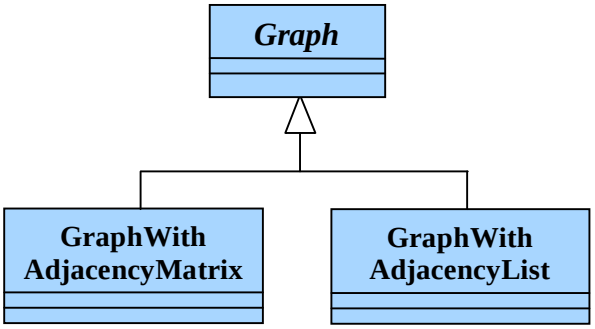
Aufgabenblatt 2

1. Klasse Graph

Die Klasse Graph soll Operationen für Graphen anbieten, so dass sich die in der Vorlesung in Pseudo-Code besprochenen Algorithmen direkt und einfach umsetzen lassen. Die Graphen können gewichtet oder ungewichtet, gerichtet oder ungerichtet sein.

Mit Schnittstellenvererbung sollen Graphen einerseits mit Adjazenzmatrizen und andererseits mit Adjazenzlisten realisiert werden.



Die Knoten (Vertex) sind nummeriert: 0, 1, 2, …, n-1. Gewichte sollen ganzzahlig sein. Ungewichtete Graphen können als gewichtete Graphen aufgefasst werden, wobei jede Kante das Gewicht 1 hat. Es sind Methoden vorzusehen, die alle Nachbarknoten zu einem Knoten zurückliefern. Dabei ist es zweckmäßig auch die Gewichte der Kanten zu den Nachbarknoten zurückzuliefern. Folgende Definitionen sind daher hilfreich:

```
typedef int Vertex;   // Knoten
typedef int Weight;   // Gewichte

struct VertexW        // Knoten mit Gewichten
{
     VertexW(int v = 0, int w = 0) : vtx(v), wgt(w) {}
     Vertex vtx;
     Weight wgt;
};
```

Abstrakte Klasse Graph

Sehen Sie folgende Methoden vor:

- int getSize() const;
  liefert die Anzahl der Knoten zurück. Die Knoten sind nummeriert: 0, 1, ..., size-1
- bool isDirected() const;
  Liefert true zurück, falls der Graph gerichtet ist und sonst false. Die Eigenschaft, ob ein Graph gerichtet ist, wird im Konstruktor der abgeleiteten Klassen festgelegt.
- bool insert(Vertex u, Vertex v, Weight w = 1);
  Fügt eine Kante (u,v) in den Graphen ein. Bei einem ungerichteten Graphen wird zusätzlich die Kante (v,u) eingefügt. w ist das Gewicht der Kante. Bei einem ungewichteten Graphen ist w = 1. Falls die Kante bereits existiert, wird die Kante nicht eingefügt und false zurückgeliefert, sonst true.

- bool remove(Vertex u, Vertex v);
  Löscht die Kante (u,v). Bei einem ungerichteten Graphen wird zusätzlich die Kante (v,u) gelöscht.Falls die Kante nicht existiert, wird false und sonst true zurueckgeliefert.
- bool getW(Vertex u, Vertex v, Weight& w) const;
  Liefert das Gewicht w der Kante (u,v) zurück. Falls die Kante nicht existiert, wird false und sonst true zurückgeliefert.
- bool isConnected(Vertex u, Vertex v) const;
  Liefert true zurück, falls die Kante (u,v) existiert und sonst false.
- bool setW(Vertex u, Vertex v, Weight w);
  Setzt das Gewicht der Kante (u,v) auf w. Bei einem ungerichteten Graphen wird zusätzlich das Gewicht der Kante (v,u) auf w gesetzt. Falls die Kante nicht existiert, wird false und sonst true zurückgeliefert.
- int read(const string& filename);
  liest eine Folge von Kanten (Knotenpaare) mit Gewichten von der Datei filename und fügt die Kanten in den Graph ein.
- int getAllNbr(int v, vector<VertexW>& ws) const;
  int getAllSucc(int v, vector<VertexW>& ws) const;
  int getAllPred(int v, vector<VertexW>& ws) const;

  getAllNbr liefert zu Knoten v alle Nachbarknoten ws. Bei gerichteten Graphen kann Nachbar Nachfolger oder Vorgänger sein.
  getAllSucc liefert zu Knoten v alle Nachfolgerknoten ws.
  getAllPred liefert zu Knoten v alle Vorgängerknoten ws.
  Rückgabewert ist die Anzahl der Knoten.
  Bei einem ungerichteten Graphen stimmen Nachbarn, Vorgänger und Nachfolger überein.
  Für jeden Knoten w, der zurückgeliefert wird, wird noch das Gewicht der Kante (v,w) zurückgeliefert.

Klasse GraphWithAdjacencyMatrix

Der Graph mit seinen Gewichten wird als zweidimensionales Feld realisiert. Der Konstruktor hat folgende Schnittstelle:

- GraphWithAdjacencyMatrix(int n, bool d = false);
  n legt die Anzahl der Knoten fest und d gibt an, ob der Graph gerichtet ist.

Klasse GraphWithAdjacencyList

Für jeden Knoten werden seine Nachbarn als Liste abgespeichert (Adjazenzliste). Besonders einfach wird die Implementierung, wenn die Adjazenzlisten als STL-Maps realisiert werden.

Beachten Sie, dass bei einem gerichteten Graphen Vorgänger- und Nachfolgerknoten in getrennten Listen gespeichert werden.

Der Konstruktor hat folgende Schnittstelle:

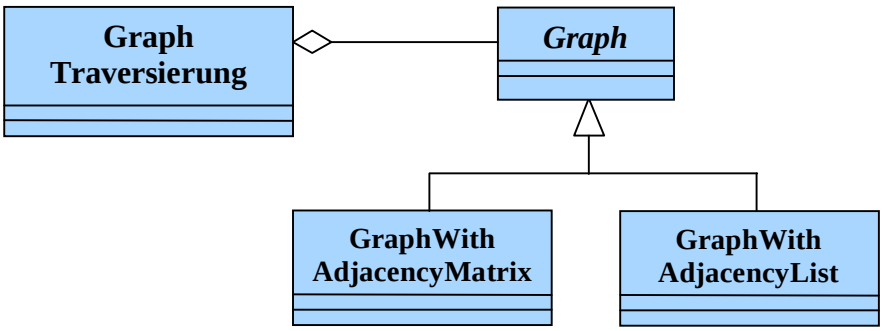
- GraphWithAdjacencyList(int n, bool d = false);
  n legt die Anzahl der Knoten fest und d gibt an, ob der Graph gerichtet ist.

Hauptpogramm

Testen Sie Ihre Klassen ausgiebig in einem Hauptprogramm.

2. Tiefensuche und Breitensuche

Die Klasse GraphTraversierung erhält einen Zeiger auf einen Graphen und besitzt die erforderlichen Datenstrukturen, um in dem Graphen eine Tiefen- und Breitensuche durchzuführen.



Die Zusammenarbeit zwischen der Klasse GraphTraversierung und der Klasse Graph ist eine Umsetzung des Prinzips der Delegation (siehe Vorlesung Programmiertechnik 2).

Sehen Sie folgende Methoden vor:

- GraphTraversierung(Graph* g = 0);
- ~GraphTraversierung();
- void setGraph(Graph* g);
- void depthFirstSearch(Vertex v);
  Besucht von Knoten v aus alle Knoten mit Tiefensuche. Die besuchten Knoten werden ausgegeben.
- void breadthFirstSearch(Vertex v);
  Besucht von Knoten v aus alle Knoten mit Breitensuche. Die besuchten Knoten werden ausgegeben.

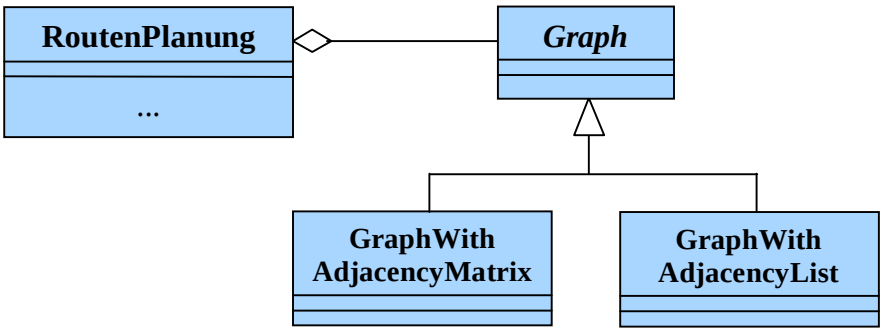
Testen Sie Ihre Algorithmen mit dem Beispiel aus der Vorlesung (Seite 2-25).

3. Routenplanung in Städten

Der Spielplan des Spiels „Scottland Yard“ (Ravensburger; Spiel des Jahres 1983) besteht aus einer Menge von 199 Knoten (Punkten) in London, die durch Taxi, Bus oder U-Bahn verbunden sind.

Es soll eine Klasse RoutenPlanung geschrieben werden, die den billigsten (kürzesten Weg) zwischen 2 Knoten nach dem Algorithmus von Dijkstra berechnet. Die Klasse RoutenPlanung benutzt

dazu ein Objekt der Klasse Graph und verwaltet die für den Algorithmus notwendigen Datenstrukturen.



Sehen Sie folgende Methoden vor:

- RoutenPlanung(Graph* g = 0);
- ~RoutenPlanung();
- void setGraph(Graph* g);
- bool shortestPath(Vertex s, Vertex z, list<Vertex>& path, Weight& path_w);
  Berechnet den kürzesten Weg path von s nach z und seine Kosten path_w.
  Liefert true zurück, falls es einen kürzesten Weg gibt und sonst false.

Die Verbindungsdaten des Spiels „Scottland Yard“ liegen in einer Datei in folgendem Format vor:

Startknoten	Endknoten	Verkehrsmittel
1	9	Taxi
1	46	U-Bahn
1	58	Bus

Die Kosten für die einzelnen Verbindungen (Kanten) können wie folgt angenommen werden:

U-Bahn:	7
Taxi	3
Bus	2

Da „Taxi“-Kanten“ relativ kurz sind, haben sie vergleichsweise geringe Kosten.

Beachten Sie, dass es Knotenpaare gibt, die unterschiedliche Verbindungen haben. In diesem Fall soll die billigere Verbindung verwendet werden.

Transformieren Sie die Scottland-Yard-Datei mit einem Texteditor in das Dateiformat für gewichtete Graphen.