Specification

We define a class named Graph representing a directed graph.

We are using an auxiliary structure named Edge which contains information about this two vertices connected by that edge and the cost of it, we also define an equality operator and a hash function to be able to use it in the unordered_set container from c++. We are also defining our own exception classes (VertexError, EdgeError) used in handling errors in manipulating a graph.

The class Graph will provide the following methods:

Graph();

Constructs an empty graph.

explicit Graph(int vertices_no);

Constructs a graph with a given number of vertices numbered consecutively from 0.

Graph(int vertices_no, int edges_no);

Constructs a random graph with a given number of vertices numbered consecutively from 0 and a given number of random edges with random costs.

set_int::const_iterator vertices_begin();

Returns an stl iterator for the beginning of the set of vertices.

set_int::const_iterator vertices_end();

Returns an stl iterator for the end of the set of vertices.

set_int::const_iterator neighbours_begin(int vertex);

Returns an stl iterator for the beginning of the set of neighbors of a vertex.

set_int::const_iterator neighbours_end(int vertex);

Returns an stl iterator for the end of the set of neighbors of a vertex.

set_int::const_iterator transpose_begin(int vertex);

Returns an stl iterator for the beginning of the set of incoming neighbors of a vertex.

set_int::const_iterator transpose_end(int vertex);

Returns an stl iterator for the end of the set of incoming neighbors of a vertex.

set_edge::const_iterator edges_begin();

Returns an stl iterator for the beginning of the set of edges.

set_edge::const_iterator edges_end();

Returns an stl iterator for the end of the set of edges.

bool is_edge(int vertex1, int vertex2);

Retursn true if there is an edge between vertex1 and vertex2

bool is_edge(pair_int edge);

Returns true if the pair of ints forms an edge.

bool is_vertex(int vertex);

Returns true if the given vertex is in the graph.

int count_vertices();

Returns the number of vertices.

int count_edges();

Returns the number of edges.

int in_degree(int vertex);

Returns the number of incoming edges towards vertex.

int out_degree(int vertex);

Returns the number of outgoing edges towards vertex.

int get_edge_cost(int vertex1, int vertex2);

Returns the cost of an edge given by vertex1 and vertex2.

void set_edge_cost(int vertex1, int vertex2, int new_cost);

Sets the cost of an edge given by vertex1 and vertex2.

void add_vertex(int vertex);

Adds a vertex to the graph.

void remove_vertex(int vertex);

Removes a vertex from the graph and all its incoming and outgoing edges.

void add_edge(int vertex1, int vertex2, int edge_cost = 0);

Adds an edge to the graph, the cost of the edge is optional.

void remove_edge(int vertex1, int vertex2);

Removes an edge from the graph.

Implementation

The implementation uses an unordered set of integers for its vertices, an unordered map for the incoming and outgoing neighbors of an edge, where the key of a map is the vertex in question and the value is an unordered_set of such neighbors. Finally the edges are stored in another unordered_set of edges. We use typedef to ease the implementation.

```
typedef std::pair<int, int> pair_int;

typedef std::unordered_set<int> set_int;

typedef std::unordered_map< int, set_int > map_int;

typedef std::unordered_set< Edge > set_edge;

set_int vertices;

map_int edges, transpose;

set_edge cost;
```