Praktikum 3: Klassen im Zusammenspiel

1. Lernziele

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

- Objektorientierte Programmierung
- Klassen- und Objektmodell
- Graph-Visualisierung mit *Graphviz*

2. Aufgabe

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

Teil 1 (Datenstruktur): Graphen (siehe Abbildung 1) 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.

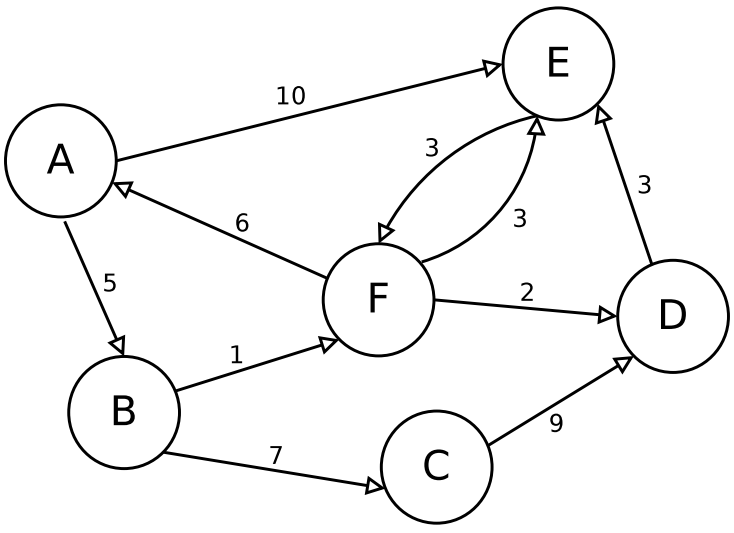

Abbildung 1: Gerichteter, gewichteter Graph

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

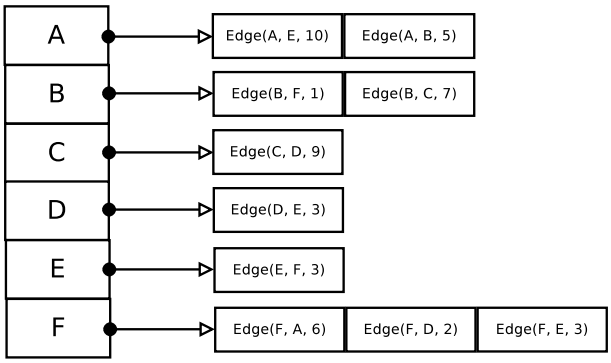

Abbildung 2: Adjazenzliste

Die gängigste Form einen Graphen in einem Programm darzustellen ist die sogenannte Adjazenzliste (siehe Abbildung 2). 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 Zeigern auf 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)`
    Liefert den Namen eines Knoten.
  - `Liste<Edge*> getEdges(void)`
    Gibt die Liste aller dem Knoten zugeordneten Kanten zurück.
  - `std::string setKey(void)`
    Setzt den Namen eines Knoten.
  - `void setNewEdge(Edge * edge)`
    Fügt dem Knoten eine neue Kante hinzu.

Attribute

- `std::string node_key`
  Attribut zur eindeutigen Identifikation des Knotens.
- `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)`

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 das Tool `Graphviz` verwendet. Details zu diesem Tool hierzu finden Sie unter `http://graphviz.org`.

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

Methoden

- `void visualize(DiGraph &graph)`
  Gibt den übergebenen Graphen auf der Konsole in der `DOT`-Notation aus. Beispiele hierzu finden Sie im Anhang in den Abbildungen 3 bis 5.

Teil 3 (Hauptprogramm): Erstellen Sie ein Hauptprogramm, welches einen beliebigen Graphen (z.B. den Graphen aus Abb. 1) erzeugt und in der `DOT`-Notation auf dem Bildschirm ausgibt.

Zur Kontrolle geben Sie die Ausgabe Ihres Programmes per *Copy & Paste* in das DOT-Online Tool (`https://s.iot3.de/dot`) ein.

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 Graphviz

```
digraph {

  edge [len=2];

  "A" [color=green];
  "D" [color=red];

  "A"   -> "B"        [label=5, color=orange];
  "B"   -> "C"        [label=7];
  "C"   -> "D"        [label=9];
  "A"   -> "E"        [label=10];
  "D"   -> "E"        [label=3];

  "F"   -> "A"        [label=6];
  "F"   -> "E"        [label=3];
  "E"   -> "F"        [label=3];
  "F"   -> "D"        [label=2, color=orange];
  "B"   -> "F"        [label=1, color=orange];
}
```

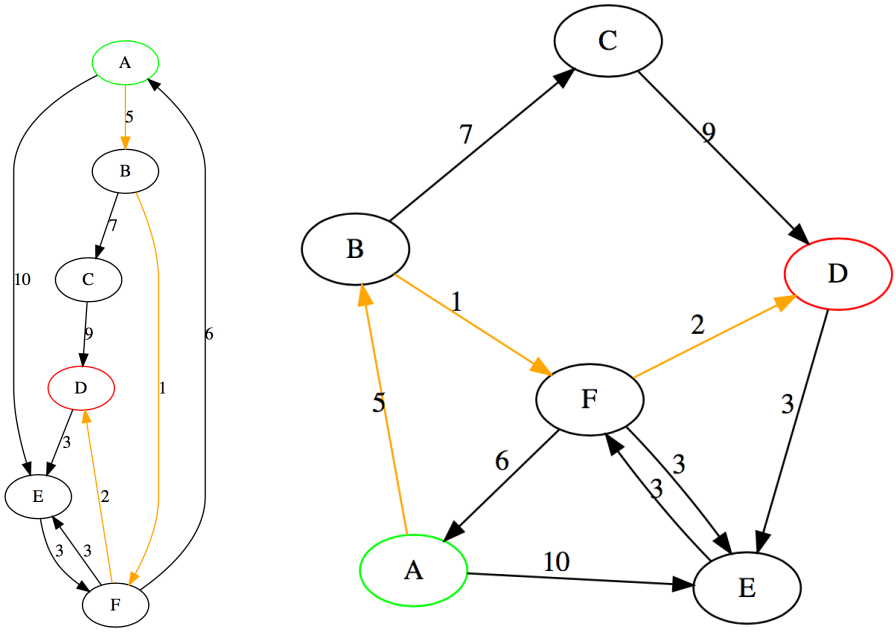

Abbildung 3: DOT - generierte Graphen

The DOT Language

The following is an abstract grammar defining the DOT language. Terminals are shown in bold font and nonterminals in italics. Literal characters are given in single quotes. Parentheses (and) indicate grouping when needed. Square brackets [and] enclose optional items. Vertical bars | separate alternatives.

graph : [**strict**] (**graph** | **digraph**) [*ID*] '{' *stmt_list* '}'
stmt_list : [*stmt* ['**;**'] *stmt_list*]
stmt : *node_stmt*
| *edge_stmt*
| *attr_stmt*
| *ID* '**=**' *ID*
| *subgraph*
attr_stmt : (**graph** | **node** | **edge**) *attr_list*
attr_list : '[' [*a_list*] ']' [*attr_list*]
a_list : *ID* '**=**' *ID* [('**;**' | '**,**')] [*a_list*]
edge_stmt : (*node_id* | *subgraph*) *edgeRHS* [*attr_list*]
edgeRHS : *edgeop* (*node_id* | *subgraph*) [*edgeRHS*]
node_stmt : *node_id* [*attr_list*]
node_id : *ID* [*port*]
port : '**:**' *ID* ['**:**' *compass_pt*]
| '**:**' *compass_pt*
subgraph : [**subgraph** [*ID*]] '{' *stmt_list* '}'
compass_pt : (**n** | **ne** | **e** | **se** | **s** | **sw** | **w** | **nw** | **c** | **_**)

Abbildung 4: Syntax der DOT-Sprache

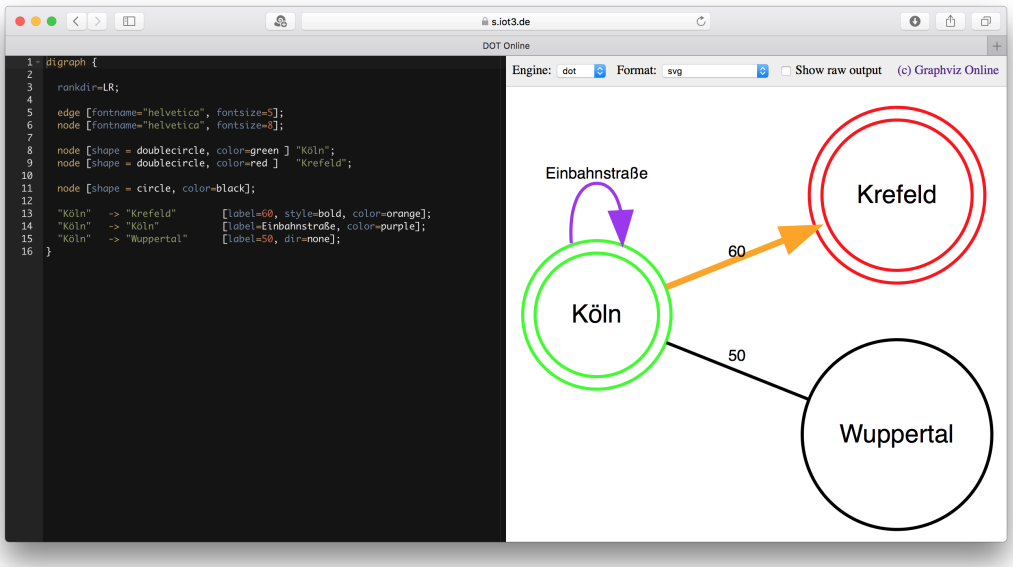

Abbildung 5: Online DOT Visualizer – `https://s.iot3.de/dot`