type double, for example:

```
WeightedGraph<string> wg = {
{ "A", { { "B", 2 }, { "C", 3 } } },
{ "B", { } },
{ "C", { { "C", 4 } } }
};
```

For this graph wg the calls to vertices and successors return the same results as for the above graph g; wg.weight("A", "B") returns the double value 2, wg.weight("A", "C") returns the value 3 and wg.weight("C", "C") the value 4.

However, all algorithms must also work for other types G as long as they meet the above requirements.

Each algorithm is represented by a function template with type parameters V (node type) and G (type of the graph) implemented, which as a first parameter a graph g with type G and as a second parameter eventually gets a start node s of type V or a list vs of nodes. The last parameter that per reference is always a suitable data structure in which the results of the algorithm −− mostly in one or more tables (maps) −− stored.

For example, a typical usage looks like this:

```
// node type string.
using V = string;
// test graph.
Graph<V> g = {
{ "A", { "B", "C" } }, // node A has children B and C.
{ "B", { } }, // Node B has no children.
{ "C", { "C" } } // Node C has itself as its successor.
};

// Do depth-first search on g and store the result in res.
DFS<V> res;
dfs(g,res);
// The nodes v of the graph by ascending completion times
// iterate over and for each node its discovery and
// print completion time.
for (V v : res.seq) {
cout << v << " " << res.det[v] << " " << res.fin[v] << endl;
}
```

For algorithms that require a prio queue, use the File prioqueue.h.

Test your implementation carefully with different graphs and possibly different start nodes and detailed!

With Prim's algorithm, it should be noted that the respective test graph must be undirected, that means he must to each edge also contain the opposite edge with the same weight.