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;