Klaus Bovermann
24.05.23

Graphen im Unterricht

Mit Hilfe von Python und Jupyter-Notebooks

Am Bespiel des Dijkstra-Algorithmus

Einführung

An vielen Stellen des Alltags begegnen uns sogenannte *Graphen*. Beispielsweise ist das Netz des öffentlichen Nahverkehrs ein Graph; die einzelnen Haltestellen (*Knoten*) sind durch Linien (*Kanten*) miteinander verbunden. Oder man kann mit Hilfe von Graphen in einem sozialen Netzwerk die Beziehungen der Menschen modellieren, also z.B. darstellen, wer mit wem befreundet ist.

Schülerinnen und Schüler sollten im Informatikunterricht der Sekundarstufe 2 diese wichtige Datenstruktur kennenlernen und grundlegende Algorithmen verstehen, entwickeln und implementieren, wie z.B.:

- Tiefen- und Breitensuche in einem Graphen
- *Euler-Path* und *Hamilton-Path*
- Kürzeste Wege; der *Dijkstra-Algorithmus*
- Spannende Bäume; *Algorithmus von Kruskal* oder *Algorithmus von Prim*
- MaxFlowProblem; *Ford-Fulkerson-Algorithmus*
- TravellingSalesmanProblem; z.B. genetische oder andere approximative Algorithmen

Ein Graph besteht aus Knoten und Kanten, welche die Knoten miteinander verbinden. In diesem Artikel werden ausschließlich Graphen mit folgenden Eigenschaften betrachtet:

- Kanten sind *ungerichtet*[1]. Eine Kante zwischen zwei Knoten A und B kann also quasi in beiden “Richtungen” durchlaufen werden. Verbindet eine Kante v die Knoten A und B miteinander, nennt man A und B die Endknoten von v.
- Es gibt keine *Mehrfachkanten*. Es gibt also zwischen zwei Knoten A und B höchstens eine Kante.

[1]Das MaxFlowProblem beschäftigt sich mit *gerichteten* Graphen, so dass für diese Unterrichtsphase die hier vorgestellte Bibliothek geändert werden muss.

• Es gibt keine *Schleifen*, also Kanten, deren Endknoten identisch sind.

In einer Unterrichtseinheit wird man die Relevanz von Graphen zunächst an Beispielen darstellen. Anschließend werden unterschiedliche Typen von Graphen sowie deren Eigenschaften vorgestellt, ggf. je nach Zeit und Lerngruppe mehr oder weniger ausführlich und mathematisch formal. Möglicherweise kann man auch auf historische Entwicklungen wie das Königsberger Brückenproblem eingehen, z.B. auch in Form von Referaten.

In diesem Artikel wird exemplarisch aufgezeigt, wie man den Lernenden die Idee des Dijkstra-Algorithmus vermitteln kann. Voraussetzung dafür ist, dass die SuS die notwendigen Begriffe in Zusammenhang mit Graphen bereits erfahren haben.

Technische Details

Als Programmiersprache bietet sich z.B. *Python* an, da die Syntax besonders einfach ist, so dass ein Algorithmus, der bereits umgangssprachlich in einer Art Pseudocode entwickelt wurde, leicht programmiert werden kann. Dazu stellt Python eine Unmenge von Bibliotheken bereit, die es gestatten, sich bei der Implementation von Algorithmen auf die wesentlichen Aspekte konzentrieren zu können, ohne viele technische Details kennen zu müssen.

Es wurde eine weitere Bibliothek entwickelt, mit der man Sprechweisen von Graphen-Algorithmen direkt nutzen kann.

Als Entwicklungsumgebung ist der Anaconda-Navigator[2] mit den *Jupyter-Notebooks*[3] besonders geeignet:
• kostenlos
• leicht zu installieren
• Browser-basiert; in allen gängigen Browsern lauffähig
• für alle Plattformen verfügbar

Jupyter-Notebooks (JNBs) erlauben es, den Schülerinnen und Schülern Lernmaterial zur Verfügung zu stellen. Ein JNB enthält Lehrtexte und Python-Code. Ein JNB erfüllt dabei mehrere Zwecke.

---

[2] siehe z.B. https://www.anaconda.com/download

[3] siehe z.B. https://jupyter.org

1. Ein JNB ist ein vorbereitetes digitales *dynamisches Arbeitsblatt*
   - ein Tutorial mit Erklärtexten und -bildern, vorbereitetem Code, Aufgaben…
2. Ein JNB ist ein sog. *Computational Essay*
   - ein digitales Notizbuch, mit dem die lesende Person interagieren kann.
3. Ein JNB ist ein sog. *Worked Example*
   - enthält lauffähige und veränderbare Programme

Typischerweise enthält ein JNB Lehrtexte, die wie ein traditionelles Schulbuch die fachlichen Grundlagen vermitteln. Anders als ein Schulbuch kann ein JNB dynamische Bilder, anklickbare Links, Videos, … enthalten. Mit Hilfe von *MarkDown*[4] (eine Art HTML-Light) können diese Inhalte formatiert werden.

Daneben findet sich in einem JNB ausführbarer Python-Code. Schülerinnen und Schüler lernen so die Syntax und Semantik von Python, können den Code interaktiv verändern und erweitern, verbessern und vervollständigen.

Eine Python-Bibliothek für die Schule

Neben der Python-Bibliothek `networkx`, die im Internet erhältlich ist und in der Anaconda-Umgebung installiert werden kann, wurde eine eigene Python-Bibliothek mit dem Namen `nrw_graph` entworfen, mit der man Graphen-Algorithmen einfach und verständlich implementieren kann. Zentrale Idee dieser Bibliothek ist es, ungerichtete Graphen zu verwalten.

Die Bibliothek stellt diverse Funktionen bereit. Dabei werden sog. `assert`-Befehle benutzt, die sicherstellen, dass lesbare Fehlermeldungen generiert werden, wenn Funktionen mit nicht zulässigen Parametern aufgerufen werden. Das hat leider zur Folge, dass die Laufzeiten der Algorithmen sehr groß sind. Kann man auf diese Sicherheitsvorkehrungen verzichten, kann man in den JNBs stattdessen die Bibliothek `nrw_graph_unsafe` importieren.

Beide Bibliotheken können von GitHub heruntergeladen werden:

https://github.com/klausNetSchulbuch/GraphWithPython.git

[4] siehe z.B. https://markdown.de/

1. Knoten

Zunächst gibt es die Knoten des Graphen. Es eignen sich dafür alle Python-Objekte, die hashfähig sind. Typischerweise ist ein Knoten charakterisiert durch einen Namen, doch auch andere Objekte sind denkbar.

In dem Beispiel

```
import nrw_graph as ng
MeinGraph = ng.nrw_graph()
MeinGraph.fuegeKnotenHinzu("HalloPython")
MeinGraph.fuegeKnotenHinzu(True)
MeinGraph.fuegeKnotenHinzu(42)
```

wird der String `"HalloPython"`, der boolsche Wert `True` und die Zahl `42` dem zunächst leeren Graphen hinzugefügt. Dabei werden alle (neuen) Knoten mit einem Attribut `'besucht'` als unbesucht gekennzeichnet.

Jeder Knoten kann beliebig viele beliebige Attribute bekommen, wobei Attribute mit Hilfe von Namen spezifiziert werden:

```
MeinGraph.setKnotenAttribut(42, "schoenheit", "Toll")
MeinGraph.setKnotenAttribut("HalloPython", "wert", -5)
```

Man kann auch einem Knoten bereits beim Hinzufügen gewünschte Attribute mitgeben:

```
MeinGraph.fuegeKnotenHinzu(“Erna",gewicht=55,
                                          wohnort="Hamburg")
```

Der Aufruf

```
MeinGraph.alleKnoten()
```

erzeugt dann eine Liste aller Knoten:

```
['HalloPython', True, 42, 'Erna']
```

während der Aufruf

```
MeinGraph.getKnoten("Erna")
```

ein Python-Dictionary erzeugt:

```
{'gewicht': 55, 'wohnort': ‘Hamburg’, ‘besucht’: False}
```

Der Aufruf

```
MeinGraph.getKnotenAttribut(42, "schoenheit")
```

liefert dann den String

```
“Toll”
```

Weitere Vereinfachungen bieten Funktionen an, mit denen Knoten markiert und als besucht gekennzeichnet werden können:

- `markiereKnoten(knoten, marke)`
  - der angegebene Knoten erhält die angegebene Marke. Das ist ein Attribut vom Typ `'marke'`
- `getKnotenMarke(knoten)`
  - die dem angegebenen Knoten zugefügte Marke wird geliefert.
  - Hinweis: ein neu hinzugefügter Knoten besitzt noch keine Marke!
- `besucheKnoten(knoten)`
  - der angegebene Knoten wird als besucht markiert. Das Attribut `besucht` hat also den Wert `True`
- `verlasseKnoten(knoten)`
  - der angegebene Knoten wird als unbesucht markiert. Das Attribut `besucht` hat also den Wert `False`
- `knotenIstBesucht(knoten)`
  - liefert genau dann `True`, wenn der Knoten als besucht markiert wurde.
  - Hinweis: jeder einem Graphen neu hinzugefügte Knoten wird als unbesucht markiert.

Daneben gibt es noch folgende Funktionen, die die Benutzung vereinfachen:

- `alleNachbarknoten(knoten)`
  - liefert eine Liste aller Knoten, die über eine Kante mit dem angegebenen Knoten verbunden sind.
- `graphEinlesen(dateiname, sep = ",", header = 0)`
  - Hat man einen neuen (leeren) Graphen erzeugt, kann man hiermit alle Knoten und gewichteten Kanten aus einer CSV-Datei einlesen. Neben einer Kopfzeile enthält die Datei durch Kommata getrennte Angaben der Form

    `start, ziel, gewicht`

2. Kanten

Analog kann man ungerichtete Kanten verwalten. Das Python-Programm

```
MeinGraph.fuegeKanteHinzu("Erna", 42)
MeinGraph.fuegeKanteHinzu(42, True, laenge=123, farbe="pink")
MeinGraph.setKantenAttribut("Erna", 42, "gewicht", 5)

print(MeinGraph.alleKanten())
print(MeinGraph.getKante(42,True))
print(MeinGraph.getKante("Erna", 42))
print(MeinGraph.getKantenAttribut(True, 42, "farbe"))
```

erzeugt dann die Ausgaben:

```
[(True, 42), (42, 'Erna')]
{'laenge': 123, 'farbe': 'pink'}
{'gewicht': 5}
pink
```

Das folgende Python-Programm zeigt das Beispiel des bekannten *HausVomNikolaus*.

```
HDN = ng.nrw_graph()
HDN.fuegeKanteHinzu('A', 'B', nr = 1)
HDN.fuegeKanteHinzu('B', 'C', nr = 2)
HDN.fuegeKanteHinzu('C', 'E', nr = 3)
HDN.fuegeKanteHinzu('E', 'D', nr = 4)
HDN.fuegeKanteHinzu('D', 'C', nr = 5)
HDN.fuegeKanteHinzu('C', 'A', nr = 6)
HDN.fuegeKanteHinzu('A', 'D', nr = 7)
HDN.fuegeKanteHinzu('D', 'B', nr = 8)
```

Nutzt man zusätzlich weitere Bibliotheken[5]:

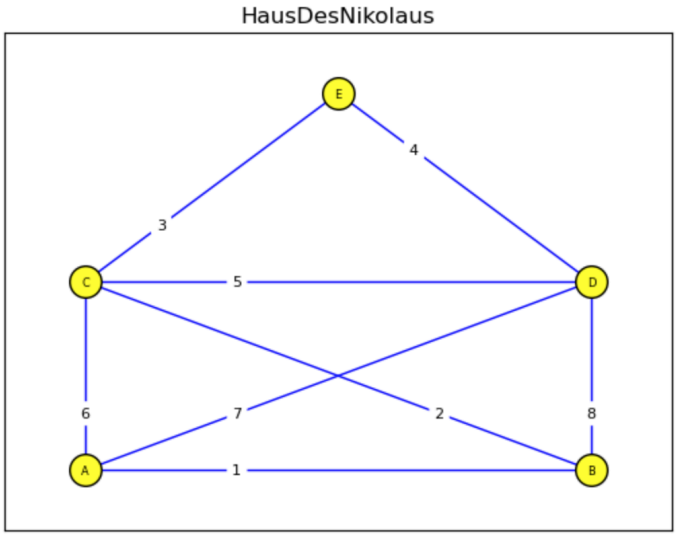
```
import pandas as pd
import matplotlib.pyplot as plt
import nrw_graph as ng
import networkx as nx
```

kann man den Graphen sogar sichtbar machen:

```
# explicitly set positions
pos = {'A' : (0, 0),
       'B' : (1, 0),
       'D' : (1, 1),
       'C' : (0, 1),
       'E' : (0.5, 2)}

kNames = {(s,z):HDN.getKantenAttribut(s,z,"nr") for (s,z) in HDN.alleKanten()}

node_options = {
    "node_color": "yellow",
    "edgecolors": "black",
    "node_size": 290,
}

edge_options = {"edge_color": "blue"}

node_label_options ="font_size": 6}

edge_label_options = {
    "font_size": 8,
    "font_color" : "black",
    "label_pos" : 0.7,
    "edge_labels" : kNames,
    "rotate" : False,
}

nx.draw_networkx_nodes(HDN, pos, **node_options)
nx.draw_networkx_edges(HDN, pos, **edge_options)
nx.draw_networkx_labels(HDN, pos, **node_label_options)
nx.draw_networkx_edge_labels(HDN, pos, **edge_label_options)

ax = plt.gca()
ax.set_title("HausDesNikolaus")

ax.margins(0.1)
plt.axis("on")
plt.show()
```

[5] Diese Bibliotheken müssen in der Anaconda-Umgebung ggf. nachträglich installiert werden. Die Bibliothek networkx muss dabei in der Betriebssystem-Shell nachinstalliert werden, da sie dem Anaconda-System unbekannt ist. Siehe dazu z.B. https://networkx.org/documentation/stable/install.html

so dass dann dieses Bild ausgegeben wird:



Die Jupyter-Notebooks

Für diese Unterrichtseinheit wurden (neben der Bibliothek `nrw_graph`) drei Jupyter-Notebooks entwickelt:

- `Dijkstra-Lernen.ipynb`
  - Hier wird an einem kleinen überschaubaren Graphen mit der Idee von Dijkstra schrittweise ein kürzester Weg erzeugt.
- `Dijkstra-Loesen.ipynb`
  - In diesem JNB wird der Dijkstra-Algorithmus automatisch ausgeführt. Der Graph sowie der Start- bzw. Zielknoten können in einer JNB-Zelle angepasst werden.
- `DijMitGui.ipynb`
  - Eine graphische Benutzungs-Schnittstelle erlaubt es, den Graphen sowie Start- und Zielknoten zu wählen. Der kürzeste Weg wird angezeigt.

Die Graphen

Für diese Unterrichtseinheit liegen die Daten von drei gewichteten Graphen in CSV-Dateien vor:

- `staedte.txt`
  - 19 deutsche Städte und einige Verbindungen.
- `stdt.txt`
  - Wie oben, jedoch mit Abkürzungen, um Tipparbeit zu sparen.
- `Highway_Germany.txt`
  - Alle deutschen Autobahnabschnitte (die Daten stammen aus dem Jahr 2000 und wurden aus einer amtlichen Datenbank entnommen, deren Quelle leider nicht mehr zu finden ist).

Jede Zeile der Datei stellt die Information für eine Kante dar in der Form

```
Start, Ziel, Länge
```

`Start` und `Ziel` werden dann als Knoten in Form eines Python-Strings genutzt, die Längenangabe ist ein Float.