Graph algorithms - practical work no. 1

Cretu Cristian - 913

Constructor

```
Graph(uint32_t vertices = 0, uint32_t edges = 0)
      : vertices(vertices), edges(edges) {}
```

Initiializes a new instance of the `Graph` class with the specified number of vertices and edges. If the number of vertices is not specified, the graph will be initialized with 0 vertices. If the number of edges is not specified, the graph will be initialized with 0 edges. **Complexity:** $O(1)$

Destructor

```
~Graph() {
    for (auto &edge : outbound) {
      edge.second.clear();
    }
    for (auto &edge : inbound) {
      edge.second.clear();
    }
  }
```

Destroys the graph and clears the memory allocated for the edges, since the inbound and outbound edges are stored in a `std::list` data structure. **Complexity:** $O(V + E)$

addEdge

```
void addEdge(uint32_t source, uint32_t target, int cost) {
    if (!this->isEdge(source, target)) {
      int edgeID = generateEdgeID();
      this->outbound[source].push_back({target, edgeID});
      this->inbound[target].push_back({source, edgeID});
      this->cost[edgeID] = cost;
    }
  }
```

Adds a new edge from the source to the target vertex with the specified cost. If the edge already exists, the method does nothing. **Complexity:** $O(1)$

removeEdge

```
void removeEdge(uint32_t source, uint32_t target) {
    if (this->isEdge(source, target)) {
        int edgeID = getEdgeID(source, target);
        outbound[source].remove_if(
            [edgeID](const std::pair<uint32_t, uint32_t> &edge) {
            return edge.second == edgeID;
            });
        inbound[target].remove_if(
            [edgeID](const std::pair<uint32_t, uint32_t> &edge) {
            return edge.second == edgeID;
            });
```

```
            cost.erase(edgeID);
        }
    }
```

Removes the edge from the source to the target vertex. If the edge does not exist, the method does nothing. **Complexity:** $O(V + E)$

addVertex

```
void addVertex(uint32_t vertex) {
    if (outbound.find(vertex) == outbound.end()) {
      outbound[vertex] = std::list<std::pair<uint32_t, uint32_t>>();
      inbound[vertex] = std::list<std::pair<uint32_t, uint32_t>>();
      vertices++;
    }
  }
```

Adds a new vertex to the graph. If the vertex already exists, the method does nothing. **Complexity:** $O(1)$

removeVertex

```
void removeVertex(uint32_t vertex) {
    if (outbound.find(vertex) != outbound.end()) {
      for (auto &edge : std::vector<std::pair<uint32_t, uint32_t>>(
               outbound[vertex].begin(), outbound[vertex].end())) {
        removeEdge(vertex, edge.first);
      }

      for (auto &edge : std::vector<std::pair<uint32_t, uint32_t>>(
               inbound[vertex].begin(), inbound[vertex].end())) {
        removeEdge(edge.first, vertex);
      }

      outbound.erase(vertex);
      inbound.erase(vertex);

      vertices--;
    }
  }
```

It removes the vertex from the graph and all the edges that have the vertex as a source or target. If the vertex does not exist, the method does nothing. **Complexity:** $O(V + E)$

getOutboundEdges

```
std::pair<EdgeIterator, EdgeIterator> getOutboundEdges(
      uint32_t vertex) const {
    if (outbound.find(vertex) != outbound.end()) {
      return {outbound.at(vertex).cbegin(), outbound.at(vertex).cend()};
```

```
    }
    static const std::list<std::pair<uint32_t, uint32_t>> empty;
    return {empty.cbegin(), empty.cend()};
  }
```

Returns a pair of iterators that define the range of outbound edges for the specified vertex. If the vertex does not exist, the method returns an empty list. **Complexity:** $O(1)$

getInboundEdges

```
std::pair<EdgeIterator, EdgeIterator> getInboundEdges(uint32_t vertex) const {
    if (inbound.find(vertex) != inbound.end()) {
        return {inbound.at(vertex).cbegin(), inbound.at(vertex).cend()};
    }
    static const std::list<std::pair<uint32_t, uint32_t>> empty;
    return {empty.cbegin(), empty.cend()};
    }
```

Returns a pair of iterators that define the range of inbound edges for the specified vertex. If the vertex does not exist, the method returns an empty list. **Complexity:** $O(1)$

getOutEdges

```
std::vector<std::pair<uint32_t, uint32_t>> getOutEdges(
    uint32_t vertex) const {
    if (outbound.find(vertex) != outbound.end()) {
    return std::vector<std::pair<uint32_t, uint32_t>>(
        outbound.at(vertex).begin(), outbound.at(vertex).end());
    }
    return std::vector<std::pair<uint32_t, uint32_t>>();
}
```

Returns a vector of outbound edges for the specified vertex. If the vertex does not exist, the method returns an empty list. **Complexity:** $O(1)$

getInEdges

```
std::vector<std::pair<uint32_t, uint32_t>> getInEdges(uint32_t vertex) const {
    if (inbound.find(vertex) != inbound.end()) {
    return std::vector<std::pair<uint32_t, uint32_t>>(
        inbound.at(vertex).begin(), inbound.at(vertex).end());
    }
    return std::vector<std::pair<uint32_t, uint32_t>>();
}
```

Returns a vector of inbound edges for the specified vertex. If the vertex does not exist, the method returns an empty list. **Complexity:** $O(1)$

getVerticesList

```
std::vector<uint32_t> getVerticesList() const {
    std::vector<uint32_t> verticesList;
    for (auto edge : outbound) {
    verticesList.push_back(edge.first);
    }
    return verticesList;
}
```

Returns a vector of vertices in the graph as const references. If the graph is empty, the method returns an empty list. **Complexity:** $O(V)$

getEdgeID

```
uint32_t getEdgeID(uint32_t source, uint32_t target) const {
    if (outbound.find(source) != outbound.end()) {
    for (const auto &edge : outbound.at(source)) {
        if (edge.first == target) {
        return edge.second;
        }
    }
    }
    return -1;
}
```

Returns the edge ID for the edge from the source to the target vertex. If the edge does not exist, the method returns -1. The EdgeID is calculated as the number of edges in the graph at the moment the edge is added. **Complexity:** $O(V)$

isEdge

```
bool isEdge(uint32_t source, uint32_t target) const {
    return getEdgeID(source, target) != -1;
}
```

Returns true if the edge from the source to the target vertex exists, otherwise returns false. **Complexity:** $O(V)$

getInDegree

```
uint32_t getInDegree(uint32_t vertex) const {
    if (inbound.find(vertex) != inbound.end()) {
    return inbound.at(vertex).size();
    }
    return 0;
}
```

Returns the in-degree of the specified vertex. If the vertex does not exist, the method returns 0. **Complexity:** $O(1)$

getOutDegree

```
uint32_t getOutDegree(uint32_t vertex) const {
    if (outbound.find(vertex) != outbound.end()) {
    return outbound.at(vertex).size();
    }
    return 0;
}
```

Returns the out-degree of the specified vertex. If the vertex does not exist, the method returns 0. **Complexity:** $O(1)$

getCost

```
int getCost(uint32_t source, uint32_t target) const {
    if (isEdge(source, target)) {
    int edgeID = getEdgeID(source, target);
    auto it = cost.find(edgeID);
    if (it != cost.end()) {
        return it->second;
    }
    }
    return -1;
}
```

Returns the cost of the edge from the source to the target vertex. If the edge does not exist, the method returns -1. **Complexity:** $O(V)$

setCost

```
void setCost(uint32_t source, uint32_t target, int cost) {
    if (isEdge(source, target)) {
    this->cost[getEdgeID(source, target)] = cost;
    }
}
```

Sets the cost of the edge from the source to the target vertex. If the edge does not exist, the method does nothing. **Complexity:** $O(V)$

getEndpoints

```
std::pair<uint32_t, uint32_t> getEndpoints(uint32_t edgeID) const {
    for (auto &edge : outbound) {
    for (auto &target : edge.second) {
        if (target.second == edgeID) {
        return {edge.first, target.first};
        }
    }
    }
    return {-1, -1};
}
```

Returns a pair of vertices that are the endpoints of the edge with the specified edge ID. If the edge does not exist, the method returns (-1, -1). **Complexity:** $O(V)$

copyGraph

```
Graph copyGraph() const {
    Graph copy(vertices, edges);
    for (auto &edge : outbound) {
    for (auto &target : edge.second) {
        copy.addEdge(edge.first, target.first,
                     getCost(edge.first, target.first));
    }
    }

    return copy;
}
```

Returns a copy of the graph. The method creates a deep copy of the graph, including all the edges and vertices. **Complexity:** $O(V + E)$

createRandomGraph

```
static Graph createRandomGraph(uint32_t vertices, uint32_t maxEdges) {
    Graph randomGraph;
    randomGraph.setVertices(vertices);

    for (uint32_t i = 0; i < maxEdges; i++) {
        uint32_t source = rand() % vertices;
        uint32_t target = rand() % vertices;

        if (source != target && !randomGraph.isEdge(source, target)) {
            int cost = rand() % 100;
            randomGraph.addEdge(source, target, cost);
        } else {
            i--;
        }
    }

    return randomGraph;
}
```

Creates a random graph with the specified number of vertices and edges. The method keeps generating random VALID edges until the number of edges reaches the specified maximum number of edges. **Complexity:** $O(V + E)$