

Praktikum 5: Delegation

1. Lernziele

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

- Ableitung von Klassen
- Abstrakte Klassen
- Polymorphie
- Delegation

2. Aufgabe

Teil 1: Erweitern Sie den Graphen aus Praktikum 3 bzw. 4 um eine Funktion, die für einen gegebenen Start- und Endknoten den kürzesten Pfad berechnet und zurückgibt. Der Funktionsprototyp soll wie folgt aussehen:

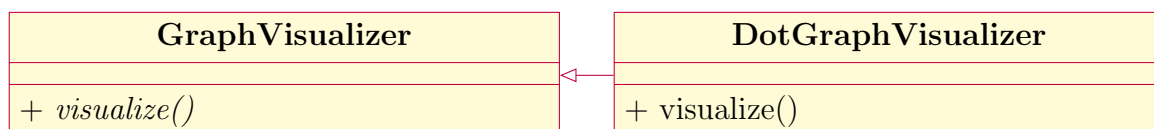
```
Liste<Edge*> dijkstraShortestPath(std::string start, std::string end)
```

Die Funktion gibt also eine Liste zurück, deren Elemente Zeiger auf diejenigen Kanten (`Edge*`) sind, die den kürzesten Pfad zwischen zwei ausgewählten Knoten (`Node`) bilden.

Erweitern Sie hierzu die Klasse `DiGraph` um entsprechende Attribute, welche den Status der Berechnung des *Kürzesten Pfades* und eben diesen Pfad speichern.

Die Funktionsweise des Dijkstra-Algorithmus können Sie dem gegebenen Algorithmus auf Seite 3 oder dem Pseudocode auf Seite 4 im Anhang entnehmen. Verwenden Sie für die Datenstruktur *PQ*, die in Praktikum 2 entwickelte Vorrangwarteschlange mit dem Datentyp `Node*` als Template.

Teil 2: Um die Visualisierung des Graphen austauschbar zu machen, erweitern Sie die Klasse `DotGraphVisualizer` um eine abstrakte Basisklasse `GraphVisualizer`.



Fügen Sie der Klasse `DiGraph` die folgenden Attribute und Methoden hinzu:
Attribute

- `GraphVisualizer* graphviz`
Instanzobjekt der `GraphVisualizer`-Klasse

Methoden

- `void setVisualizer(GraphVisualizer* graphviz)`
- `GraphVisualizer* getVisualizer()`

Teil 3: Erweitern Sie die Funktion `visualize()` um folgende Hervorhebungen des kürzesten Pfades im Graphen zu erzielen:

- Die Kanten des kürzesten Weges sollen farblich hervorgehoben werden.
- Die Verbindungslinien zwischen Knoten, die nicht auf dem kürzesten Pfad liegen, sollen keine Pfeile, sondern Linien ohne Pfeilspitzen sein.
- Die Start- und Endknoten des kürzesten Pfades sollen ebenfalls hervorgehoben werden (Form/Farbe).

Die Ausgabe erfolgt wie in Praktikum 3 als **DOT**-Datei auf der Konsole, die per *Copy & Paste* in das DOT-Online Tool (<https://s.iot3.de/dot>) zur Kontrolle/Darstellung des Graphen eingefügt werden kann.

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.

Sie erhalten zu Beginn des Praktikums eine Datei (siehe Praktikum 4), die den Graphen enthält, für den Sie den kürzesten Pfad bestimmen sollen.

4. Anhang

Algorithm 1: DIJKSTRASHORTESTPATH finds the shortest path between two nodes in a graph with non-negative weights

Input: A graph $G = (V, E)$ as ordered pair of vertices and edges, a start node $s \in G$, an end node $d \in G$

Output: The shortest path from s to d in G

```
1  $PQ \leftarrow \{\}$ 
2 for each  $v \in G$  do
3   if  $v \neq s$  then
4      $dist[v] \leftarrow \infty$ 
5   else
6      $dist[v] \leftarrow 0$ 
7      $previous[v] \leftarrow null$ 
8      $PQ.insert(v, dist[v])$ 
9 while  $PQ$  is not empty do
10    $u \leftarrow PQ.extractMin()$ 
11   for each neighbor  $v$  of  $u$  do
12      $alt \leftarrow dist[u] + weight(u, v)$ 
13     if  $alt < dist[v]$  then
14        $dist[v] \leftarrow alt$ 
15        $previous[v] \leftarrow u$ 
16        $PQ.decreaseKey(v, alt)$ 
17  $S \leftarrow []$ 
18  $u \leftarrow d$ 
19 while  $u$  is not null do
20   insert  $u$  at the beginning of  $S$ 
21    $u \leftarrow previous[u]$ 
22 return  $S$ 
```

```
1 ALGORITHMUS Dijkstra
2
3 BESCHREIBUNG:
4 Der Dijkstra-Algorithmus sucht in einem Graphen den kürzesten Weg zwischen zwei Knoten
5
6 PARAMETER IN:
7     Einen Graphen: G
8     Startknoten: s
9     Endknoten: d
10    Array mit Knoten: nodes
11    Anzahl der Knoten im Graph: used
12
13 PARAMETER OUT:
14    Liste mit kürzestem Weg von s nach d: result
15
16
17 KERNALGORITHMUS
18
19    Neue Liste result aus Kanten anlegen
20
21    Neue Warteschlange (PriorityQueue) 'PQ' aus Knotenelementen anlegen
22
23    float dist[used]
24
25    Node* previous[used];
26
27    FUER i = 0 BIS used SCHRITTWEITE 1
28
29        FALLS nodes[i] == s DANN
30
31            dist[i] = 0
32
33        SONST
34            dist[i] = unendlich
35
36        ENDE FALLS
37
38        previous[i] = NULL
39
40        Füge der Warteschlange den Wert nodes[i] mit Priorität dist[i] hinzu.
41
42    ENDE FUER
43
44    SOLANGE ( PQ nicht leer ist)
45
46        u = Element mit höchster Priorität aus PQ
47
48        Lege Array outEdges an mit allen ausgehenden Kanten von u an
49
50        size = Anzahl der Elemente in outEdges
51
52        FÜR i = 0 BIS size SCHRITTWEITE 1
53
```

```
54         Node *v = Endknoten von outEdges[i]
55
56         float alt = dist[u] + Gewicht von outEdges[i]
57
58         FALLS alt < dist[v] DANN
59
60             dist[v] = alt
61
62             previous[v] = u
63
64             Ändere die Priorität von v in der Warteschlange auf alt
65
66         ENDE FALLS
67
68     ENDE FÜR
69
70     ENDE SOLANGE
71
72     u = d
73
74     SOLANGE( previous[u] != NULL)
75
76         Die Kante von previous[u] nach u der Liste result hinzufügen
77
78         u = previous[u]
79
80     ENDE SOLANGE
81
82     Rückgabe von result
83
84 ENDE ALGORITHMUS
```