Topologische Sortierung

Hausarbeit

im Modul

Algorithmen und Datenstrukturen

in der Studienrichtung Informatik

von

Marc Wegeleben

Leipzig, den 21.03.2023

Gutachter: Prof. Dr. Holger Perlt

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder veröffentlicht, noch einer anderen Prüfungsbehörde vorgelegt.

Leipzig, den 21.03.2023

. . . . . .

(Unterschrift des Kandidaten)

Inhaltsverzeichnis

1 Einleitung

Topologische Sortierung bezeichnet in der Mathematik eine Reihenfolge von Elementen. Nicht wie bei anderen Sortierverfahren, wird hier nicht mehr nach der Eigenschaft der zu sortierenden Elemente unterschieden, sondern nur noch nach paarweisen Beziehungen. Betrachtet man beispielhaft die Elemente als Aufgaben im eigenen Alltag oder in der Wirtschaft, so müssen manche Aufgaben unbedingt vor anderen ablaufen, manche sind flexibler. Hieraus lässt sich ableiten, dass hier die Rede von Halbordnungen ist und nicht Totalordnungen. Denn eine topologische Reihenfolge muss nicht eindeutig sein. In manchen gegebenen Fällen sind nämlich mehrere Anordnungen der Elemente möglich. In mengentheoretischer Sichtweise handelt es sich bei der topologischen Sortierung um eine lineare Erweiterung einer partiellen Ordnung. 1930 konnte Edward Szpilrajn zeigen, dass aus dem Auswahlaxiom folgt, dass jede partielle Ordnung zu einer linearen Ordnung erweiterbar ist. Das möchte aber nicht weiter diskutiert werden, denn das Auswahlaxiom wird eher bei unendlichen Mengen relevant, bei endlichen Mengen ist auch ohne dieses auf Weiteres zu schließen. Dieses Konzept konnte Daniel J. Lasser im Jahr 1961 folgend entwickeln und machte einen Algorithmus möglich, um für endliche Mengen eine topologische Sortierung zu erstellen. Im Folgenden möchte man sich dem Problem und der Lösung widmen. [1, 2, 3]

2 Das Problem

Wie eben erwähnt ist eine topologische Sortierung nicht so gewöhnlich wie andere Sortierverfahren. Ist beispielhaft eine Menge gegeben, die die größten Gebäude der Welt enthält, so ist es möglich diese mit einem Sortieralgorithmus nach der Größe (aufsteigend oder absteigend) zu sortieren. Was ist aber der Fall, wenn man nur Beziehungen zwischen diesen Elementen vorliegen hat? Sortieralgorithmen, die nach der eben genannten Größe sortieren, scheitern daran. Durch das topologische Sortieren wird es aber möglich gemacht, dass man genau diese Beziehungen betrachtet. Ein in der folgenden Abbildung gezeigter Graph zeigt eine mögliche beispielhafte Beziehung.

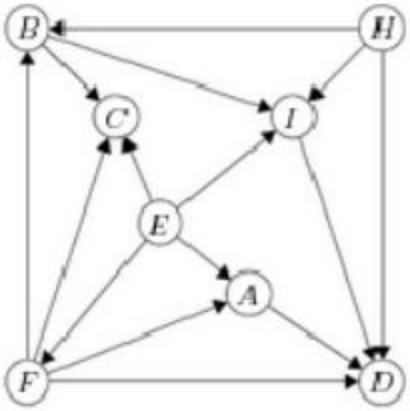

Abbildung 1: Beispielhafter Graph [4]

Dieser Graph stellt Beziehungen zwischen den Elementen, von *A-I* (ohne *G*), dar. Die Objekte, die man bisher als Elemente bezeichnet hat, können in der Graphentheorie als Knoten bezeichnet werden. Konkrete Beziehungen zwischen diesen Knoten werden über Kanten abgebildet. Die Kanten sind gerichtet, da diese nur in eine Richtung verlaufen. Solche Graphen nennt man gerichtete Graphen, welche voraussetzend für topologische Betrachtungen werden.

2.1 Beispiel: Kleidungsstücke

Um eine solche in der Abbildung gezeigte Beziehung realistischer oder mehr vorstellbar zu machen, ersetze man die Buchstaben als Kennzeichnung der Knoten zu Kleidungsstücken. Bei einer gewöhnlichen Morgenroutine eines Menschen, hat wohl jeder eine gewisse Reihenfolge wie er seine Kleidungsstücke anzieht. Im Folgenden wird davon ausgegangen, dass man dies tut und dies der Fall ist. Durch diese Reihenfolge und Abhängigkeit der Elemente entstehen Beziehungen zwischen den Knoten. Die folgende Abbildung zeigt einen beispielhaften Graph.

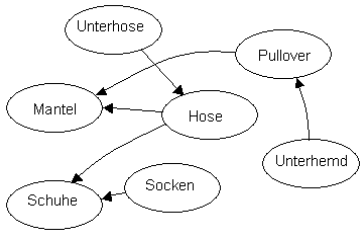

Abbildung 2: Beispielhafter Graph (Kleidungsstücke) [5]

Somit müssen Unterhosen vor Hosen angezogen werden, das Unterhemd vor dem Pullover und kein Pullover sollte nach einem Mantel angezogen werden. Drei Mögliche Abläufe könnten so aussehen:

Socken, Unterhemd, Pullover, Unterhose, Hose, Mantel, Schuhe
Unterhemd, Pullover, Socken, Unterhose, Hose, Mantel, Schuhe
Unterhose, Hose, Socken, Schuhe, Unterhemd, Pullover, Mantel

Die einzigen Möglichkeiten für diesen Graph und für den ersten Knoten belaufen sich auf entweder die Socken, das Unterhemd oder die Unterhose. Wichtig bleibt nur, dass jeder Knoten, der einen Vorgänger hat, diesen auch betrachtet und nicht vernachlässigt. Unter diesen Fällen gibt es noch weitere 78 Fälle, für welche sich korrekte topologische Reihenfolgen erzeugen lassen. Insgesamt sind das 81 Fälle, die topologisch sortiert sind.

Nun wurde beispielhaft verständlich gemacht wie topologisch sortiert werden kann. Im Folgenden möchte man auf die mathematischen Grundlagen eingehen.

3 Mathematische Beschreibung des Problems

Im Folgenden werden die Elemente der Menge M und bezüglich ihrer Relation R dargestellt. Für M und R ergeben sich die folgenden Eigenschaften

3.1 Irreflexiv

Für irreflexiv gilt:

$$\neg(x\ R\ x) \tag{3.1}$$

Ein Element bzw. Knoten sollte nicht in Relation zu sich selbst stehen.Vereinfacht gesagt sollte man nicht die Schuhe vor den Schuhen anziehen können oder Socken vor den Socken.

3.2 Transitiv

Für transitiv gilt:

$$\text{Wenn } x\ R\ y \text{ und } y\ R\ z\text{, so muss auch } x\ R\ z \text{ gelten} \tag{3.2}$$

Transitiv bedeutet, dass man nie für einen betrachteten Knoten Abhängigkeiten vernachlässige. Abhängigkeiten im Sinne von Vorgängern bzw. Kleidungsstücken. Ziehe man das Unterhemd an, dann den Pullover und weiterhin gilt, dass man nach dem Pullover den Mantel anziehe, so gilt, dass man vor dem Mantel das Unterhemd angezogen haben muss. [1]

3.3 Azyklisch

Azyklisch bedeutet, dass der Graph nicht zyklisch sein darf. Heißt einfacher gesagt, dass es keine Kreise im gegebenen Graphen geben darf. Würde es Kreise geben, so ist keine topologische Betrachtung anwendbar. Heißt ein Graph somit azyklisch, so gilt, dass er irreflexiv ist.

3.4 Ordnungsrelationen

Auch wenn für den Graph in der Gesamtheit keine Totalordnungen gelten, sondern Halbordnungen, ist dies nicht für die Betrachtung einer einzelnen Möglichkeit der Anordnung der Fall. Allerdings sei anzumerken, dass eine strenge Totalordnung keine Totalordnung ist, da eine strenge Totalordnung nicht reflexiv ist. Die strenge Totalordnung verletzt somit die Eigenschaft nicht, dass der Graph transitiv ist.

Beispielsweise ist gegeben:

Socken, Unterhemd, Pullover, Unterhose, Hose, Mantel, Schuhe

Für diese mögliche topologische Sortierung in sich selbst, gilt somit eine strenge bzw. starke Totalordnung.

Für eine strenge Totalordnung gilt ebenso, dass der Graph transitiv ist. Dafür wurde die mathematische Grundlage eben gelegt. Allerdings kommt noch eine weitere Eigenschaft herbei, nämlich die Trichotomie.

Trichotomie beschreibt, dass für eine

$$\textit{Relation} \prec \textit{auf } M \textit{ gilt: entweder } x \prec y \textit{ oder } x = y \textit{ oder } y \prec x$$

somit äquivalent mit (3.3)

$$\forall x,y \in M,\ (x \prec y) \vee (x = y) \vee (y \prec x)$$

Einfacher gesagt heißt das, dass die Reihenfolge der Anordnung der Elemente eindeutig ist. Es steht fest, ob man als erstes Kleidungsstück entweder die Socken, Unterhose oder Unterhemd angezogen habe. Die Menge M bildet bezüglich der Relation $\prec$ bzw. R eine strenge Halbordnung. So wie sich die strenge Totalordnung von der Totalordnung darin unterscheidet, dass Reflexivität und Antisymmetrie irreflexiv werden, so ist auch Folgendes bei einer strengen Halbordnung der Fall. Denn eine strenge Halbordnung ist transitiv und irreflexiv. Generell wird keine Relation dann über die Menge betrachtet, sondern nur ausreichende Teilmengen davon. Die Relation R wäre dann über einen transitiven Abschluss gegeben. Auf den transitiven Abschluss wird im Weiteren nicht mehr tiefer eingegangen.

Daraus ist abzuleiten, dass zu einer strengen Halbordnung R eine Totalordnung $\tilde{R}$ zu finden ist, für welche gilt:

$$\forall x,y \in M \ \textit{für}\ x\ R\ y \wedge x\ \tilde{R}\ y \tag{3.4}$$

Mit vorherigen Erläuterungen sind nun folgende Aussagen zu treffen:

Sei M eine Menge und

$$R \subseteq M\ x\ M$$

$$\text{so heißt} \tag{3.5}$$

$$T \subseteq M\ x\ M$$

topologische Sortierung für von M für R, wenn T eine strenge Totalordnung auf M ist, und Folgendes ebenso gilt:

$$R \subseteq T \tag{3.6}$$

Allerdings sei zu erwähnen, dass theoretisch die Bedingungen ebenso für Fälle im Unendlichen möglich sind. In Bezug auf die Praxis und Algorithmen in Anwendungen, ist es besser sich auf einen im Endlichen definierten Raum zu beziehen.

Wenn die Eigenschaft azyklisch gegeben sei, ist des Weiteren zu schließen, dass für

$$T \text{ von } M \text{ für } R, (M, R) \text{ ein gerichteter Graph ist} \tag{3.7}$$

3.4.1 Ordnungsrelationen am beispielhaften Graph

Um die Ordnungsrelationen („≺“ steht für „vor“) für den in Abbildung 1 erwähnten Graphen aufzustellen, so folgt:

$A \prec D, B \prec C, B \prec I, E \prec C, E \prec I, E \prec A, E \prec F,$

$F \prec C, F \prec D, F \prec A, F \prec B, H \prec B, H \prec D, H \prec I, I \prec D$

Daraus kann man folgende mögliche lineare Anordnung ableiten:

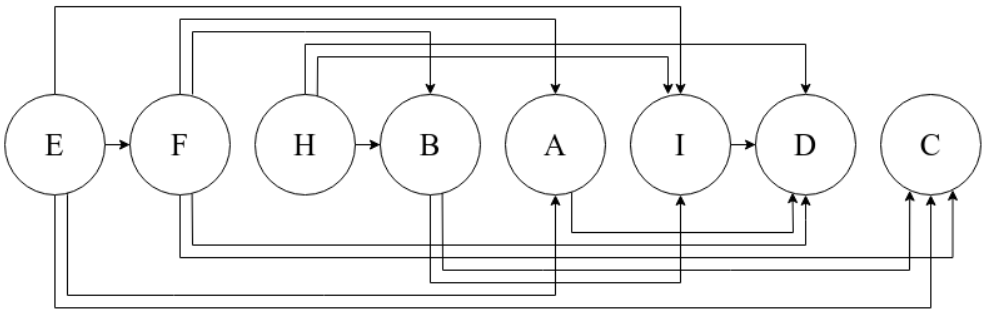

Abbildung 3: Lineare Anordnung des resultierenden Graphen [6]

4 Zeitkomplexität

Generell hängt die Zeitkomplexität von der Größe der Eingabe ab. Die Eingabe bezieht sich auf den gegebenen Graphen bzw. auf die Kanten und Knoten. Für einen Graphen mit *n* Knoten und *m* Kanten beträgt die Zeitkomplexität *O(n+m).* Das liegt daran, dass jeder Knoten und jede Kante einmal besucht werden müssen, um den Algorithmus auszuführen.

Diesen Fall kann man auch wie folgt beschreiben: *Average-Case = O(n+m)*

Allerdings gibt es auch einen weiteren Fall. Nämlich wenn ein vollständig verbundener Graph vorliegt. Dabei ist jeder Knoten mit jedem anderen Knoten verbunden. Für einen Graphen mit *n* Knoten und *m* Kanten beträgt die Zeitkomplexität *O(n^2)*

Diesen Fall kann man auch wie folgt beschreiben: *Worst-Case = O(n^2)* [7]

5 Implementierung in Python

In diesem Kapitel der Abhandlung soll es über die Implementierung des topologischen Sortierens in Python gehen. Jedoch sei gesagt, dass eine Umsetzung auch in anderen Programmiersprachen möglich ist oder generell in einer anderen Herangehensweise. Dennoch ist zu sagen, dass die mitgegebene Lösung in Python alle topologischen Sortierungen eines gegebenen Graphen

anzeigen kann und es möglich ist, jegliche Bezeichnung (als String) des Knotens zu wählen. Als voraussetzend soll sein, dass Python auf dem ausführenden System zur Verfügung steht, das Programm demnach interpretiert werden kann. Entweder kann das Programm direkt über eine IDE ausgeführt werden oder über eine Kommandozeile gestartet werden. In der Kommandozeile ist in den Pfad zu navigieren, worin die Datei *main.py* liegt.
Darin ist mit den folgenden Befehlen das Programm startbar:
python3 main.py oder *python main.py*

5.1 Beschreibung des Programms

Bei der Beschreibung der grundlegenden Funktionsweise, möchte man auch an dem Punkt starten, wo die Knoten bzw. Kanten definiert werden. Möchte man also einen Graphen abbilden, so müssen alle Kanten in dem dazugehörigen Tupel abgebildet werden. Alle diese Tupel sind unveränderlich und in einer Liste *edges* enthalten. Man möchte nun *edges* wie folgt definieren (Graph aus Aufgabenstellung bzw. Abbildung 1):

```
edges = [
        ("A", "D"), ("B", "C"),
        ("B", "I"), ("E", "C"),
        ("E", "I"), ("E", "A"),
        ("E", "F"), ("F", "C"),
        ("F", "D"), ("F", "A"),
        ("F", "B"), ("H", "B"),
        ("H", "D"), ("H", "I"), ("I", "D")
        ]
```
(5.1)

Hinzuzufügen ist, dass auch Eingaben dabei möglich sind, dass die Kanten aus fortlaufenden natürlichen Zahlen ab 0 aufgebaut sind.

```
edges = [(0, 1), (1,2)]
```
(5.2)

Es ist darauf zu achten, dass die Formen der Eingabemöglichkeiten sich nicht mischen. Es empfiehlt sich die erste genannte Möglichkeit der Eingabe zu verwenden, denn Folgendes ist

äquivalent für weitere Berechnungen (im Weiteren wird mit *node_dict* die Funktionsweise geklärt):

$$[(0, 1), (1,2)] \Leftrightarrow [("0", "1"), ("1","2")] \tag{5.3}$$

Danach ist ein neuer Graph zu instanziieren.

$$graph = Graph(edges) \tag{5.5}$$

Aus diesen gegebenen Kanten lässt sich folglich die dazugehörige Anzahl *N* von Knoten bestimmen, die für weitere Berechnungen benötigt wird.

$$self.N = len(set(sum(edges, ()))) \tag{5.4}$$

Dabei wird eine Adjazenzliste *adjList* als Datenstruktur verwendet, um den Graphen darzustellen. Eine Adjazenzliste repräsentiert einen gerichteten Graphen als Liste von Knoten und den jeweils damit verbundenen ausgehenden Kanten. Jeder Knoten in der Liste enthält eine Liste von Nachbarn, die durch ausgehende Kanten verbunden sind. In diesem Programm wird die Adjazenzliste durch eine Liste von Mengen dargestellt, wobei jedes Element der Liste einem Knoten im Graphen entspricht und die Menge die ausgehenden Kanten des Knotens darstellt.

```
adjList = [
        {3}, {2, 7}, set(), set(),
        {0, 2, 5, 7}, {0, 1, 2, 3},
        {1, 3, 7}, {3},
        ]
```
(5.6)

Wenn die Eingabe vollständig aus Strings besteht, wird das erkannt und das *dictionary node_dict* erzeugt.

$$node\_dict = \{'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'H': 6, 'I': 7\} \tag{5.7}$$

node_dict erzeugt die Zuordnung von Knoten-Namen (z. B. "A", "B", "C") zu Knoten-IDs (d.h. Ganzzahlen). Gleiches gilt auch für Strings, die Ganzzahlen enthalten (z. B. "0", "1", "2") und

natürlich für alle weiteren Möglichkeiten. Dies wird so verwendet, um die Knoten des Graphen in eine eindeutige Ganzzahl-ID zu übersetzen, da das Programm zur Verarbeitung von Ganzzahlen und nicht von Strings optimiert ist. Um dies zu erreichen, durchläuft das Programm alle Kanten im Graphen und erstellt eine Menge aller Knoten. Diese Menge wird sortiert, wobei jedes Element der sortierten Menge eine eindeutige Ganzzahl-ID enthält. Diese IDs und ihre entsprechenden Knotennamen werden dann in *node_dict* eingefügt. *node_dict* wird dann verwendet, um die Knoten-Namen in die entsprechenden Ganzzahl-IDs zu übersetzen, wenn es erforderlich ist, in einer numerischen Umgebung weiterzuarbeiten. Erforderlich wird das nämlich auch (Ganzzahl-IDs zu Knotennamen), wenn alle möglichen topologischen Sortierungen berechnet wurden und man der Ansicht und Ausgabe entsprechend wieder die eigentlichen Knotennamen zurückbekommen möchte.

Aus *node_dict* wird danach ein *result* erzeugt, welches eine brauchbare Struktur erzeugt. Anschließend wird *edges* gleich *result* gesetzt.

```
result = [
        (0, 3), (1, 2),
        (1, 7), (4, 2),
        (4, 7), (4, 0),
        (4, 5), (5, 2),
        (5, 3), (5, 0),
        (5, 1), (6, 1),
        (6, 3), (6, 7), (7,3)
        ]
```
(5.8)

Es ist die Aussage zu treffen, dass somit dynamisch und sortiert eine brauchbare Datenstruktur erzeugt wird. Optional wäre es durchaus möglich mit der Funktion *ord()* (standardgemäß integriert in Python) eine vergleichbare Struktur zu erzeugen. Allerdings würden die zurückgegeben Ganzzahlen den definierten *Unicode character* darstellen. Damit ist nicht gesichert, dass die Datenstruktur somit ab 0 beginnen würde.

Beim Instanziieren wird ebenso noch überprüft, ob der Graph zyklisch ist oder nicht. Ist dieser zyklisch, gibt die Methode *is_cyclic() True* zurück, wenn nicht, dann *False*. Ist es der Fall, dass

der Graph zyklisch ist, so kann keine toplogische Sortierung stattfinden. Für den Fall *False*, dass der Graph azyklisch ist, wird die Funktion *print_all_topological_orders()* aufgerufen. Diese Funktion ruft die Funktion *find_all_topological_orders()* auf, welche alle Möglichkeiten berechnet. Diese Funktion verwendet Rekursion, um alle möglichen Anordnungen zu durchlaufen. Wenn ein Knoten gefunden wird, dessen Eingangsgrad 0 ist und der nicht im aktuellen Pfad enthalten ist, wird der Eingangsgrad der Nachbarn dieses Knotens verringert und der Knoten zum Pfad hinzugefügt. Dann wird die Rekursion fortgesetzt, bis alle Knoten besucht wurden. Wenn eine topologische Sortierung gefunden wird, wird sie als Zeichenfolge in der Ergebnisliste gespeichert. Diese Ergebnisliste wird noch passend für eine anschaulich Ausgabe formatiert und dann in *print_all_topological_orders()* ausgegeben.

Die folgende Ausgabe enthält 36 Möglichkeiten für topologische Anordnungen des Graphen.

Die ersten Drei:

E, F, A, H, B, C, I, D

E, F, A, H, B, I, C, D

E, F, A, H, B, I, D, C

5.2 Deque als Datenstruktur?

In diesem Abschnitt möchte man nochmal genauer darauf eingehen, warum genau eine Deque (*double-ended queue*, sprich: „Deck“) als Datenstruktur verwendet wurde.

Im Allgemeinen wird die Verwendung damit begründet, um eine effiziente Implementierung von Warteschlangen und der Laufzeitkomplexität zu ermöglichen.

In der Funktion *is_cyclic()* wird *deque* verwendet, um eine Warteschlange von Knoten zu erstellen, die als nächstes besucht werden sollen. Wenn ein Knoten besucht wird, werden alle seine Nachbarn in die Warteschlange eingefügt. Wenn ein Knoten aus der Warteschlange entfernt wird, wird er als besucht markiert.

deque ermöglicht es, Elemente am Anfang oder Ende der Schlange hinzuzufügen oder zu entfernen, was in diesem Fall nützlich ist, um eine effiziente Breitensuche zu implementieren. Das geschieht mit *O(1)* und nicht mit *O(n)*, was bei Listen der Fall wäre.

In der Funktion *find_all_topological_orders()* wird *deque* verwendet, um eine aktuelle Pfadliste von Knoten im Graphen zu verwalten. Wenn ein neuer Knoten in den Pfad aufgenommen wird, wird er am Ende der *deque*-Instanz hinzugefügt. [8]

6 Zusammenfassung

In der Arbeit wurde das topologische Sortieren behandelt, dabei wurde zuerst eine Einführung gegeben, um dieses Konzept grundlegend zu beschreiben. Mit dem realitätsnahen Beispiel Kleidungsstücke konnte gezeigt werden, was es mit der topologischen Sortierung auf sich hat. Nachdem man das verständlich gemacht hat, wurde auf die mathematisch dahinterliegende Grundlage eingegangen. Die Erkenntnis konnte gemacht werden, dass einige Eigenschaften bzw. Bedingungen für den gegebenen Graphen gelten müssen, um die topologische Sortierung überhaupt anzuwenden.

Bei der Zeitkomplexität konnte man feststellen, dass es nicht nur einen einzigen Fall gibt, welcher die Zeitkomplexität beschreibt. Abschließend wurde eine Implementierung in Python vorgestellt. Die vorliegende Beschreibung des Programms bzw. der Implementierung soll nur ein grundlegendes Verständnis über das Programm geben. Weiteres kann durch eigenständiges Nachforschen oder durch die Betrachtung des Quelltextes, mit den darin enthaltenden Kommentaren, erarbeitet und verstanden werden.

Literaturverzeichnis

[1]: Niklaus Wirth: Algorithmen und Datenstrukturen, Pascal Version. 4. Auflage. Teubner Verlag, Stuttgart 1995
[2]: Edward Szpilrajn: Sur l'extension de l'ordre partiel. In: Fundamenta Mathematicae. Band 16, 1930
[3]: Daniel J. Lasser: Topological Ordering of a List of Randomly-Numbered Elements of a Network. In: Communications of the ACM. Band 4, 1961
[4]: Prof. Dr. Holger Perlt, Algorithmen und Datenstrukturen (Aufgabenstellung für die Hausarbeit von Marc Wegeleben)
[5]: Wikipedia, unter https://upload.wikimedia.org/wikipedia/de/5/51/Kleidergraph.png, (abgerufen am 06.03.2023)
[6]: Diagrams.net: Selbsterstellter Graph, unter https://app.diagrams.net/ (abgerufen am 14.03.2023)
[7]: ETH Zürich (DA Skript): 24. Gaphen (2021), unter https://lec.inf.ethz.ch/DA/2021/slides/daLecture17.handout.pdf (abgerufen am 14.03.2023)
[8]: Vitaly Shchurov, Python: deque vs. list (06.12.2020), unter https://dev.to/v_it_aly/python-deque-vs-listwh-25i9 (abgerufen am 06.03.2023)

Abbildungsverzeichnis