Praktikum 3: Klassen im Zusammenspiel

1. Lernziele

Die folgenden, in der Vorlesung behandelten Themen sollen vertieft und angewendet werden:

- Objektorientierte Programmierung
- Klassen- und Objektmodell
- Einbinden von externen Bibliotheken

2. Aufgabe

In diesem Praktikum soll ein Modell für die Implementierung eines gewichteten, gerichteten Graphen entwickelt werden.

Teil 1 (Datenstruktur): Graphen (siehe Abbildung 1a) bestehen aus Knoten und Kanten. Knoten sind darin als Kreise und Kanten als Pfeile dargestellt. Erstellen Sie hierzu die Klasse `DiGraph` als Hauptklasse für den Graphen, sowie die Klassen `Node` und `Edge`, für die Knoten und Kanten.

Da ein Graph aus einer beliebigen Anzahl von Knoten und Kanten bestehen kann, muss eine geeignete Datenstruktur zur Speicherung der Objekte gewählt werden.

Die gängigste Form einen Graphen in einem Programm darzustellen ist die sogenannte Adjazenzliste (siehe Abbildung 1b). In einer Adjazenzliste werden zu jedem Knoten alle von ihm ausgehenden Kanten abgespeichert. Realisieren Sie dies in Ihrer Implementierung, indem Sie der Klasse `Node`, die in der Vorlesung entwickelte Liste hinzufügen.

In den Klassen müssen zusätzlich folgende Methoden vorhanden sein:

DiGraph:

Methoden

- `void addNode(Node * node)`
  Fügt die Adresse eines Klassenobjekts vom Typ `Node` dem Graphen hinzu.
- `void addEdge(std::string key1, std::string key2, float weight)`
  Erstellt ein neues Klassenobjekt vom Typ `Edge` mit Startknoten `key1`, Endknoten `key2` sowie einem Kantengewicht von `weight`. Anschließend wird die Kante der Adjazenzliste des Knotens `key1` hinzugefügt.

- `Liste<Node*> getNeighbours(std::string key)`
  Gibt eine Liste aller benachbarter Knoten des Knotens mit dem Schlüssel `key` zurück.
- `Liste<Edge*> getEdges(std::string key)`
  Gibt eine Liste aller abgehenden Kanten des Knotens mit dem Schlüssel `key` zurück.
- `Liste<Node*> getNodes()`
  Gibt eine Liste aller Knoten des Graphen zurück.

Attribute

- `Node** nodes`
  Array von Knoten mit dem Bezeichner `nodes`, in dem alle Knoten des Graphen gespeichert sind.

Node:

Methoden

- Getter- und Setter-Methoden, um die Attribute in der Klasse `Node` zu setzen und auszulesen:
  - `std::string getKey(void)`
  - `int getPositionX(void)`
  - `int getPositionY(void)`
  - `Liste<Edge*> getEdges(void)`
  - `void setKey(std::string new_key)`
  - `void setPositionX(int x)`
  - `void setPositionY(int y)`
  - `void setNewEdge(Edge * edge)`

Attribute

- `std::string node_key`
  Attribut zur eindeutigen Identifikation des Knotens.
- `int position_x` und `int position_y`
  Attribute für die Position des Knotens zur späteren Darstellung in der `show` Methode im Graphen.
- `Liste<Edge*> edges`
  Attribut für die Adjazenzliste im Knoten, in der die Adressen aller ausgehenden Kanten gespeichert sind.

Edge:

Methoden

- Getter- und Setter-Methoden, um die Attribute in der Klasse `Edge` zu setzen und auszulesen:
  - `float getWeight(void)`
  - `Node * getStartNode(void)`
  - `Node * getEndNode(void)`
  - `void setWeight(float w)`
  - `void setStartNode(Node * n)`
  - `void setEndNode(Node * n)`
- Getter-Methoden: `getWeight()`, `getStartNode()`, `getEndNode()`

Attribute

- `Node *startnode` und `Node *endnode`
  Zum Speichern der Adressen der verbundenen Knoten.
- `float weight`
  Zum Speichern des Kantengewichts.

Erstellen Sie für jede Klasse geeignete Konstruktoren und Destruktoren.

Liste:

Methoden

- `unsigned int size(void)`
  Gibt die Anzahl der in der Liste vorhandenen Elemente zurück.
- `Liste(const Liste &old)`
  Kopierkonstruktor für die Liste.

Teil 2 (Darstellung): Damit der Graph in einer anschaulichen Form auf dem Bildschirm ausgegeben werden kann, wird in diesem Praktikum die externe Bibliothek SFML verwendet. Die Einrichtung von SFML auf ihrem Privatsystem, zur Vorarbeit für das Praktikum, können Sie den allgemeinen Praktikumsinformationen entnehmen.

Um die Darstellung des Graphen lose gekoppelt von der Graph-Implementierung zu halten, erstellen Sie die Klasse `SFMLGraphVisualizer`:

Attribute

- `sf::RenderWindow window;`
  Attribut vom SFML Fensterdatentyp zur Visualisierung des gezeichneten Graphen.

Methoden

- `void visualize(DiGraph &graph)`
  Zeichnet den Graphen in ein neues Fenster `img`

Wenn Sie die Funktionen gerne selbst implementieren möchten, können Sie hierzu die SFML Dokumentation unter `http://www.sfml-dev.org` nutzen. Eine Beispiel Implementierung, die allerdings noch vervollständigt werden muss, finden Sie im Anhang A.

Teil 3 (Hauptprogramm): Erstellen Sie ein Hauptprogramm, welches einen beliebigen Graphen (z.B. den Graphen aus Abb. 1a) erzeugt und auf dem Bildschirm ausgibt.

3. Testat

Voraussetzung ist jeweils ein fehlerfreies, korrekt formatiertes Programm. Der korrekte Programmlauf muss nachgewiesen werden. Sie müssen in der Lage sein, Ihr Programm im Detail zu erklären und ggf. auf Anweisung hin zu modifizieren.

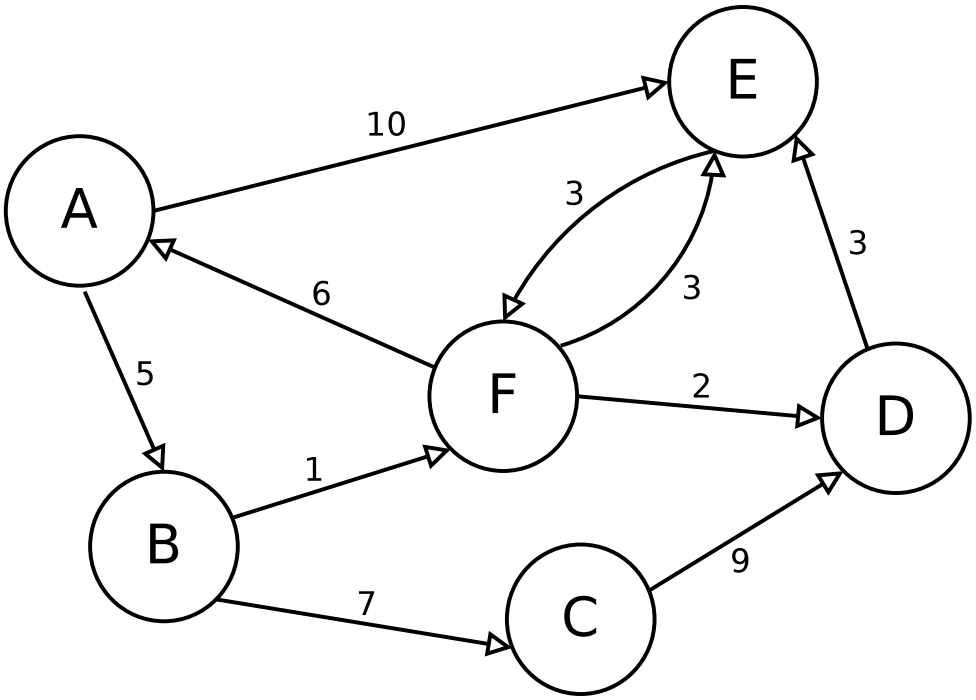
Das Praktikum muss spätestens zu Beginn des nächsten Praktikumtermins vollständig bearbeitet und abtestiert sein.

A. Visualisierung des Graphen mit SFML

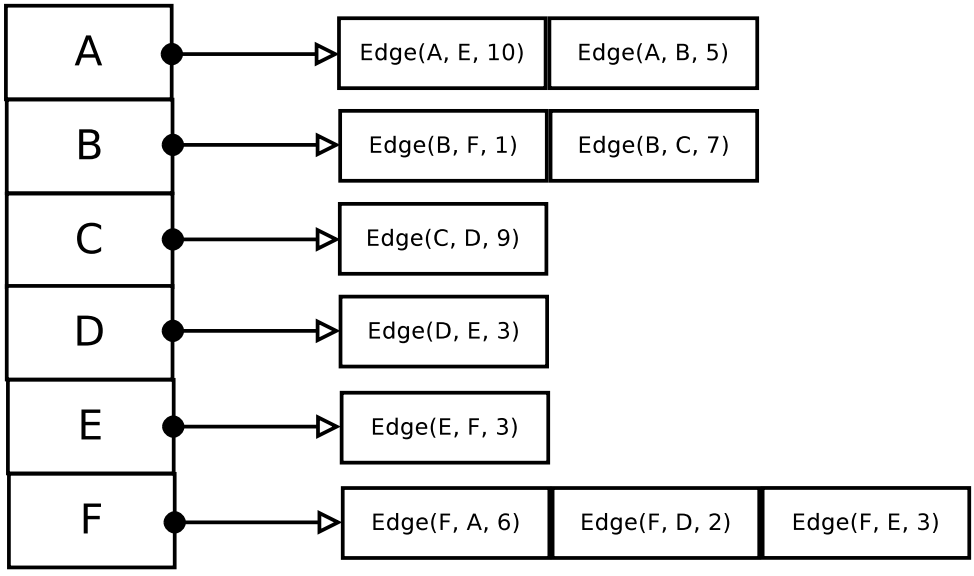
```
#include <SFML/Graphics.hpp>
#include <SFML/Window.hpp>
#include <math.h>
#include <sstream>

#define NODERADIUS 30
using namespace std;


class SFMLGraphVisualizer {
private:
    sf::RenderWindow window;
    sf::Font font;

public:

    SFMLGraphVisualizer() {
        //load my font
        this->font.loadFromFile("arial.ttf");
    }

    void visualize(DiGraph &g) {

        window.create(sf::VideoMode(1024, 768), "myGraph");

        Liste<Node *> nodes = g.getNodes();

        while (window.isOpen()) {
            // Process events
            sf::Event event;

            while (window.pollEvent(event)) {
                // Close window: exit
                if (event.type == sf::Event::Closed)
                    window.close();
            }

            // Clear screen
            window.clear(sf::Color::White);
            for (unsigned int i = 0; i < nodes.size(); i++) {
                Node *node = nodes.getValueAt(i);

                sf::Color color(255, 0, 0);
                drawNode(*node, color);

                // Ausgehende Kanten einzeichnen
                Liste<Edge *> edges = g.getEdges(node->getKey());

                for (unsigned int j = 0; j < edges.size(); j++) {
                    drawEdge(*(edges.getValueAt(j)), sf::Color::Black,
                      edges.getValueAt(j)->getWeight(), 1);
```

```
52                  }
53              }
54              // Update the window
55              window.display();
56          }
57      }
58
59
60      void drawNode(Node &node, sf::Color nodeColor) {
61
62          sf::CircleShape Circle(NODERADIUS);
63          Circle.setPosition(node.getPositionX() - NODERADIUS,
64            node.getPositionY() - NODERADIUS);
65          Circle.setFillColor(sf::Color::White);
66          Circle.setOutlineColor(nodeColor);
67          Circle.setOutlineThickness(5);
68
69          window.draw(Circle);
70
71          sf::Text Label(node.getKey(), font, 32);
72          Label.setPosition(node.getPositionX() - NODERADIUS / 2 + 5,
73            node.getPositionY() - NODERADIUS / 2 - 5);
74          Label.setFillColor(sf::Color::Blue);
75
76          window.draw(Label);
77      }
78
79
80      void drawEdge(Edge &edge, sf::Color color, double weight,
81                    int thickness = 1, int arrowMagnitude = 20) {
82
83          sf::Vector2f p(edge.getStartNode()->getPositionX(),
84            edge.getStartNode()->getPositionY());
85          sf::Vector2f q(edge.getEndNode()->getPositionX(),
86            edge.getEndNode()->getPositionY());
87
88
89          // Berechne den Winkel
90          const double PI = 3.141592653;
91          double angle = atan2((double) p.y - q.y, (double) p.x - q.x);
92
93          // Berechne verkuerzten Pfeil
94          q.x = (int) (q.x + NODERADIUS * cos(angle));
95          q.y = (int) (q.y + NODERADIUS * sin(angle));
96          p.x = (int) (p.x - NODERADIUS * cos(angle));
97          p.y = (int) (p.y - NODERADIUS * sin(angle));
98
99          sf::Vertex line[2] =
100                 {
101                         sf::Vertex(sf::Vector2f(p.x, p.y), color),
102                         sf::Vertex(sf::Vector2f(q.x, q.y), color)
103                 };
104
```

```
105         // thickness
106         window.draw(line, 2, sf::Lines);
107 
108         std::stringstream weightstring;
109         weightstring << weight;
110         sf::Text Labelweight(weightstring.str(), font, 32);
111         int size = sqrt(pow(p.x - q.x, 2) + pow(p.y - q.y, 2));
112         Labelweight.setPosition(p.x - (size / 2)*cos(angle)+10*sin(angle),
113                                 p.y - (size / 2)*sin(angle)+10*cos(angle));
114         Labelweight.setFillColor(sf::Color::Blue);
115         window.draw(Labelweight);
116 
117         //Erstes Segment
118         p.x = (int) (q.x + arrowMagnitude * cos(angle + PI / 8));
119         p.y = (int) (q.y + arrowMagnitude * sin(angle + PI / 8));
120         sf::Vertex first[2] =
121                 {
122                         sf::Vertex(sf::Vector2f(p.x, p.y), color),
123                         sf::Vertex(sf::Vector2f(q.x, q.y), color)
124                 };
125         window.draw(first, 2, sf::Lines);
126 
127 
128         //Zweites Segment
129         p.x = (int) (q.x + arrowMagnitude * cos(angle - PI / 8));
130         p.y = (int) (q.y + arrowMagnitude * sin(angle - PI / 8));
131         sf::Vertex second[2] =
132                 {
133                         sf::Vertex(sf::Vector2f(p.x, p.y), color),
134                         sf::Vertex(sf::Vector2f(q.x, q.y), color)
135                 };
136         window.draw(second, 2, sf::Lines);
137     }
138 };
```



(a) Gerichteter, gewichteter Graph



(b) dazugehörige Adjazenzliste

Abbildung 1: Datenstruktur “DiGraph”