10th Recitation

Graphs Depth-First Search and Breadth-First Search

Instructions:

- Download the file `aed2324_p10.zip` from the course page and unzip it (it contains the `lib` folder, the `Tests` folder with the files `Graph.h`, `funWithBFS.h` , `funWithBFS.cpp`, `funWithDFS.h`, `funWithDFS.cpp` and `tests.cpp`, and the files `CmakeLists.txt` and `main.cpp`);
- In CLion, open a project by selecting the folder containing the files from the previous point;
- If you can't compile, perform "Reload CMake Project" on the `CMakeLists.txt` file;
- Implement it in the file `Graph.h`, `funWithBFS.cpp` and `funWithBFS.cpp`
- Note that all the tests are uncommented. They should fail when run for the first time (before implementation). As you solve the exercises, the respective tests should pass.

1. Graphs: Depth-First and Breadth-First Search

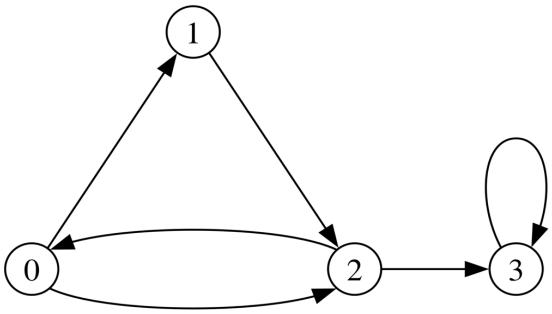
Consider the `Graph` class below, as defined in the `Graph.h` file:

```
template <class T> class Vertex {
    T info;
    vector<Edge<T> > adj;
public:
    //...
    friend class Graph<T>;
};

template <class T> class Edge {
    Vertex<T> * dest;
    double weight;
public:
    //...
    friend class Graph<T>;
    friend class Vertex<T>;
};
```

```
template <class T> class Graph {
    vector<Vertex<T> *> vertexSet;
    //...
public:
    //...
};
```

and the following graph as an example:



1.1.1) In the `Graph` class, implement the member function below:

```
vector<T> dfs() const
```

This function returns a vector containing the graph elements (content of the vertices) when a depth-first search traversal is performed on the graph by default starting on the first node, labeled 0. First, implement the auxiliary function below, which visits a vertex *v* and its adjacents, recursively, updating a parameter with the list of visited nodes:

```
void Graph<T>::dfsVisit(Vertex<T> *v, vector<T> &res) const
```

Execution example, traversing the graph depicted above:
output: *result* = `{0, 1, 2, 3}`

1.1.2) Using the same auxiliary function as before, implement the member-function below

```
vector<T> dfs(const T &source) const
```

which returns a vector containing the graph elements (content of the vertices) when a depth-first search is performed on the graph, starting at the vertex `source`.

Execution example, for the graph depicted above:

input: *source* = `0`
output: *result* = `{0, 1, 2, 3}`

input: *source* = `1`
output: *result* = `{1, 2, 0, 3}`

input: *source* = `2`
output: *result* = `{2, 0, 1, 3}`

input: *source* = `3`
output: *result* = `{3}`

1.2) In the `Graph` class, implement the member function below:

```
vector<T> bfs(const T &source) const
```

This function returns a vector containing the graph elements (content of the vertices) when a breadth-first search is performed on the graph, starting at the vertex `source`.

Execution example, for the graph depicted above:

input: *source* = `0`
output: *result* = `{0, 1, 2, 3}`

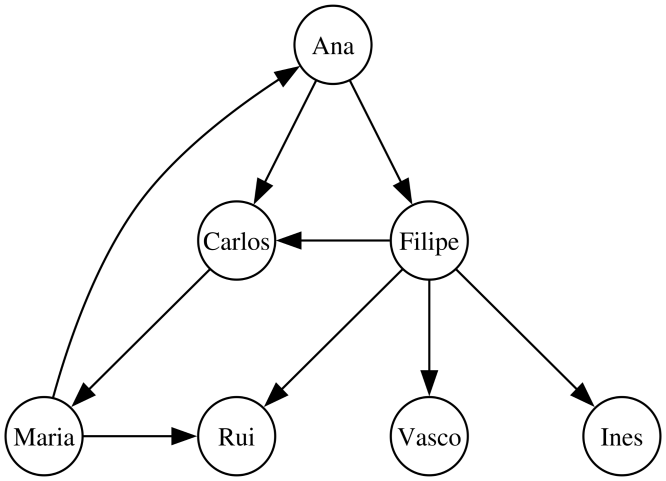
input: *source* = `1`
output: *result* = `{1, 2, 0, 3}`

input: *source* = `2`
output: *result* = `{2, 0, 3, 1}`

input: *source* = `3`
output: *result* = `{3}`

2. Practical applications of Depth-First and Breadth-First Algorithms

A social network can be represented by the Graph class, where the vertices represent individuals in the network and the edges represent a friendship relationship (the direction of the edge indicates who has sent the friend request).



In order to see how close an individual is to the rest of their friends, implement a method that allows you to obtain the friends of a person, or the friends of their friends, and so on (attending to the initiative of the friendship requests, indicated by the direction of the edge). Consider the notion of distance between two vertices, defined as the minimum number of edges in any path between them.

2.1.1) Using a depth-first search approach, implement the member function below, in the `funWithDFS.cpp` file:

```
vector<Person> nodesAtDistanceDFS(const Graph<Person> *g, const Person &source, int k);
```

which returns a vector of people (`vector<Person>`), with the individuals in the friendship group who are at distance of exactly `k` from the person `source`.

As before, implement the auxiliary function below, which visits a vertex *v* and its adjacents, recursively, updating a parameter with the list of visited nodes.

```
void nodesAtDistanceDFSVisit(const Graph<Person> *g, Vertex<Person> *v, int k, vector<Person> &res);
```

Execution example, for the graph depicted above:

input: *source* = `Ana`, *k* = `0`
output: *result* = `{Ana}`
Itself.

input: *source* = `Ana`, *k* = `1`
output: *result* = `{Carlos, Filipe}`
Friends of Ana

input: *source* = `Ana`, *k* = `2`
output: *result* = `{Ines, Maria, Rui, Vasco}`
Friends of Ana's friends.

2.1.2) Using a breadth-first search approach, implement the member function below, in the `funWithDFS.cpp` file:

```
vector<Person> nodesAtDistanceBFS(const Graph<Person> *g, const Person &source, int k);
```

Execution example, for the graph depicted above:

input: *source* = `Ana`, *k* = `0`
output: *result* = `{Ana}`
Itself.

input: *source* = `Ana`, *k* = `1`
output: *result* = `{Carlos, Filipe}`
Friends of Ana

input: *source* = `Ana`, *k* = `2`
output: *result* = `{Ines, Maria, Rui, Vasco}`
Friends of Ana's friends.

2.2) After learning of important news, an individual sends it out through his/her friendships. Determine which individual (vertex) disseminates the news first hand (as a novelty) to the largest number of recipients and what is the number of recipients reached.

In the Graph class, implement the member function below, in the `funWithBFS.cpp`:

```
int maxNewChildren(const Graph<Person> *g, const Person &source, Person &info)
```

This function returns the maximum number of children in a graph starting at vertex *v*; the *info* parameter is the content of that graph vertex.

Suggestion: adapt the breadth-first search algorithm.

Execution example, for the graph depicted above:

input: *source* = `Ana`
output: *result* = `3`; *info* = `Filipe`
Ana started spreading the news to her friends, but it was Filipe who spread it as a novelty to more people, 3 in total (Carlos already knew about it through Ana).

2.3) In the Graph class, implement the member function below, in the `funWithDFS.cpp` file:

```
bool isDAG(Graph<int> g);
```

This function checks whether the directed graph is acyclic (Directed Acyclic Graph, or DAG for short), that is, the graph does not contain cycles, in which case the function returns true. Otherwise, the function returns false.

Suggestion: Adapt the depth-first search algorithm.