Praktikumsaufgabe 4

Straßenkarten als Graphen[1]

Formalia:

1. Die Lösung zu dieser Praktikumsaufgaben besteht aus mehreren Modulen und xml-Dateien. Alle diese Dateien speichern Sie bitte in einem **Verzeichnis**, z.B.:

   `P04NameVorname`

2. Den bereitgestellten Unit-Test `p04unit.py` kopieren Sie bitte in Ihr Lösungsverzeichnis.

   Wenn Sie sich an die vorgegebenen Namen (insbesobdere Modul-Namen) gehalten haben, können Sie dann diesen Unit-Test aus diesem Verzeichnis entweder im Anaconda-Prompt mit der Anweisung

   ```
   python p04unit.py
   ```

   oder im Spyder ausführen.

3. **Erst wenn Ihre Lösung den Unit-Test besteht, können Sie Ihre Lösung in der Zoom-Sitzung (gleicher Link wie Online-Sprechstunde/Vorlesung) abnehmen lassen**.

4. Falls Sie selber Ihre Fehler im Unit-Test nicht beheben können, haben Sie in der Zoom-Sitzung zum Praktikum die Möglichkeit, die Fehler mit den anderen TeilnehmerInnen im Breakout-Raum zu besprechen und sich helfen zu lassen.

5. Falls Sie in der Breakout-Raum-Gruppe nicht weiter kommen rufen Sie Dozenten-Hilfe. Beachten Sie dass aufgrund der Anzahl der Breakout-Räume es etwas dauern kann, bis die *Dozenten-Hilfe* kommt. **Bringen Sie deshalb Geduld und Verständnis mit in die Praktikums-Zoom-Sitzung**.

6. **Die Abnahme Ihrer Praktikums-Lösung kann nur in der für Ihre Gruppe stattfindenden Zoom-Sitzung (gleicher Link wie Online-Sprechstunde/Vorlesung) erfolgen. Die Teilnahme an allen Praktikums-Zoom-Sitzungen für Ihre Gruppe ist Pflicht für den Erhalt der Bonus-Punkte.**

7. Sollten Sie in der vorgesehenen Zoom-Sitzung die Abnahme nicht schaffen haben Sie **am Ende** der nächsten Zoom-Sitzung für Ihre Gruppe die Möglichkeit eine zurückliegende Praktikums-Aufgabe abnehmen zu lassen.

8. Die Gruppeneinteilung und die Termine für die Zoom-Praktikums-Sitzungen werden über den Moodle-Kurs bekannt gegeben.

Bem.: Die Praktikums-Anleitung ist immer als ein Pflichtenheft zu verstehen. Deshalb **müssen alle vorgegebenen Namen und die Anzahl und Art der Parameter von Funktionen/Methoden im Code genau übernommen werden und dürfen nicht willkürlich geändert werden!**
Lesen Sie die ganze Anleitung **mehrfach genau durch** und **überlegen** Sie sich ggf. auf Papier die algorithmische Lösung bevor Sie mit dem Codieren beginnen!

[1] In Anlehnung an die Praktikums-Aufgaben der Kollegen Schöttl und Tasin aus dem SoSem 2018.

Bem. zur Objektorientierung und den UML-Klassen-Diagrammen:

a) In dieser Praktikumsaufgabe besitzen viele Attribute ein öffentliches (public) Zugriffsrecht. Dies widerspricht dem Data-Hiding, das eine Säule der Objektorientierung ist, und wurde nur aus Konsistenzgründen zu parallelen Kursen beibehalten.

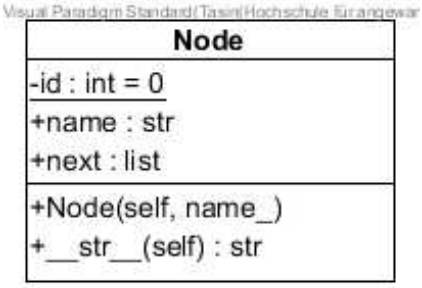
b) Die UML-Klassen-Diagramme in dieser Praktikums-Aufgabe verwenden die Python Typen und in den Methoden wird das Python-spezifische Schüsselwort `self` verwendet. Dies geschieht ebenfalls aus Konsistenzgründen zu parallelen Kursen. UML-Diagramme sollten eigentlich Programmiersprachen unabhängig sein!

Eine Straßenkarte kann als Graph dargestellt werden. Ein Graph besitzt Knoten (Klasse `Node`) und Kanten (Klasse `Edge`), die die Knoten verbinden. Um eine Straßenkarte als Graph darzustellen, verwendet man die Knoten des Graphen für die Straßenkreuzungen (und Einmündungen). Die Kanten des Graphens stellen die Straßen zwischen den Kreuzungen dar. Der Einfachheit halber werden wir im Folgenden nur gerichtete Kanten, die genau einen Startknoten und einen Zielknoten besitzen, zur Darstellung von Straßen verwenden. Diese können als Einbahnstraßen interpretiert werden.
In der Informatik verwendet man je nach Anwendung unterschiedliche Datenstrukturen, um einen Graphen im Rechner zu speichern. Die gängigsten sind die Knotenliste, die Kantenliste, die Adjazenzmatrix und die Adjazenzliste. Hier verwenden wir die Adjazenzliste, dabei speichert jeder Knoten alle seine ausgehenden Kanten.
Da wir nur gerichtete Kanten (jede Kante verläuft immer von einem Startknoten zu einem Zielknoten, nicht aber zurück) betrachten, ist jede ausgehende Kante auch durch ihren Zielknoten eindeutig bestimmt.
Im ersten Schritt reicht es deshalb aus, anstelle der ausgehenden Kante (für die später eine eigene Klasse `Edge` ertsellt wird) zunächst nur den Zielknoten (also den Nachfolgeknoten) zu speichern.
Sowohl die Knoten als auch die Kanten besitzen einen unveränderlichen Namen, die Kanten zusätzlich Gewichte (die der Weglänge entsprechen). Auch weitere Informationen (wie. z. B. die Art der Straße, Geschwindigkeitsbeschränkungen etc.) könnten sinnvolle Attribute einer Kante sein.
Für die Arbeit mit Karten sind neben den genannten Daten auch eine Reihe von Operationen wichtig. Um die Übersicht zu behalten und den Nutzern eine einfache Programmierschnittstelle zu bieten, empfiehlt sich die Organisation in Klassen.

1. Die Klasse Node: (zur Darstellung der Knoten)

1.1. Implementieren Sie die Klasse Node, deren Struktur in folgendem Klassendiagramm abgebildet ist. Speichern Sie die Klasse im Modul `node1.py`.

`next` ist dabei eine Liste der Nachfolgeknoten. `id` eine ganze Zahl (Sie können diese statische Klassenvariable zunächst einfach auf 0 setzen).



Konstruieren Sie in der Methode `__str__` zuerst einen String, der die nötigen Informationen für den Knoten beinhaltet. Studieren Sie hierzu ganz genau die Ausgabe des unten angegebenen Test-Codes, der in einem weiteren Modul `test04.py` zu erstellen ist, um auch den Import von Klassen zu üben.

```
import node1
n1 = node1.Node('A')
n2 = node1.Node('B')
n3 = node1.Node('C')
n1.next.append(n2)
n1.next.append(n3)
print(n3.name)
print(n1)
print(n2)
```

```
C
A ---> B
  ---> C
B <end>
```

1.2 Kopieren Sie das Modul `node1.py` in das Modul `node2.py`. Modifizieren Sie in `node2.py` die Klasse Node der letzten Teilaufgabe so, dass `name` eine nur lesbare Eigenschaft (Property) (des entsprechenden privaten Attributs) darstellt (Tipp: Property ohne Setter). Importieren Sie nun im Modul `test04.py` das Modul `node2.py` und führen darin nochmals die obigen Tests für die neue Klasse Node (die alten Tests können Sie mittels eines Dokumentations-Kommentares auskommentieren) aus und zusätzlich den folgenden Test:

```
n4 = node2.Node('D')
print(n4.name)
n4.name = 'B'       # diese Zeile soll nun zu einem Fehler fuehren
```

1.3. Kopieren Sie nun das Modul `node2.py` in das Modul `node3.py`. Im Modul `node3.py` modifizieren Sie die Klasse Node der letzten Teilaufgabe, so dass es möglich ist, den Konstruktor auch ohne Angabe eines Knotennamens aufzurufen. In diesem Fall soll der (eindeutige) Standardname `"Knoten XXX"` verwendet werden, wobei `XXX` für die laufende Nummer des Knotens steht. Im Konstruktor muß deshalb nun die statische Klassenvariable `id` inkermentiert werden.

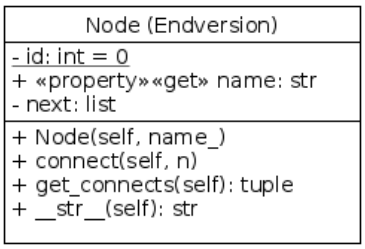
1.4. Modifizieren Sie die Klasse Node innerhalb von Modul `node3.py` der letzten Teilaufgabe so, dass das Hinzufügen eines Nachfolgeknotens nur noch über die Methode `connect` möglich ist. Kommentieren Sie im Modul `test04.py` alle bisherigen Tests aus, importieren Sie das Modul `node3.py` und führen Sie nun die folgenden Tests aus, die die nebenstehende Ausgabe ergeben sollen:

```
n1 = node3.Node()
n2 = node3.Node('B')
n3 = node3.Node()
n1.connect(n2)
n1.connect(n3)
print(n1)
print(n2)
```

```
Knoten 1 ---> B
         ---> Knoten 3
B <end>
```

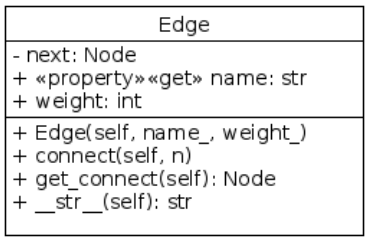
1.5. Ergänzen Sie im Modul `node3.py` die Klasse Node um die Methode `get_connects`, die die Listenelemente der Nachfolgeknoten als `tuple` zurückgibt.

Das endgültige Klassendiagramm der Klasse `Node` ist damit



2. Die Klasse Edge:

2.1. Implementieren Sie im Modul `edge.py` die Klasse `Edge`, deren Struktur in folgendem Klassendiagramm abgebildet ist.



`connect` setzt dabei den (maximal einen) Zielknoten der Kante (der Ausgangsknoten wird hier zunächst nicht betrachtet). `get_connect` gibt den Zielknoten zurück. `name` stellt wie in der Klasse `Node` eine nur lesbare Eigenschaft (Property) dar. Speichern Sie die Klasse im Modul `edge.py`. Der folgende Test, der in das Modul `test04.py` aufzunehmen ist nachdem alle anderen Tests auskommentiert wurden, soll die nebenstehende Ausgabe ergeben:

```
n1 = node3.Node('A')
e1 = edge.Edge('E', 5)
e1.connect(n1)
print('vorher:', e1)
e1.weight = 3
print('nachher:', e1)
```

```
vorher: E/5
nachher: E/3
```

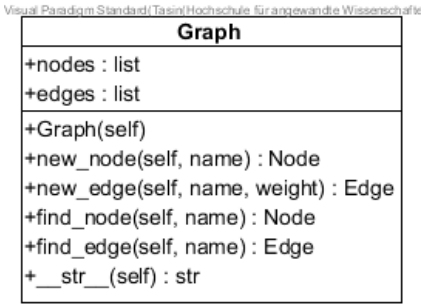
2.2. Kopieren Sie das Modul `node3.py` in das Modul `node.py`. Modifizieren Sie im Modul `node.py` die Klasse `Node`, so dass die Methode `connect` nun jeweils eine Kante mit dem Knoten verbindet. Der folgende Test, der in das Modul `test04.py` aufzunehmen ist nachdem alle anderen Tests auskommentiert wurden), soll die nebenstehende Ausgabe ergeben (also ist auch die Methode `__str__()` anzupassen):

```
n1 = node.Node('A')
n2 = node.Node('B')
n3 = node.Node('C')
e1 = edge.Edge('E', 5)
e2 = edge.Edge('F', 2)
n1.connect(e1)
n1.connect(e2)
e1.connect(n2)
e2.connect(n3)
print(n1)
print(n2)
```

```
A --E/5--> B
  --F/2--> C
B <end>
```

3. Die Klasse Graph:

3.1. Implementieren Sie eine Klasse `Graph`, die eine Liste von Knoten und eine Liste von Kanten als Attribute enthält, im Modul `graph.py`. Die Methoden `find_node(self, name)` und `find_edge(self, name)` werden unten im Abschnitt 3.2 genau beschrieben.



Der folgende Test, der in das Modul `test04.py` aufzunehmen ist nachdem alle anderen Tests auskommentiert wurden), soll die nebenstehende Ausgabe ergeben:

```
g = graph.Graph()
n1 = g.new_node('A')
n2 = g.new_node()
e1 = g.new_edge('E', 5)
n1.connect(e1)
e1.connect(n2)
print(g)
```

```
Knoten:
-------
A --E/5--> Knoten 2
Knoten 2 <end>

Kanten:
-------
E/5
```

3.2. Erweitern Sie die Klasse `Graph` um die Methoden `find_node(self, name)` (die das Knoten-Objekt mit dem angegebenen Namen zurückgibt) und `find_edge(self, name)` (die das Edge-Objekt mit dem angegebenen Namen zurückgibt). Testen Sie die Methoden, z. B. mit `print(g.find_node("Knoten 2"))`.

3.3. Unter einem Pfad versteht man eine Folge im Graph benachbarter Knoten. Erweitern Sie die Klasse `Graph` um die Methode `path_length(self, path_node_names)`, die die Knotennamen des Pfades als Liste akzeptiert und die Gesamtlänge des Pfades ermittelt. Existiert ein derartiger Pfad nicht, soll -1 zurückgegeben werden.

Tipp: Es kann sinnvoll sein, eine **statische Methode** `find_edge_between(n1, n2)` einzuführen, die im Graphen eine Kante (`Edge`-Objekt) sucht die die beiden `Node`-Objekte `n1` und `n2` verbindet. Falls eine derartige Kante nicht existiert gibt sie `None` zurück.

Testen Sie die Methode an dem folgenden Graphen, der als ASCII-Grafik dargestellt ist, im Modul `test04.py`, in dem alle anderen Tests wiederum auskommentiert werden.

```
      a/3   f/1
    A --> B <-- F
    ^     |
e/2 |     | b/7
    |     v
    D <-- C --> E
      c/5   d/4
```

Für

```
path_node_names = ["A", "B", "C", "D", "A"] bzw.
path_node_names = ["A", "B", "C", "A"] bzw.
path_node_names = ["A", "B", "C", "E"]
```

sollte die Methode `path_length()` die folgenden Ergebnisse liefern:
`17` bzw. `-1` bzw `14`.

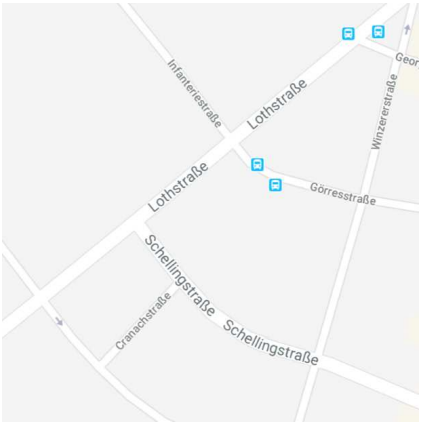
4. Klasse Reader und XML:

Die Ihnen zur Verfügung gestellte Klasse `Reader` im Modul `reader.py` implementiert eine Lesemethode für XML-Dateien, die einen Graphen in textueller Form darstellen. Reader besitzt dabei die Methode `read`, die als Argument den Pfad zur XML-Datei erwartet und den Graph als Instanz der Klasse Graph zurückgibt. `Reader` importiert dazu die Klasse Graph und nutzt deren Methoden und Attribute.

4.1. Betrachten Sie die beigefügten beispielhaften XML-Dateien und versuchen Sie, sie zu verstehen. Diese können mit einem normalen Text-Editor geöffnet werden, also auch mit dem Editor von Spyder.

Zeichnen Sie als ASCII-Grafik innerhalb eines Dokumentations-Kommentares im Modul `test04.py` den Graphen von `Plan.xml`.

Erstellen Sie eine eigene XML-Datei, die den folgenden Ausschnitt einer Straßenkarte repräsentiert. (Die Kreuzungen können beliebig benannt werden. Es reicht, Einbahnstraßen zu modellieren; achten Sie aber auf die Möglichkeit, von der Kreuzung Winzererstr./Görresstr. zu beliebigen Stellen der Karte zu gelangen.) Speichern Sie die Karte unter dem Namen `Karte.xml`. Die Gewichte sind dabei die Längen der Teilstrecken in Meter, die frei (aber vernünftig) gewählt werden können (z.B. in Google-Maps nachmessen).



4.2. Schreiben Sie im Modul `test04.py` ein Programm, das `Karte.xml` einliest und eine Liste der Knoten und Kanten (wie in Teilaufgabe 3.1. definiert) ausgibt.

4.3. *Für Profis:* Implementieren Sie eine Methode `find_path(self, start, target)` der Klasse `Graph`, die einen Weg von einem `Node`-Objekt `start` zu einem `Node`-Objekt `target` findet. Der Weg muss nicht optimal sein. Testen Sie Ihre Methode im Modul `test04.py`, indem Sie einen Weg von der Kreuzung Winzererstr./Görresstr. zur Kreuzung Schellingstr./Cranachstr. suchen.

Tipp: Versuchen Sie, das Problem, von A nach B zu kommen, **rekursiv** durch einfachere Probleme auszudrücken. Ein zusätzliches Attribut `is_visited` in der Klasse `Node` (oder ein anderes geeignetes Attribut in der Klasse Graph) kann helfen.

Zum Testen benötigen Sie die `Node-Objekte` `start` und `target`, die Sie z. B. mit der Methode `Graph/find_node` aus dem Knotennamen ermitteln können.

Bemerkung:

Um in einem Graphen, dessen Kanten alle **keine negativen Gewichte besitzen**, einen kürzesten Pfad von einem Startknoten zu einem Endknoten zu finden, kann man den **Algorithmus von Dijkstra** verwenden. Dabei handelt es sich um einen Greedy-Algorithmus der die spezielle Datenstruktur *Priority Queue* verwendet. Er wird im Detail in meinem FWP2 Wahlfach *Algorithmendesign und höhere Datenstrukturen* besprochen.

4.4. *Freiwillig:* Das bisherige Klassendesign berücksichtigt nicht vollständig die Kapselung von Daten innerhalb einer Klasse (vgl. auch obige Bem.):

Auf die Attribute einer Klasse sollte gar nicht direkt von außen zugegriffen werden können (dafür gibt es Properties bzw. Getter und Setter), bei Methoden sollte zwischen privaten (internen) und öffentlichen unterschieden werden. Überprüfen Sie Ihr Design in Hinblick auf Kapselung und modifizieren Sie es entsprechend. Passen Sie auch die Klassendiagramme an.