Directed Graph

Documentation 1 – Programming Language C++

Representation:

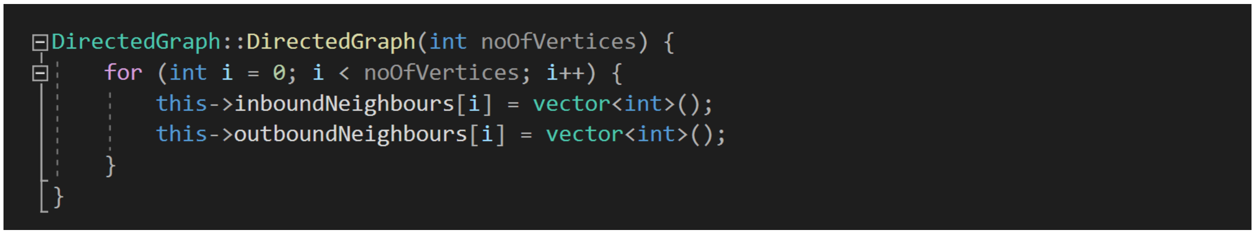
The Abstract Data Type Directed Graph is represented using three maps. For each vertex, we kept a collection of its neighbours (inbound and outbound) and another map which contains pairs of edges and costs.

- inboundNeighbours: it will store for every vertex a vector of all the vertices which are the starting point of a edge, where the ending point is the given vertex
- outboundNeighbours: it will store for every vertex a vector of all the vertices which are the starting point of a edge, where the starting point is the given vertex
- costOfEdge: for every edge (vertex1, vertex2) stored as a pair it will store the cost associated to that edge

Interface

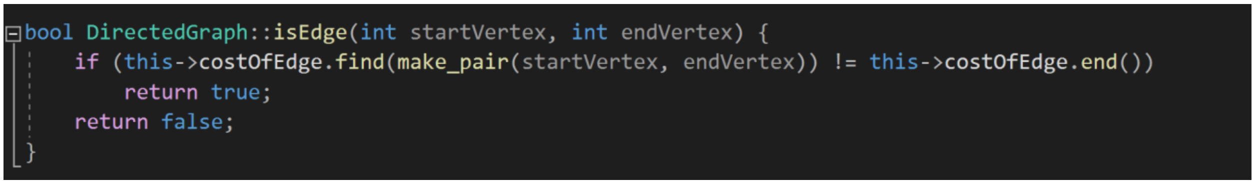
The interface of the ADT is consisted of the following functions:

- DirectedGraph(numberOfVertices):
    - constructor function
    - creates an directed graph with numberOfVertices vertices: from 0 to numberOfVertices – 1 and 0 edges



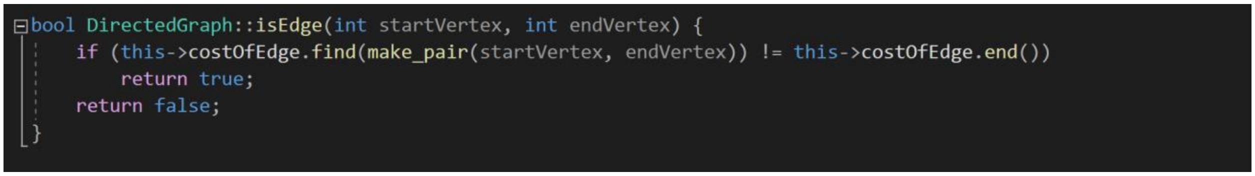
```
DirectedGraph::DirectedGraph(int noOfVertices) {
    for (int i = 0; i < noOfVertices; i++) {
        this->inboundNeighbours[i] = vector<int>();
        this->outboundNeighbours[i] = vector<int>();
    }
}
```

- isEdge(startVertex, endVertex)
    - check if the edge (startVertex, endVertex) is an edge into the graph
    - Preconditions:
    - startVertex, endVertex – integers



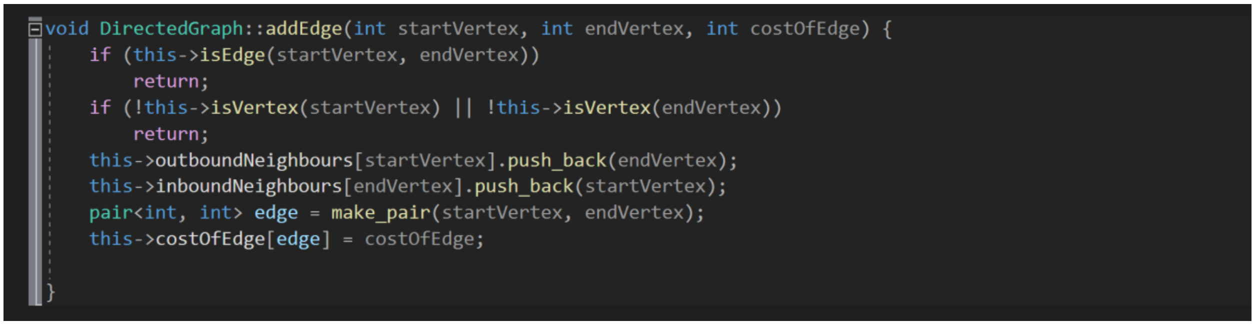
```
bool DirectedGraph::isEdge(int startVertex, int endVertex) {
    if (this->costOfEdge.find(make_pair(startVertex, endVertex)) != this->costOfEdge.end())
        return true;
    return false;
}
```

- isVertex(vertex)
    - check if the vertex belongs to the graph
    - Preconditions:
        - Vertex – integer



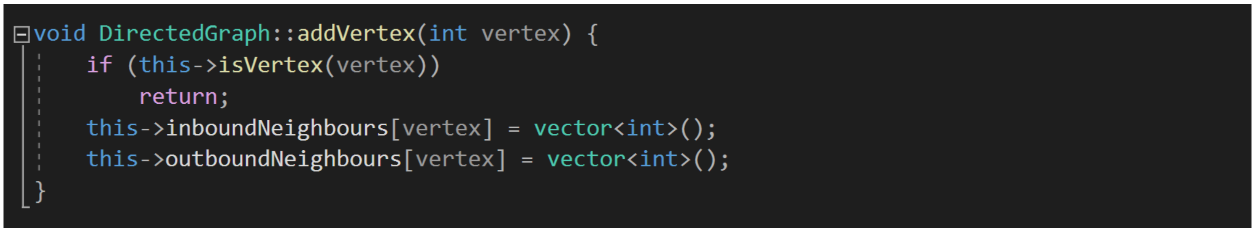
```
bool DirectedGraph::isEdge(int startVertex, int endVertex) {
    if (this->costOfEdge.find(make_pair(startVertex, endVertex)) != this->costOfEdge.end())
        return true;
    return false;
}
```

- addEdge(startVertex, endVertex, costOfTheEdge):
    - add the edge (startVertex, endVertex) with cost costOfTheEdge to the graph
    - Preconditons:
        - startVertex, endVertex – integer
        - the edge doesn’t aready exist into the graph



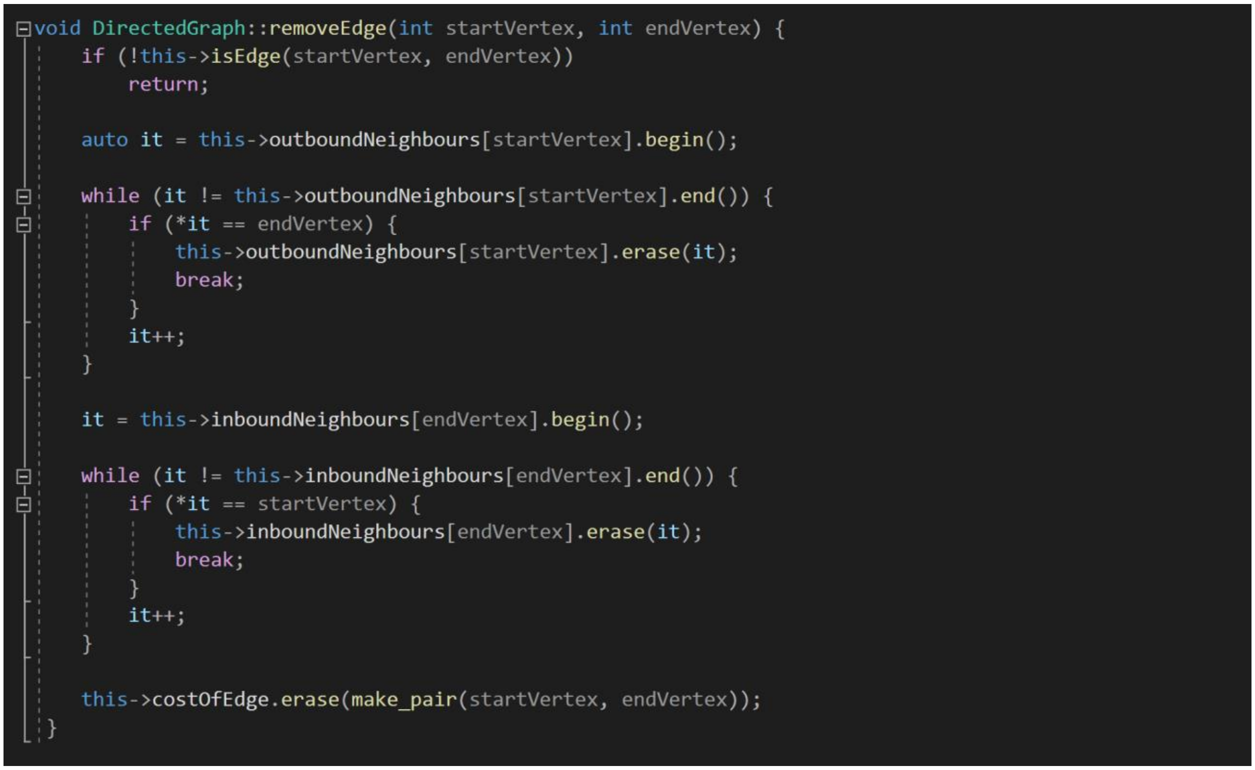
```
void DirectedGraph::addEdge(int startVertex, int endVertex, int costOfEdge) {
    if (this->isEdge(startVertex, endVertex))
        return;
    if (!this->isVertex(startVertex) || !this->isVertex(endVertex))
        return;
    this->outboundNeighbours[startVertex].push_back(endVertex);
    this->inboundNeighbours[endVertex].push_back(startVertex);
    pair<int, int> edge = make_pair(startVertex, endVertex);
    this->costOfEdge[edge] = costOfEdge;

}
```

- addVertex(vertex)
    - add a new vertex to the graph
    - Preconditions:
        - the vertex doesn’t already exist in the graph



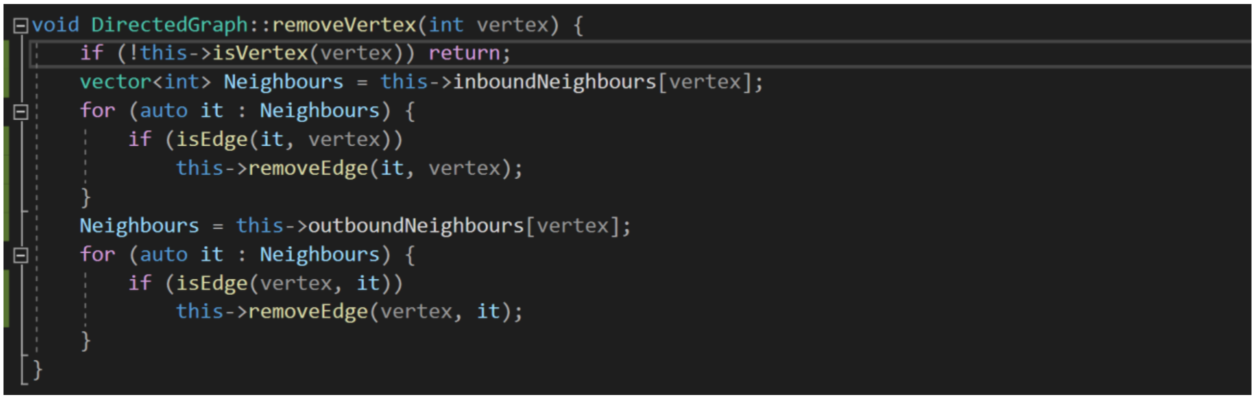
```
void DirectedGraph::addVertex(int vertex) {
    if (this->isVertex(vertex))
        return;
    this->inboundNeighbours[vertex] = vector<int>();
    this->outboundNeighbours[vertex] = vector<int>();
}
```

- removeEdge(startVertex, endVertex)
    - remove the edge (startVertex, endVertex) from the graph
    - Preconditions:
        - startVertex, endVertex – integers
        - the edge has to exist into the graph



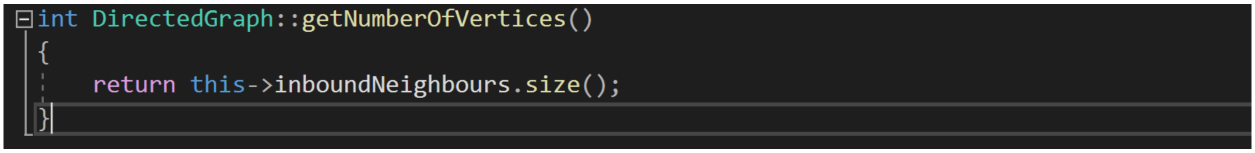
```cpp
void DirectedGraph::removeEdge(int startVertex, int endVertex) {
    if (!this->isEdge(startVertex, endVertex))
        return;

    auto it = this->outboundNeighbours[startVertex].begin();

    while (it != this->outboundNeighbours[startVertex].end()) {
        if (*it == endVertex) {
            this->outboundNeighbours[startVertex].erase(it);
            break;
        }
        it++;
    }

    it = this->inboundNeighbours[endVertex].begin();

    while (it != this->inboundNeighbours[endVertex].end()) {
        if (*it == startVertex) {
            this->inboundNeighbours[endVertex].erase(it);
            break;
        }
        it++;
    }

    this->costOfEdge.erase(make_pair(startVertex, endVertex));
}
```

- removeVertex(vertex)
    - remove the vertex from the graph
    - Preconditions:
        - the vertex doesn’t already exist in the graph



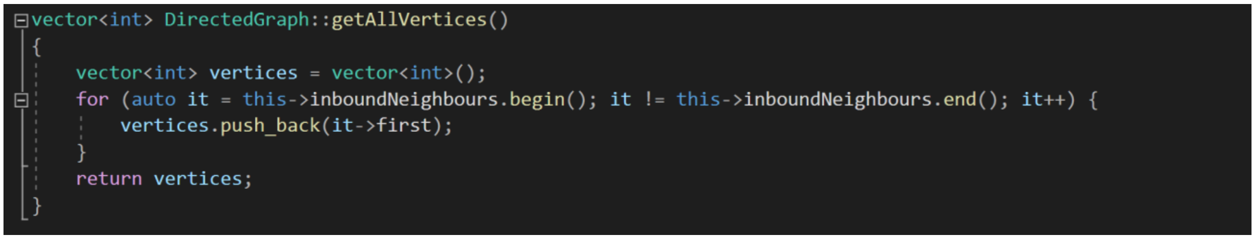
```cpp
void DirectedGraph::removeVertex(int vertex) {
    if (!this->isVertex(vertex)) return;
    vector<int> Neighbours = this->inboundNeighbours[vertex];
    for (auto it : Neighbours) {
        if (isEdge(it, vertex))
            this->removeEdge(it, vertex);
    }
    Neighbours = this->outboundNeighbours[vertex];
    for (auto it : Neighbours) {
        if (isEdge(vertex, it))
            this->removeEdge(vertex, it);
    }
}
```

- getNumberOfVertices()
    - returns the total number of vertices of the graph



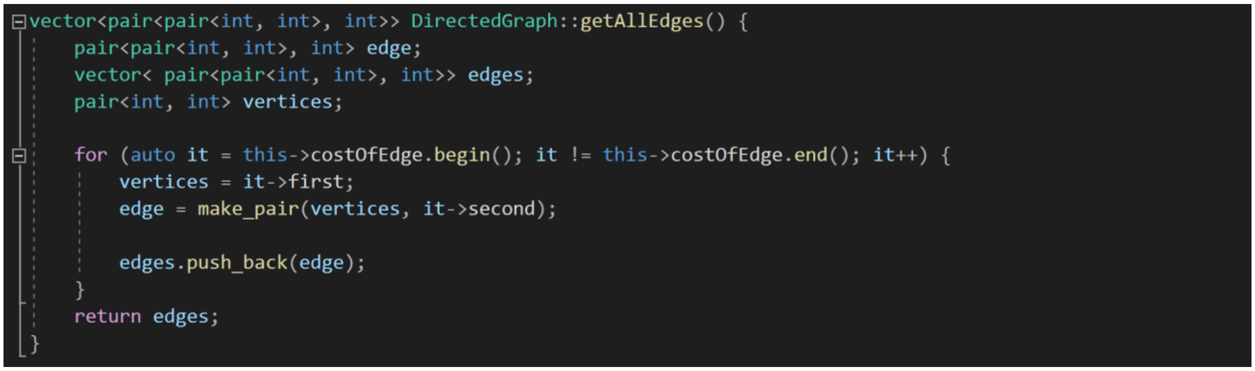
```cpp
int DirectedGraph::getNumberOfVertices()
{
    return this->inboundNeighbours.size();
}
```

- getAllVertices()
    - returns a list containing all the vertices of the graph



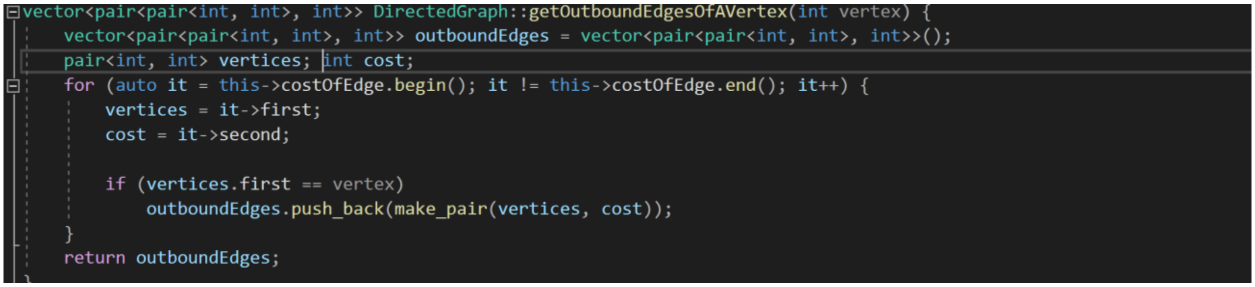
```cpp
vector<int> DirectedGraph::getAllVertices()
{
    vector<int> vertices = vector<int>();
    for (auto it = this->inboundNeighbours.begin(); it != this->inboundNeighbours.end(); it++) {
        vertices.push_back(it->first);
    }
    return vertices;
}
```

- getAllEdges()
    - returns a list containing tuples, which represent all the edges from the graph



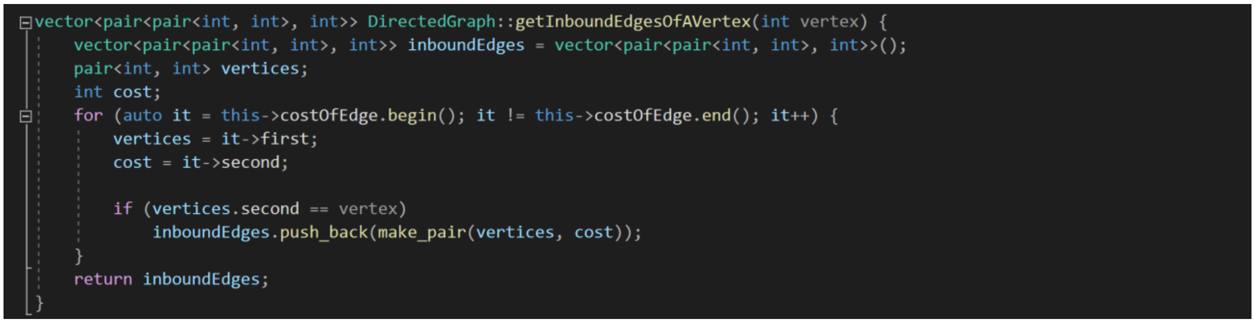
```cpp
vector<pair<pair<int, int>, int>> DirectedGraph::getAllEdges() {
    pair<pair<int, int>, int> edge;
    vector< pair<pair<int, int>, int>> edges;
    pair<int, int> vertices;

    for (auto it = this->costOfEdge.begin(); it != this->costOfEdge.end(); it++) {
        vertices = it->first;
        edge = make_pair(vertices, it->second);

        edges.push_back(edge);
    }
    return edges;
}
```

- getOutboundEdges(vertex)
    - returns a list containing all the outbound edges of a given vertex
    - Precodition:
        - vertex has to exist into the graph



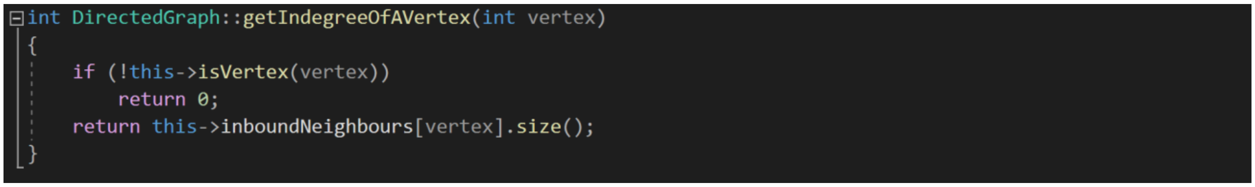
```cpp
vector<pair<pair<int, int>, int>> DirectedGraph::getOutboundEdgesOfAVertex(int vertex) {
    vector<pair<pair<int, int>, int>> outboundEdges = vector<pair<pair<int, int>, int>>();
    pair<int, int> vertices; int cost;
    for (auto it = this->costOfEdge.begin(); it != this->costOfEdge.end(); it++) {
        vertices = it->first;
        cost = it->second;

        if (vertices.first == vertex)
            outboundEdges.push_back(make_pair(vertices, cost));
    }
    return outboundEdges;
```

- getInboundEdges(vertex)
  - returns a list containing all the inbound edges of a given vertex
  - Preconditions:
    - vertex – integer
    - vertex has to exist into the graph



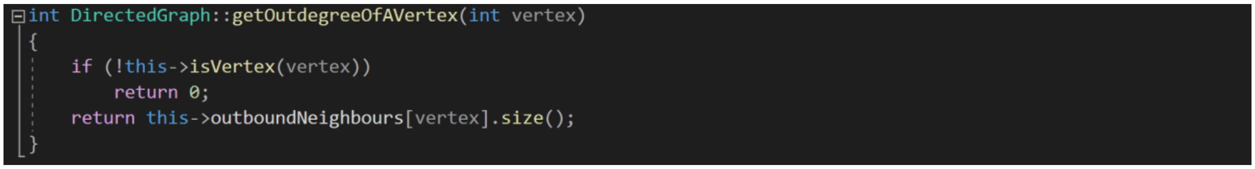
```
vector<pair<pair<int, int>, int>> DirectedGraph::getInboundEdgesOfAVertex(int vertex) {
    vector<pair<pair<int, int>, int>> inboundEdges = vector<pair<pair<int, int>, int>>();
    pair<int, int> vertices;
    int cost;
    for (auto it = this->costOfEdge.begin(); it != this->costOfEdge.end(); it++) {
        vertices = it->first;
        cost = it->second;

        if (vertices.second == vertex)
            inboundEdges.push_back(make_pair(vertices, cost));
    }
    return inboundEdges;
}
```

- getIndegreeOfAVertex(vertex)
  - returns an integer representing the number of inbound vertices of a given vertex
  - Preconditions:
    - vertex – integer
    - vertex has to exist into the graph



```
int DirectedGraph::getIndegreeOfAVertex(int vertex)
{
    if (!this->isVertex(vertex))
        return 0;
    return this->inboundNeighbours[vertex].size();
}
```

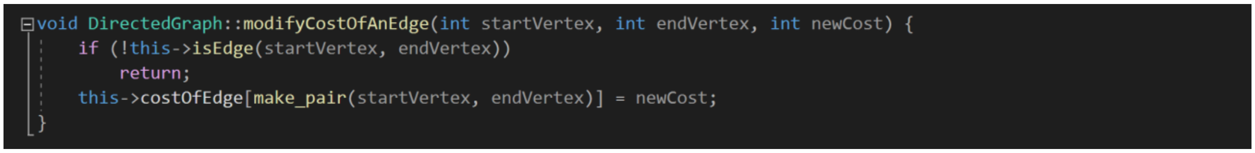
- getOutDegreeOfAVertex(vertex)
  - returns an integer containing the number of outbound vertices of a given vertex in the graph
  - Preconditions:
    - vertex – integer
    - the vertex has to exist into the graph



```
int DirectedGraph::getOutdegreeOfAVertex(int vertex)
{
    if (!this->isVertex(vertex))
        return 0;
    return this->outboundNeighbours[vertex].size();
}
```

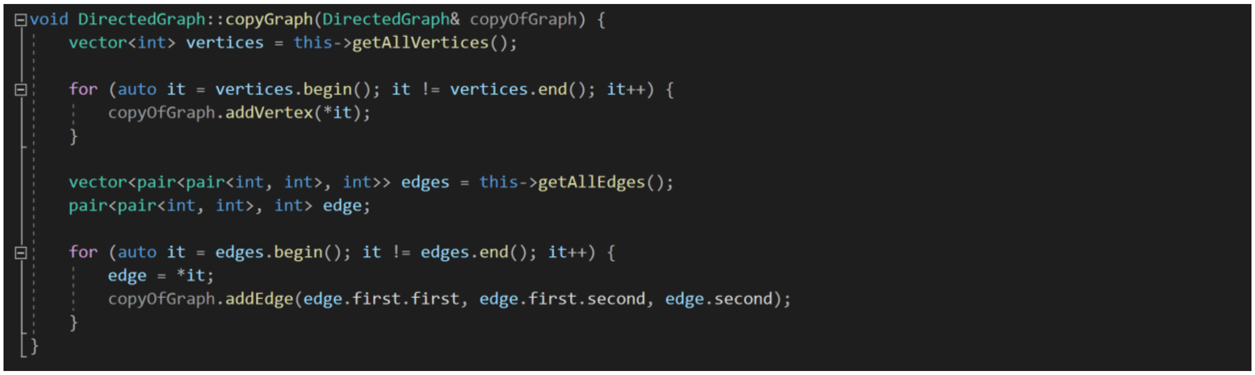
- modifyCostOfAnEdge(startVertex, endVertex, newCostOfEdge)

  - change the cost of the edge (startVertex, endVertex) with the newCostOfEdge
  - Preconditions:
    - startVertex, endVertex, newCostOfEdge – integers
    - the edge (startVertex, endVertex) has to exist into the graph



```cpp
void DirectedGraph::modifyCostOfAnEdge(int startVertex, int endVertex, int newCost) {
    if (!this->isEdge(startVertex, endVertex))
        return;
    this->costOfEdge[make_pair(startVertex, endVertex)] = newCost;
}
```

- copyGraph()
  - returns a new graph which represent the copy of the initial graph

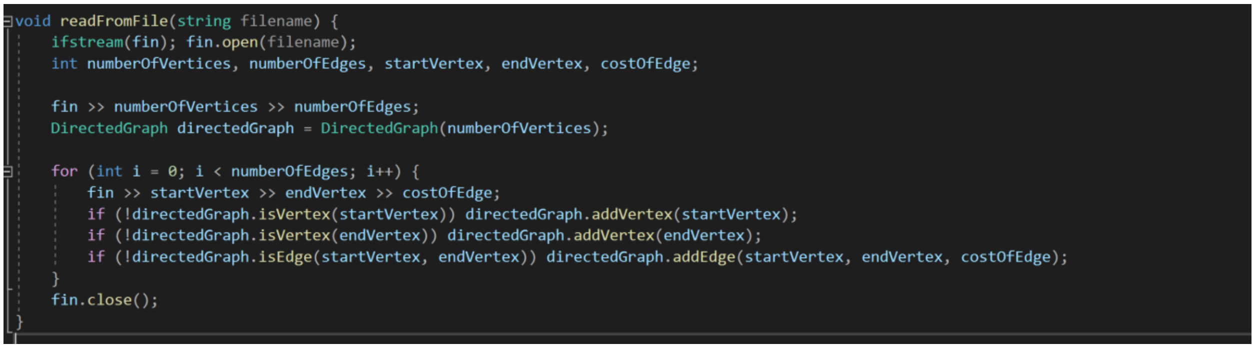


```cpp
void DirectedGraph::copyGraph(DirectedGraph& copyOfGraph) {
    vector<int> vertices = this->getAllVertices();

    for (auto it = vertices.begin(); it != vertices.end(); it++) {
        copyOfGraph.addVertex(*it);
    }

    vector<pair<pair<int, int>, int>> edges = this->getAllEdges();
    pair<pair<int, int>, int> edge;

    for (auto it = edges.begin(); it != edges.end(); it++) {
        edge = *it;
        copyOfGraph.addEdge(edge.first.first, edge.first.second, edge.second);
    }
}
```

External functions

The external functions implemented are:

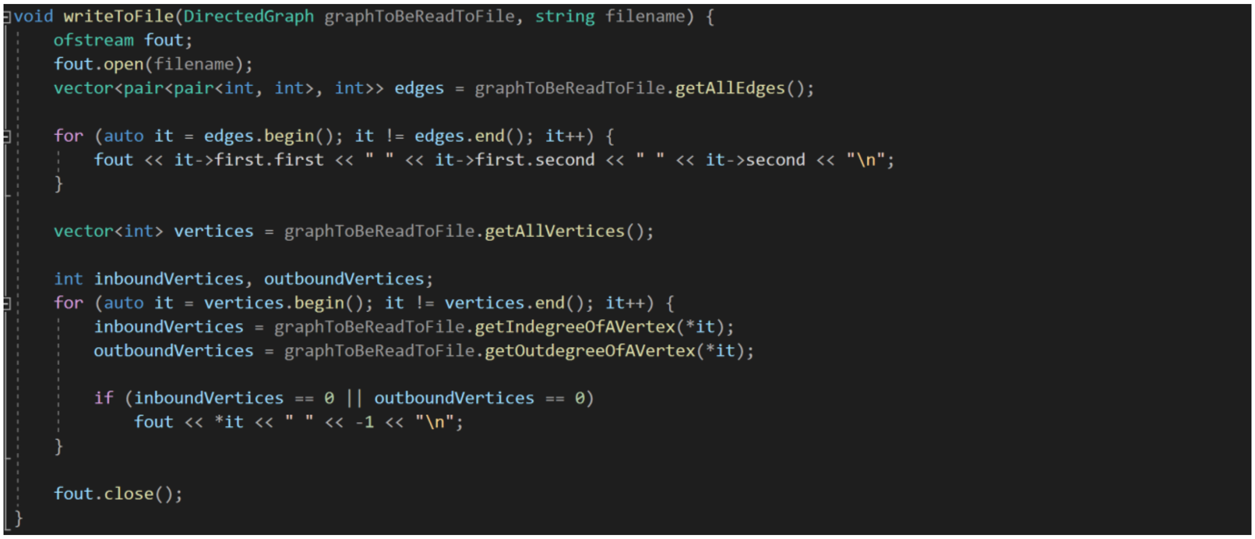
- readGraphFromFile(filename)
  - read vertices and edges from the file and construct a graph according to the information read
  - return: a new graph



```cpp
void readFromFile(string filename) {
    ifstream(fin); fin.open(filename);
    int numberOfVertices, numberOfEdges, startVertex, endVertex, costOfEdge;

    fin >> numberOfVertices >> numberOfEdges;
    DirectedGraph directedGraph = DirectedGraph(numberOfVertices);

    for (int i = 0; i < numberOfEdges; i++) {
        fin >> startVertex >> endVertex >> costOfEdge;
        if (!directedGraph.isVertex(startVertex)) directedGraph.addVertex(startVertex);
        if (!directedGraph.isVertex(endVertex)) directedGraph.addVertex(endVertex);
        if (!directedGraph.isEdge(startVertex, endVertex)) directedGraph.addEdge(startVertex, endVertex, costOfEdge);
    }
    fin.close();
}
```

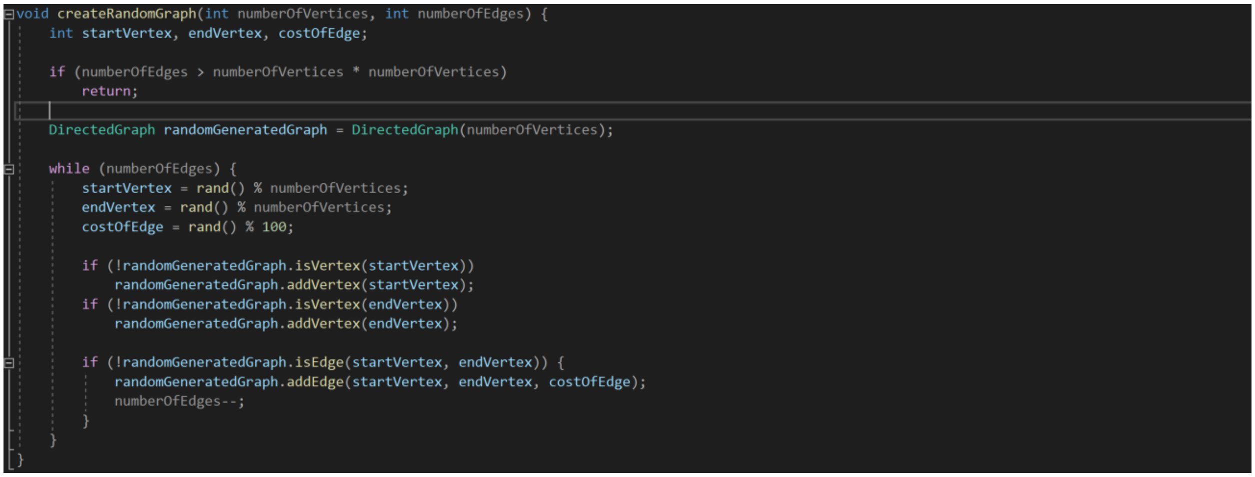
- writeGraphToFile(graph, filename)

- write the vertices and edges of the graph in a file



```
void writeToFile(DirectedGraph graphToBeReadToFile, string filename) {
    ofstream fout;
    fout.open(filename);
    vector<pair<pair<int, int>, int>> edges = graphToBeReadToFile.getAllEdges();

    for (auto it = edges.begin(); it != edges.end(); it++) {
        fout << it->first.first << " " << it->first.second << " " << it->second << "\n";
    }

    vector<int> vertices = graphToBeReadToFile.getAllVertices();

    int inboundVertices, outboundVertices;
    for (auto it = vertices.begin(); it != vertices.end(); it++) {
        inboundVertices = graphToBeReadToFile.getIndegreeOfAVertex(*it);
        outboundVertices = graphToBeReadToFile.getOutdegreeOfAVertex(*it);

        if (inboundVertices == 0 || outboundVertices == 0)
            fout << *it << " " << -1 << "\n";
    }

    fout.close();
}
```

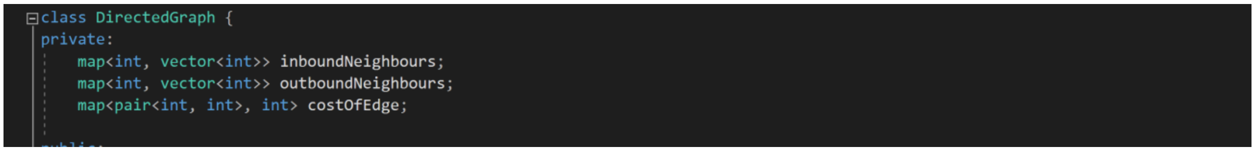
- getRandomGraph(int numberOfVertices, int numberOfEdges)
    - creates a new graph with random vetices and random edges
    - return: a new graph
    - Precondition: the numberOfVertices <= numberOfVertices^2



```
void createRandomGraph(int numberOfVertices, int numberOfEdges) {
    int startVertex, endVertex, costOfEdge;

    if (numberOfEdges > numberOfVertices * numberOfVertices)
        return;

    DirectedGraph randomGeneratedGraph = DirectedGraph(numberOfVertices);

    while (numberOfEdges) {
        startVertex = rand() % numberOfVertices;
        endVertex = rand() % numberOfVertices;
        costOfEdge = rand() % 100;

        if (!randomGeneratedGraph.isVertex(startVertex))
            randomGeneratedGraph.addVertex(startVertex);
        if (!randomGeneratedGraph.isVertex(endVertex))
            randomGeneratedGraph.addVertex(endVertex);

        if (!randomGeneratedGraph.isEdge(startVertex, endVertex)) {
            randomGeneratedGraph.addEdge(startVertex, endVertex, costOfEdge);
            numberOfEdges--;
        }
    }
}
```

Class representation

For representing the neighbours, both inbound and outbound, we used 2 maps, and for the costs of the edges we also used a map.



```
class DirectedGraph {
private:
    map<int, vector<int>> inboundNeighbours;
    map<int, vector<int>> outboundNeighbours;
    map<pair<int, int>, int> costOfEdge;

public:
```