Laboratory class 2 and 3 for students who want to practise C++ advanced programming

Task

Define class graph which defines directed unweighted graphs.
Class graph should be defined inside namespace graphs;
Graph nodes are denoted using integer numbers (any number including 0 or negative number is allowed as node description). Gaps in numbering are allowed (for instance nodes of 3-nodes graph may be denoted as: 0, -3, 5)

Use STL C++ library containers to implement graph class. You may use any container that you think is useful, but using any specialized library which defines graphs and computation on them are not allowed of course.

Space complexity of graph class should be linear to number of graphs edges.
It means that implementation using adjacency matrix isn't allowed. Implementation using adjacency lists is required (word 'list' is only part of name of graphs representation method, you may use other STL containers, not only list)

Define the following public methods of graph class

Instance methods:

```
bool add_node(int n);
```
Method adds node n to the graph. If graph has already contained node denoted as n, method does nothing and returns false. In the case of successful addition, method returns true.

```
bool del_node(int n);
```
Method deletes node n from the graph. If graph hasn't contained node denoted as n, method does nothing and returns false. In the case of successful deletion, method returns true. Method also deletes all edges adjacent to the deleted node (outgoing and ingoing).

```
bool add_edge(int n1, int n2);
```
Method adds edge <n1,n2> to the graph. If graph has already contained such edge or it hasn't contained n1 or n2 node, method does nothing and returns false. In the case of successful addition method, returns true.

```
bool del_edge(int n1, int n2);
```
Method deletes edge <n1,n2> from the graph. If graph hasn't contained such edge, method does nothing and returns false. In the case of successful deletion, method returns true.

All the above methods excluding del_node should have constant average time complexity.

```
vector<int> nodes();
```
Method returns vector containing nodes' numbers (for mentioned above 3-nodes graph method returns vector [0, -3, 5]).

```
vector<pair<int,int>> out_edges(int n);
```
Method returns vector containing all edges which are outgoing from specified node. Formally each edge is pair of integers and first element of this pair always equals n.

```
int nodes_count() const;
```
Method returns number of graph's nodes.
This method should have constant worst-case time complexity.

```
int edges_count() const;
```
Method returns number of edges of whole graph.
This method should have constant worst-case time complexity (it means that you can't count edges in this method, you must store number of edges and methods used for graph modification should update this number).

Static methods

```
static graph reverse(const graph& g);
```
Method creates new graph with the same set of nodes but reversed directions of all edges. Original graph stays unchanged.

```
static vector<int> shortest_path(graph& g, int n1, int n2);
```
Method finds shortest path from node n1 to node n2. Returned vector contains numbers of consecutive nodes on the path. For instance for example graph from attached file graph1.txt the correct answer is [3,5,2,7].
Hint: use widely known BFS (breadth-first search) algorithm.

```
static int coloring(graph& g, map<int,int>& colors);
```
Method finds optimal no-collision graph node coloring. It means that method assigns to each graph node a number (usually called color) in such a way that there is no adjacency nodes (nodes connected by edge) with the same color number. Optimal coloring means that the minimal possible number of colors should be used .Formally method returns found assignment by its second parameter which is a map for which nodes descriptions in graph are map keys and assigned colors are map values. Returned value is number of used colors (it should be as minimal as possible). Colors should be numbered starting from 1 and consecutive.
Hint1: use brute-force backtracking search (the best known example of backtracking search is solving N-Queens problem)
Hint2: define private recurrent helper function

Define also two operators which should be friends of graph class

```
friend ostream& operator<<(ostream &out, const graph& g);
```
Operator writes graph description to the given stream (not only console, it can be for instance file stream). Graph description has the following format:

- each line is related to one node
- each line has the following format:
  - first element is node description
  - next element is colon
  - next part of line are descriptions of nodes which are reached by the edges outgoing from the node denoted by the first element of the line
  - last element of the line is semicolon

Files graph0.txt, graph1.txt, graph2.txt may be used as examples of correct output of <<operator.

```
friend istream& operator>>(istream &in, graph& g);
```
Operator reads graph description from the given stream (console or file stream) and using this description overwrites graph which is its second parameter (all previously existing graph nodes and edges are deleted). Input format for >>operator is the same as output format of <<operator.
Hint: Don't read stream char by char, nodes description can be positive or negative and can contain many digits. Try to read whole number as one element.

Grading:
Students who are present at online laboratory 2 and 3 class are not required to do C++ task (but they may do it of course)

1) All instance methods and <<operator are basic operations and any error within implementation of these methods can cause negative penalty points (this refers only to students absent from regular online 2 and 3 laboratory class). Maximum penalty is -4 points and it is penalty for students absent from regular online 2 and 3 laboratory class who haven't sent solutions of this task to theirs teachers.

2) Error free implementation of the methods mentioned above without implementation of >>operator and any static method means simply no penalty and no bonus point.

Next we assume error free implementation of all instance methods and <<operator

3) Error free implementation of >>operator and reverse method means 1 additional bonus point.

4) Error free implementation of shortest_path method means 1 additional point (implementation of >>operator is also required for testing purposes)

5) Error free implementation of coloring method means 2 additional points (implementation of >>operator is also required for testing purposes)

6) It means that error free implementation of all required methods means 4 bonus points