Fachdidaktik Informatik

Semesterbegleitende Übung

Breitensuche

Ausarbeitung Leitprogrammartiger Unterrichtsunterlagen für den Informatikunterricht am Gymnasium

Sabina Schellenberg & Johannes Popp

sabinasc@student.ethz.ch & jpopp@ethz.ch

08-923-377 & 09-931-999

Lehrdiplom Informatik

Version: 0.1

HS 2017

Inhaltsverzeichnis

1 Konzeption **1**
1.1 Concept Map . . . 1
1.2 Aufbau der LPU . . . 1
1.2.1 Für wen sind die LPU geeignet? . . . 1
1.2.2 Vorwissen . . . 1
1.2.3 Konzeption . . . 3
1.3 Lernziele . . . 3
1.3.1 Leitidee . . . 3
1.3.2 Dispositionsziel . . . 3
1.3.3 Operationalisierte Lernziele . . . 3
1.4 Sequenzierung der Unterrichtseinheit . . . 4

2 Unterlagen **5**
2.1 Graphen . . . 6
2.1.1 Begriffe . . . 6
2.1.2 Definitionen . . . 6
2.1.3 Zusammenfassung und Kontrollaufgaben . . . 12
2.2 Breitensuche . . . 13
2.2.1 Einführung . . . 13
2.2.2 Algorithmus . . . 15
2.2.3 Anwendung und Erweiterung . . . 19
2.2.4 Beispiele . . . 23
2.3 Lösungen . . . 25

Literatur **29**

Kapitel 1

Konzeption

1.1 Concept Map

Das Thema der vorliegenden Arbeit ist die Ausarbeitung des Algorithmus der Breitensuche in Form Leitprogrammartiger Unterrichtsunterlagen (LPU) für das Gymnasium. Zu diesem Zweck wurde folgende Concept-Map (s. Abb. 1.1) entworfen, die sowohl das Vorwissen als auch einen kleinen Ausblick auf weitere Themen geben soll. Dabei wurden solche Konzepte blau gefärbt, welche als Vorwissen schon bekannt sein sollten, aber im Rahmen dieser Arbeit nochmal wiederholt werden. Neue Konzepte wurden grün eingefärbt und werden in dieser Arbeit behandelt bzw. eingeführt. Weitere Konzepte, die nicht mehr in dieser Arbeit behandelt werden, wurden orange gefärbt. Insbesondere haben möchten wir die Darstellung von Graphen in Form von Adjazenmatrizen wiederholen und fundiert mit den SuS besprechen. Deshalb wurde das Konzept *grün* gefärbt. Wir haben uns bewusst für die Darstellung mit Adjazenzmatrizen entschieden, da wir dieses für den Algorithmus der Breitensuche benötigen. Alternativ kann man auch Adjazenzlisten (*orange*) verwenden. Beide Konzepte sind aber Listen von Listen und für SuS nicht trivial.
Aus dieser Concept Map ergibt sich eine Sequenzierung des Unterrichts, bei dem zuerst die grundlegenden Begriffe eines Graphen repetiert werden, damit das Vorwissen der Schüler aktiviert wird. Darauf aufbauend werden dann die neuen Konzepte eingeführt. Ein kurzer inhaltlicher Überblick dieser Sequenzierung wird in Kapitel 1.4 vorgestellt.

1.2 Aufbau der LPU

1.2.1 Für wen sind die LPU geeignet?

Die Unterlagen richten sich an Schülerinnen und Schüler der Gymnasialstufe (11. oder 12. Schuljahr) und sind für den Informatikunterricht im Ergänzungsfach Informatik geeignet, idealerweise für Schüler mit mathematisch/naturwissenschaftlichem Schwerpunkt. Die vorliegenden Unterlagen sollten im zweiten Quartal des Ergänzungsfaches Informatik eingesetzt werden können.

1.2.2 Vorwissen

Es wird davon ausgegangen, dass die Schüler bereits einige Grundlagen der Informatik kennengelernt haben, insbesondere den Begriff Algorithmus. Zum Vorwissen der Schüler

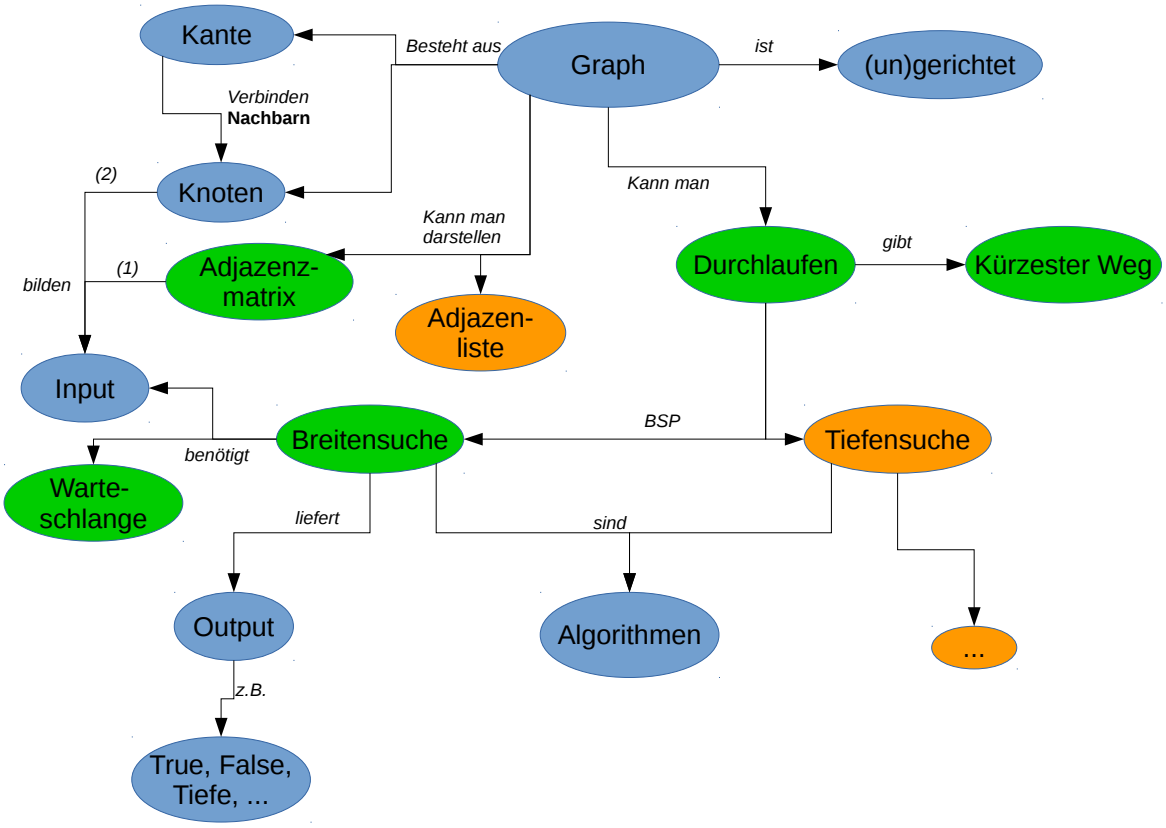


Abbildung 1.1: Concept-Map zum Thema Breitensuche. Blaue Konzepte und grüne Konzept werden in dieser Arbeit behandelt, wobei die Blauen das Vorwissen abbilden und Grüne neue Konzepte sind. Orangene Konzepte bilden einen Ausblick auf Konzepte, welche im Rahmen dieser Arbeit nicht mehr behandelt werden.

sollte ausserdem der Begriff des Graphen gehören.
Es wird davon ausgegangen, dass die Schüler den Begriff des Graphen bereits kennen. Dieses Vorwissen soll zu Beginn nochmals aktiviert werden. Dabei werden die wichtigsten Begriffe repetiert: der Aufbau eines Graphen aus Knoten und Kanten, gerichtete und ungerichtete Graphen sowie der Begriff von Nachbarsknoten. Damit wird eine klare Basis für die in den Unterlagen verwendeten Begriffe geschaffen.
Der Begriff des Algorithmus wird als Vorwissen vorausgesetzt, aber nicht nochmals aktiviert bzw. repetiert, da davon ausgegangen wird, dass sich die Schüler schon vertiefter mit dem Begriff auseinandergesetzt haben und im Rahmen des Ergänzungsfaches Informatik schon verschiedene Algorithmen kennengelernt bzw. entwickelt haben. Die SuS können Algorithmen in Form von Pseudocode lesen und selber verfassen.
Für die Implementationen und die Programmierübungen werden keine Details repetiert, da die Schüler auch hier im Rahmen des Ergänzungsfaches bereits einige Programme implementiert und die Grundlagen von Datentypen, Kontrollstrukturen und modularem Aufbaus kennengelernt und viel geübt haben sollten. Wichtig sind für die Programmieraufgaben in diesen Unterlagen unter anderem Unterprogramme, Arrays und Schleifen. Zusätzlich sollen die Schüler wissen, wie Programme modular aufgebaut werden können und bereits in der Lage sein, einfache Programme in TigerJython zu schreiben. Die Dar-

stellung der Graphen als Listen von Listen in Python wird explizit repetiert.
Neue Begriffe und Konzepte werden in den Unterlagen so eingeführt, dass sie am vorhandenen Wissen anknüpfen und dieses erweitern.

1.2.3 Konzeption

Die Unterlagen sind so aufgebaut, dass schrittweise neue Konzepte eingeführt und erläutert werden. Im Fokus soll die Entwicklung beziehungsweise Förderung des Algorithmischen Denkens stehen. Die Schüler sollen nicht nur das Konzept und die Anwendung der Breitensuche kennenlernen, sondern auch lernen, wie ein Algorithmus entwickelt wird und wie ein Algorithmus formuliert werden soll, damit er für eine breite Klasse von Problemen eingesetzt werden kann. Die LPU sollen dazu beitragen, dass die Schüler sich intensiv mit der Formulierung der Problemstellung sowie der Entwicklung des Algorithmus auseinandersetzen und schlussendlich auch in der Lage sind, den Algorithmus zu implementieren.

1.3 Lernziele

Aus der Concept-Map lassen sich folgende Lernziele für diese Arbeit ableiten.

1.3.1 Leitidee

Graphen spielen eine wichtige Rolle in unserem Alltag und werden sehr häufig zur Darstellung von Zusammenhängen verwendet (S-Bahnnetzwerk, Facebook, Websites, ...). Mit diesen Netzwerken sind viele Fragen verbunden: Über wie viel Freundschaften bin ich mit einer anderen Person verbunden? Wie lange ist die kürzeste Verbindung von A nach B? Damit man ein grundlegendes Verständnis dafür entwickelt, muss man sich überlegen, wie man Graphen durchsuchen kann. Dies kann man erreichen indem man in der Breite oder in der Tiefe sucht. Mithilfe der Breitensuche soll ein elementarer Einstieg aufgezeigt werden.
Graphen sind aber nicht nur „schöne“ Objekte, sonder mithilfe dieser können wir Probleme mathematisch modellieren. Viele Probleme lassen sich auf Graphen reduzieren (SAT (Hro14)) und sind ein grundlegendes Werkzeug um Probleme zu klassifizieren.

1.3.2 Dispositionsziel

Die SuS wissen, dass man gewisse Probleme mit Hilfe von Graphen modellieren und mit Graphenalgorithmen lösen kann. Sie können analysieren und beurteilen, ob Problemstellungen mithilfe von Breitensuche in einem Graphen gelöst werden können.

1.3.3 Operationalisierte Lernziele

Nach dieser Einheit können die SuS ...

1. ... die Begriffe und Unterschiede zur Darstellung von einfachen Graphen verstehen: (un)gerichtet, Knoten und Kante.
2. ... verschiedene Darstellungsmöglichkeiten von Graphen aufzählen und diese ineinander überführen: Zeichnung, Knoten-/ Kantenmenge und Adjazenzmatrix.

3. ... die Nachbarn von Knoten auf verschieden Darstellungen von Graphen bestimmen.
4. ... ein Programm schreiben, welches die Nachbarn eines bestimmten Knoten eines beliebigen Graphen ausgibt.
5. ... ein gegebenes Problem mit einem Graphen modellieren.
6. ... die Funktionsweise einer Breitensuche in einem Graphen beschreiben.
7. ... für einen Graphen mittels Breitensuche beurteilen, ob ein Knoten von einem anderen Knoten aus erreichbar ist.
8. ... für einen Graphen mittels Breitensuche den kürzesten Weg von einem Knoten zu einem anderen Knoten finden (in Bezug auf die Anzahl Kanten, die traversiert werden müssen).
9. ... für einen Graphen mittels Breitensuche eine Folge von Knoten angeben, welche auf kürzestem Weg von einem Knoten zu einem anderen Knoten führen.
10. ... ein gegebenes Problem mit einem Graphen modellieren und mittels Breitensuche eine passende Lösung des Problems finden.
11. ... die Breitensuche in Tiger Jython implementieren.
12. ... die Laufzeit einer Breitensuche beurteilen.
13. ... den Speicherbedarf der Breitensuche beurteilen.

1.4 Sequenzierung der Unterrichtseinheit

Im ersten Abschnitt möchten wir die SuS für das Thema motivieren und abholen. Dazu präsentieren wir das Problem des kürzesten Weges im ÖV. Anhand dessen zeigen wir dann auf, dass wir zuerst gewisse Konzepte wiederholen und erweitern müssen, um dieses Problem zu lösen.
Aus der Concept-Map und den Lernzielen ergibt sich folgender Ablauf für die LPU: In einem ersten Abschnitt werden nochmal die Grundlagen zu Graphen und ihrer Darstellungen wiederholt und anhand von Beispielen und Aufgaben geübt. Dabei wird darauf geachtet zuerst eine zeichnerische Darstellung, dann eine Mengendarstellung und zum Schluss die Adjazenzmatrix einzuführen. Hierbei wurde sich an (CLR04, OW12) orientiert. Zum Schluss des ersten Abschnitts soll mit der Einführung von Nachbarn und dem Schreiben eines solchen Programms eine Vorarbeit für den kommenden Abschnitt gelegt werden.
Im zweiten Abschnitt werden Grundlagen über das Modellieren von Problemen mit Hilfe von Graphen und das Lösen der Probleme mit Hilfe von Graphenalgorithmen vermittelt. Insbesondere werden Problemstellungen betrachtet, die sich mit Hilfe von der Breitensuche lösen lassen. Der Algorithmus für die Breitensuche wird Schritt für Schritt erarbeitet, erweitert und implementiert sowie an konkreten Beispielen angewendet. Für den Algorithmus wird ein Vorgehen mit Färben von Knoten verwendet, wie es auch in (CLR04) zu finden ist.

Kapitel 2

Unterlagen

Überblick

Haben Sie sich schon mal überlegt, wie Sie mithilfe des ÖV von einer Station zu ihrer Destination gelangen können. Zum Beispiel: Was ist die kürzeste Verbindung zwischen *Hauptwil* und *Horn* in St. Gallen (s. Abb. 2.1). Wahrscheinlich haben Sie sich auf der Website des Betreibers erkundigt und dort eine Antwort erhalten. Dies war eine automatisierte Bearbeitung Ihrer Anfrage und hinter Website steckt also ein Programm, dass mithilfe eines Algorithmus Ihre Anfrage bearbeitet und eine Ausgabe generiert.

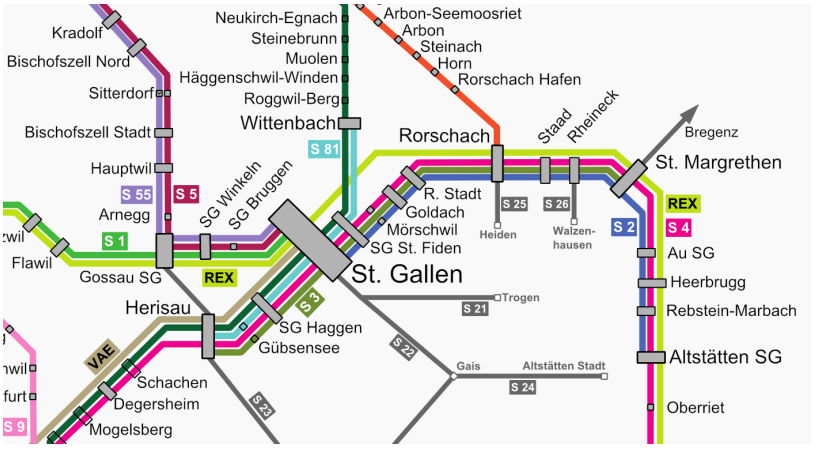


Abbildung 2.1: Darstellung eines Ausschnitts des Sankt Galler S-Bahn-Netzes als (ungerichteter) Graph. Jede Station bildet dabei einen Knoten und die Verbindung zwischen den Knoten ist eine Kante.

Damit man aber das komplexe Gebilde eines Fahrplans für einen Algorithmus (Computerprogramm) vereinfachen kann, muss man sich auf eine Darstellung einigen. Jede Station eines Fahrplannetzes ist ein Knotenpunkt, von dem verschiedene Verbindungen zu an deren Stationen ausgehen. Die mathematische Darstellung eines solchen Gebildes nennt man Graph.
Diesen Begriff haben Sie schon einmal kennengelernt und Sie wissen auch, dass man damit verschiedene Objekte mit einander verbinden kann. Eine interessante Frage ist immer, über wie viele Verbindungen zwei Objekte mit einander verknüpft sind. Wir möchten nun mit diesem Kapitel überlegen, wie man einen Graphen durchsuchen kann. Ein Beispiel

für einen solchen Algorithmus ist die Breitensuche. Ziel ist es, dass Sie den Algorithmus verstehen und anwenden können.

2.1 Graphen

2.1.1 Begriffe

Als erstes möchten wir uns mit Graphen auseinander setzen. Wie wir oben gesehen haben kann man zum Beispiel das Liniennetz eines ÖV-Betreibers als Graph darstellen.
Im Allgemeinen ist ein Graph eine Struktur, die eine Menge von Objekten (z.B. Stationen) zusammen mit den zwischen diesen Objekten bestehenden Verbindungen (z.B. S-Bahn, Bus) repräsentiert.
Die Objekte werden dabei **Knoten** des Graphen genannt. Die paarweisen Verbindungen zwischen Knoten heissen **Kanten**. Die Kanten können **gerichtet** oder **ungerichtet** sein, wenn die Verbindungen zwischen den Knoten eine Richtung beinhalten (oder nicht). Dies wäre in unseren Fall eine Verbindung zwischen zwei Stationen, die nur in eine Richtung befahrbar ist; eine Einbahnstrasse.
Am einfachsten kann man Graphen zeichen, indem man Knoten durch Punkte und die Kanten durch Linien (ungerichtet) oder durch Pfeile im gerichteten Fall darstellt.

Beispiel 2.1. Anschauliche Beispiele für Graphen sind ein Stammbaum oder das S-Bahn-Netz (s. Abb. 2.2) einer Stadt. Bei einem Stammbaum stellt jeder Knoten ein Familienmitglied dar und jede Kante ist eine Verbindung zwischen einem Elternteil und einem Kind. In einem S-Bahn-Netz stellt jeder Knoten eine S-Bahn-Station dar und jede Kante eine direkte Zugverbindung zwischen zwei Stationen.

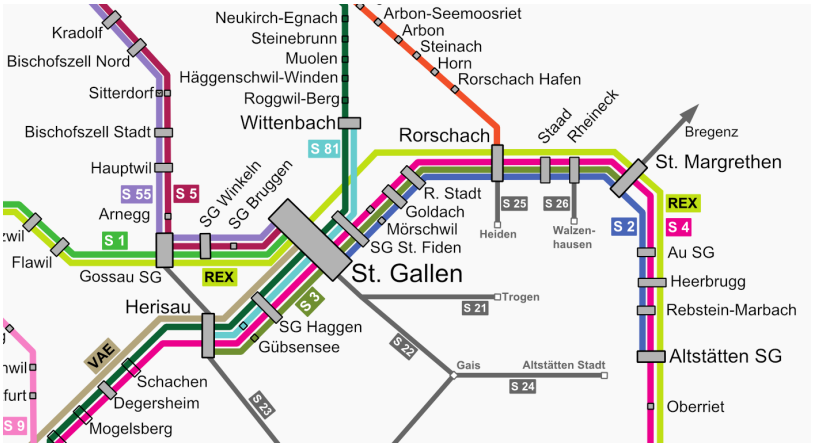


Abbildung 2.2: Darstellung eines Ausschnitts des Sankt Galler S-Bahn-Netzes als (ungerichteter) Graph. Jede Station bildet dabei einen Knoten und die Verbindung zwischen den Knoten ist eine Kante.

2.1.2 Definitionen

Wir führen direkt die Definition für Graphen ein wobei wir zwei Arten unterscheiden: gerichtete und ungerichtete Graphen.

Definition 2.1. Mathematisch besteht ein **Graph** $G = (V, E)$ aus einer **Menge** von **Knoten** V (engl. *vertex*) und einer **Menge** von **Kanten** E (engl. *edge*). Die Anzahl Knoten wird mit $|V|$ und die Anzahl Kanten mit $|E|$ bezeichnet.

Definition 2.2. Jede Kante eines **ungerichteten Graphen** e besteht aus zwei Knoten v_1 und v_2 und wird selbst als Menge dargestellt: $e = \{v_1, v_2\}$. Die Reihenfolge spielt dabei keine Rolle: $\{v_1, v_2\} = \{v_2, v_1\}$.

Beispiel 2.2. Die Abblidung 2.3 zeigt einen ungerichteten Graphen mit 5 Knoten und 7 ungerichteten Kanten.

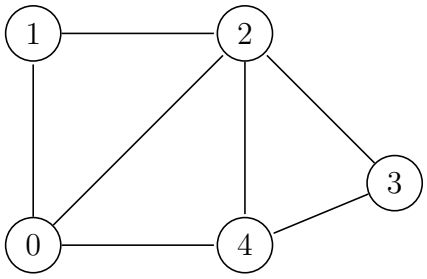


Abbildung 2.3: Ein gerichteter Graph mit 6 Knoten.

Die mathematische Darstellung des Graphen lautet:

$$G = (V, E); \quad V = \{0, 1, 2, 3, 4\};$$
$$E = \{\{1, 2\}, \{1, 0\}, \{2, 0\}, \{2, 4\}, \{2, 3\}, \{3, 4\}, \{4, 0\}.\}$$

Definition 2.3. Jede Kante eines **gerichteten Graphen** e besteht aus einem Startknoten v_1 und einem Zielknoten v_2 und wird selbst als 2-Tupel dargestellt: $e = (v_1, v_2)$. Hierbei spielt die Reihenfolge ein Rolle: $(v_1, v_2) \neq (v_2, v_1)$.

Beispiel 2.3. Die Abblidung 2.4 zeigt einen gerichteten Graphen mit 6 Knoten und 7 Kanten.

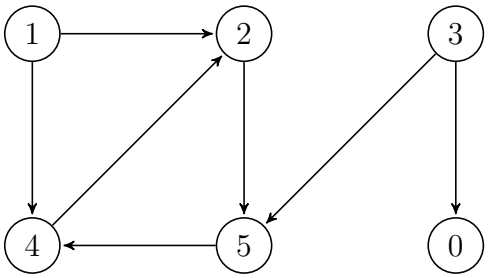


Abbildung 2.4: Ein gerichteter Graph mit 6 Knoten.

Die mathematische Darstellung des Graphen lautet:

$$G = (V, E); \quad V = \{0, 1, 2, 3, 4, 5\};$$
$$E = \{(1, 2), (1, 4), (2, 5), (3, 5), (3, 0), (4, 2), (5, 4)\}.$$

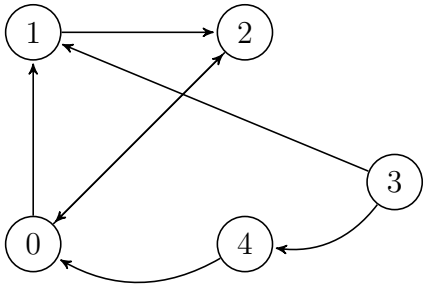


Abbildung 2.5: Ein gerichteter Graph.

Aufgabe 2.1. Bilden Sie für folgenden Graph (Abb. 2.5) die entsprechende mathematische Darstellung $G = (V, E)$.

Aufgabe 2.2. Zeichnen Sie den entsprechenden Graph, der zu folgender Mathematischen Darstellung gehört:

$$G = (V, E); \quad V = \{0, 1, 2, 3, 4, 5, 6\};$$
$$E = \{(1, 3), (1, 6), (2, 1), (3, 4), (4, 2), (5, 4), (6, 3)\}.$$

Definition 2.4. Für einen Graphen $G = (V, E)$ nehmen wir an, dass die Knoten in beliebiger Weise von 0 bis $|V| - 1$ nummeriert sind. So bildet die **Adjazenzmatrix-Darstellung** des Graphen G eine $|V| \times |V|$-Matrix $A = a_{ij}$ mit den Elementen

$$a_{ij} = \begin{cases} 1 & \text{falls } (i, j) \in E, \\ 0 & \text{sonst} \end{cases}.$$

Für ungerichtete Graphen ersetzt man (i, j) durch $\{i, j\}$.

Das Element a_{ij} hat zwei Indizes i und j. Der erste Index (i) gibt uns die Zeile der Matrix an und der zweite (j) die Spalte:

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} & \cdots \\ a_{10} & a_{11} & a_{12} & \cdots \\ a_{20} & a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

Man muss bei ungerichteten Graphen beachten, dass eine Verbindung zwischen zwei Knoten in beide Richtungen durchlaufen werden kann und beide Verbindungen in die Matrix eingetragen werden müssen.

Beispiel 2.4. Die Adjazenz-Matrizen A zu dem Graph in Abb. 2.3 lautet:

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Beispiel 2.5. Die Adjazenz-Matrizen A zu dem Graph in Abb. 2.4 lautet:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Aufgabe 2.3. Bilden Sie für den Graphen aus Abbildung 2.5 die entsprechende Adjazenzmatrix $G = (V, E)$.

Aufgabe 2.4. Zeichnen Sie zu folgender Adjazenzmatrix den entsprechenden Graphen. Überlegen Sie sich zu erst, ob es sich um einen gerichteten oder ungerichteten Graphen handelt.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Programmierung: Damit auch ein Computerprogramm einen Graphen lesen kann, kann man diesen als Liste von Listen darstellen. Der folgende Befehl erstellt eine Liste mit drei einträgen.

```
a = [1,5,3]
print a[1]
```

Mit dem zweiten Befehl wird das zweite Element ausgegeben - in diesem Fall 5. Es ist wichtig zu beachten, dass in Python das erste Element mit 0 markiert ist. Damit wir eine Matrix speichern speichern wir nun mehrere Listen in einer Liste

```
A = [ [0,3,5], [2,4,1], [-1,7,9] ]
print A[1]
print A[0][2]
```

Die Ausgabe ergibt:

```
[2 ,4 ,1]
5
```

Wir speichern also Zeile für Zeile eine Matrix in diesem Beispiel wurde folgende Matrix gespeichert:

$$A = \begin{pmatrix} 0 & 3 & 5 \\ 2 & 4 & 1 \\ -1 & 7 & 9 \end{pmatrix}$$

Aufgabe 2.5. Stellen Sie folgende Matrix in Python dar. Überprüfen Sie dabei ihre Eingabe indem Sie die zweite Zeile und das Element a_{23} ausgeben (print).

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Definition 2.5. Zwei Knoten sind zueinander **benachbart**, wenn eine direkte Verbindung durch eine Kante besteht. In einem ungerichteten Graphen sind immer beide Knoten benachbart. Hingegen spielt in einem gerichteten Graphen die Richtung der Kante eine Rolle, so dass die Nachbarschaft nur in die Richtung gelten kann, in welche der Pfeil zeigt.

Beispiel 2.6. In ungerichteten Graphen sind immer die zwei Knoten benachbart, wenn sie verbunden sind. So ist in der folgenden Abbildung 2.6 auf der rechten Seite der Knoten 2 und 3 benachbart. Bei gerichteten Graphen jedoch spielt die Richtung eine Rolle. So

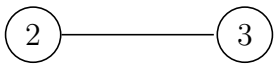


Abbildung 2.6: Zwei Knoten in einem ungerichteten Graphen.

hat im folgenden Beispiel (Abb. 2.7) der Knoten 0 den Nachbarknoten 1, aber der Knoten 1 hat keinen Nachbarknoten.

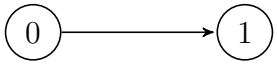


Abbildung 2.7: Zwei gerichteten Graphen.

Man kann die Nachbarn direkt aus der Adjazenzmatrix ablesen. Dazu betrachtet man die Zeile des Knoten und immer wenn eine Eins geschrieben steht, heisst dass, der Knoten in der entsprechenden Spalte ein Nachbar ist.

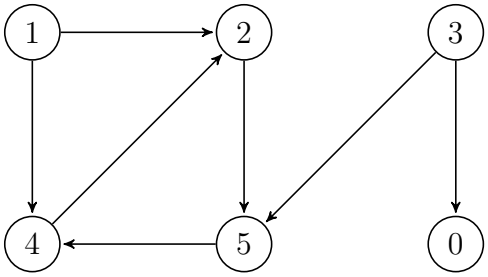


Abbildung 2.8: Ein gerichteter Graph mit 6 Knoten.

Beispiel 2.7. Betrachten wir nochmal den Graphen der Abbildung 2.4 mit seiner Adjazenzmatrix:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Hier hat der Knoten 1 die Nachbarn 2 und 4. Die entsprechende Zeile ist **fett** markiert in der Matrix. Umgekehrt hat der Knoten 2 nur den Knoten 5 als Nachbar und nicht den Knoten 1.

Aufgabe 2.6. Welche Nachbarn hat der Knoten 3, der Knoten 1 und der Knoten 0 des folgenden Graphen in Abbildung 2.9.

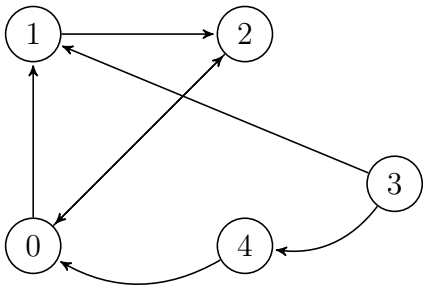


Abbildung 2.9: Ein gerichteter Graph.

Aufgabe 2.7. Bestimmen Sie die Nachbarknoten der Knoten 0,2 und 4 der folgenden Adjazenzmatrix. Gehen Sie dabei davon aus, dass wie immer die Knoten von Null an durchnummeriert sind.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Aufgabe 2.8. Implementieren Sie eine Funktion NACHBARKNOTEN(*graph*, *knoten*): Sie hat als Input einen Graphen (*graph*) und einen Knoten (*knoten*) und gibt als Output eine Liste von Nachbarknoten des Knoten aus. Testen Sie die Funktion, indem Sie sie mit verschiedenen Knoten aus einem Graphen aufrufen.

2.1.3 Zusammenfassung und Kontrollaufgaben

In diesem Kapitel haben Sie die Darstellung von Graphen wiederholt. Insbesondere haben Sie verschiedene Darstellungen der Graphen kennen gelernt: Zeichnung, Menge von Kanten und Knoten und Adjazenzmatrix. Zusätzlich kennen Sie den Unterschiede zwischen gerichteten und ungerichteten Graphen, welche sich auch in den verschiedenen Darstellungen zeigen. Sie können auch die Graphen in einem Programm mithilfe von Listen in Listen darstellen und ausgeben. Zusätzlich haben Sie das Konzept der benachbarten Knoten (Nachbar) in Graphen kennengelernt und können Nachbarknoten in verschiedenen Darstellungsformen bestimmen.

Kontrollaufgabe 1. Beschreiben Sie Unterschiede und Gemeinsamkeiten von gerichteten und ungerichteten Graphen.

Kontrollaufgabe 2. Nennen Sie drei weitere Beispiele aus dem Alltag für Graphen. Bestimmen Sie dabei immer was die Knoten und was die Kanten darstellen. Handelt es sich dabei um gerichtete oder ungerichtete Graphen?

Kontrollaufgabe 3. Betrachten Sie folgenden Graphen (s. Abb. 2.10) und bestimmen Sie seine Knoten- und Kantemenge und bestimmen Sie zusätzlich die Adjazenzmatrix.

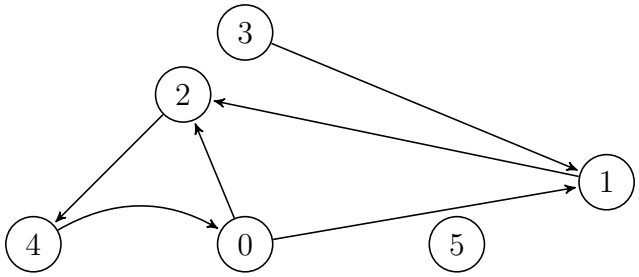


Abbildung 2.10: Ein weiterer Graph.

Kontrollaufgabe 4. Betrachten Sie folgende Knoten- und Kantenmenge. Zeichnen Sie den dazugehörigen Graphen und bestimmen Sie die dazugehörige Adjazenzmatrix.

$$G = (V, E); \quad V = \{2, 3, 4, 5, 6, 7, 9\};$$
$$E = \{\{7, 3\}, \{9, 6\}, \{2, 6\}, \{3, 2\}, \{3, 4\}, \{4, 2\}, \{5, 4\}, \{7, 9\}\}.$$

Kontrollaufgabe 5. Betrachten Sie folgende Adjazenzmatrix, zeichnen Sie den dazugehörigen Graphen und bestimmen Sie die Knoten- und Kantenmenge.

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

2.2 Breitensuche

2.2.1 Einführung

Modellierung mit Graphen: Wir haben gesehen dass ein Graph aus Knoten und Kanten besteht, wobei eine Kante jeweils zwei Knoten verbindet. Viele Probleme aus unserem Alltag können mit diesen Grundelementen repräsentiert werden. Zum Beispiel kann man das Netz des öffentlichen Verkehrs als Graphen darstellen, indem man alle Stationen als Knoten abbildet, und die Linien, die die Stationen verbinden, als gerichtete Kanten im Graphen, je nachdem in welche Richtung die Verkehrsmittel von der einen Station zur nächsten fahren können.

Traversieren eines Graphen: Es gibt viele verschiedene Algorithmen für Graphen, und eine wichtige Klasse von Algorithmen ist das Durchsuchen bzw. Traversieren von Graphen. Beim Traversieren eines Graphen werden die Knoten des Graphen besucht, und man bewegt sich dabei entlang den Kanten. Der Graph kann auch traversiert werden, um einen bestimmten Knoten im Graphen zu suchen.

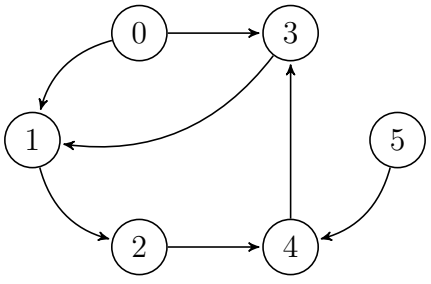


Abbildung 2.11: Ein Beispiel-Graph

Erreichbarkeit: Betrachten Sie den Graphen in Abb. 2.11. Ausgehend von einem Startknoten möchten wir untersuchen, ob ein Zielknoten erreichbar ist oder nicht. Ist der Knoten 2 von Knoten 0 aus erreichbar? Ist Knoten 5 auch erreichbar?
Um diese Fragen zu beantworten haben Sie vermutlich automatisch den Graphen betrachtet und visuell beurteilt, ob Sie vom Knoten 0 aus gewissen Kanten folgen können um den Zielknoten zu erreichen.

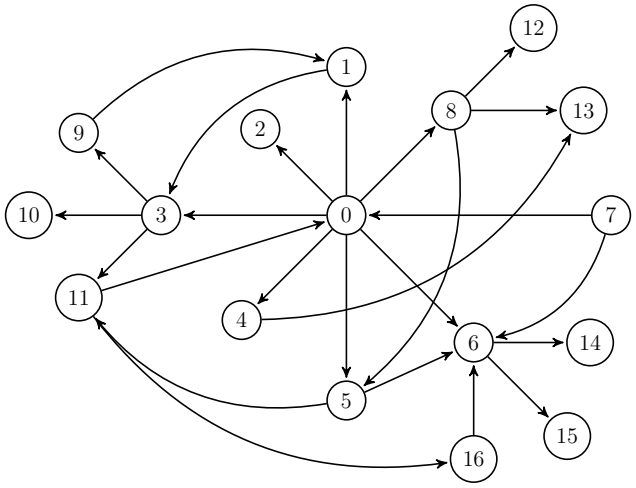


Abbildung 2.12: Ein etwas komplexerer Graph

Betrachten Sie nun den Graphen in Abb.2.12 und beurteilen Sie, ob man von Knoten 7 den Knoten 16 erreichen kann. Kann man von Knoten 5 den Knoten 1 erreichen?
Wie Sie merken, ist das Lösen der Aufgabe in einem etwas komplizierteren Graphen schon viel schwieriger. Stellen Sie sich jetzt aber vor, Sie haben einen Graphen mit tausenden von Knoten vor sich. Eine visuelle Beurteilung wird nun fast unmöglich und das Problem ist von Hand nicht mehr lösbar. Wir möchten also einen Algorithmus entwickeln, damit ein Computer das Problem für uns lösen kann. Ein Computer kann aber nicht auf eine visuelle Beurteilung zurückgreifen. Er braucht eine Repräsentation des Graphen, z.B. eine Adjazenzmatrix, mit welcher er arbeiten kann. Mit Hilfe der Adjazenzmatrix kann er dann jeweils für einen Knoten seine Nachbarsknoten "sehen". Um nun gewisse Probleme zu lösen braucht er zusätzlich zur Reprsentation als Adjazenzmatrix auch noch Instruktionen, die ihm mitteilen, wie das Problem überhaupt gelöst werden kann.

Algorithmus entwickeln: Wir möchten also einen Algorithmus entwickeln, der für einen Startknoten s in einem Graphen beurteilen kann, ob ein anderer Knoten von s aus erreichbar ist oder nicht. Bei der Beantwortung der Frage zu den Graphen in Abb. 2.11 und 2.12 haben Sie wahrscheinlich intuitiv begonnen, ein paar Wege auszuprobieren und je nachdem wieder zu verwerfen, z.B. wenn Sie gemerkt haben, dass Sie von einem Knoten aus nicht mehr weiter kommen. Damit das Problem von einem Computer gelöst werden kann braucht er aber ein klares, strukturierteres Vorgehen, um die Frage nach der Erreichbarkeit zu beantworten.
Es gibt verschiedene Ansätze für ein solches Vorgehen. Bevor wir nun ein solches Vorgehen entwickeln überlegen wir uns noch zusätzlich, ob wir ausser der Erreichbarkeit noch weitere Informationen haben wollen, da dies die Wahl unseres Vorgehens beeinflussen wird.
Stellen Sie sich zum Beispiel ein Labyrinth vor. Als Erstes möchten Sie natürlich wissen, ob es überhaupt einen Weg heraus gibt! Wenn der Ausgang tatsächlich erreichbar ist, möchten Sie vermutlich den kürzesten Weg aus dem Labyrinth heraus finden. Die Anforderung, den kürzesten Weg zu finden, taucht in vielen weiteren Problemen auf, und wir möchten hier deshalb einen Algorithmus entwickeln, der nicht nur die Frage der Erreichbarkeit beantworten kann, sondern auch, wie lange der kürzeste Weg zum Ziel ist.

Nicht zuletzt interessiert uns dann auch der konkrete Weg, der vom Startknoten zum Zielknoten führt.

Kürzester Weg: Im Graphen aus Abb. 2.11 ist Knoten 4 von Knoten 0 über Knoten 1 und 2 erreichbar (s. Abb. 2.13), aber auch über Knoten 3, 1, und 2 (s. Abb. 2.14). Der erste Weg ist jedoch der Kürzere: er traversiert nur 3 Kanten, während der zweite Weg 4 Kanten traversiert.

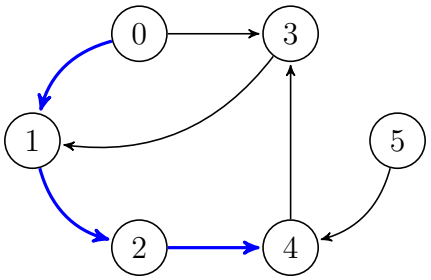


Abbildung 2.13: Weg von Knoten 0 zu Knoten 4 über Knoten 1 und 2

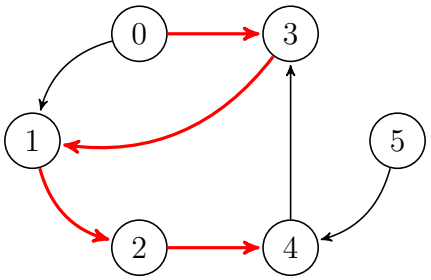


Abbildung 2.14: Weg von Knoten 0 zu Knoten 4 über Knoten 3, 1 und 2

Aufgabe 2.9. Wir möchten nun also ein Vorgehen entwickeln, welches in einem Graphen von einem Startknoten aus einen Knoten auf dem kürzesten Weg sucht und uns darüber informiert, ob er vom Startknoten aus erreichbar ist. Überlegen Sie sich, wie so ein Vorgehen aussehen könnte.

2.2.2 Algorithmus

Wie wir schon erwähnt haben gibt es mehrere Möglichkeiten für ein Vorgehen, welches einen Graphen traversiert und nach einem bestimmten Knoten sucht. In diesem Kapitel werden wir einen Algorithmus, nämlich die Breitensuche, Schritt für Schritt erarbeiten. Die Breitensuche ermöglicht es uns nämlich einen Knoten zu suchen, und, falls er gefunden wurde, seine kürzeste Distanz beziehungsweise auch den konkreten Weg zum Knoten ganz einfach herauszufinden.
Der Algorithmus für die Breitensuche benötigt drei Inputs: einen Graphen, einen Startknoten und einen Zielknoten. In einem ersten Schritt wird der Startknoten betrachtet. Im Graphen in Abb. 2.15 können vom Startknoten 0 aus alle seine Nachbarn, nämlich die Knoten 1 und 3 erreicht werden. Alle Nachbarn, die von diesem Startknoten aus erreichbar sind, sind in Abb. 2.15 grau dargestellt.

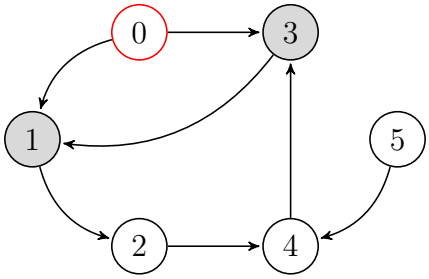


Abbildung 2.15: Startknoten (rot markiert) und seine Nachbarn (grau)

Wir möchten nun zum Beispiel wissen, ob Knoten 4 von diesem Startknoten aus erreichbar ist. Als Erstes prüfen wir also, ob sich der Knoten unter den Nachbarn des Startknotens befindet. Da dies nicht der Fall ist, müssen wir weitersuchen. Da wir mit Distanz 1 den Knoten nicht gefunden haben, müssen wir bei einer grösseren Distanz weitersuchen, also betrachten wir alle Knoten, die mit einer maximalen Distanz von 2 vom Startknoten aus erreichbar sind. Dies sind alle Knoten, die von den Nachbarn des Startknotens aus erreichbar sind. Wir wählen also zuerst den ersten Nachbarn, Knoten 1, und betrachten dessen Nachbarn.

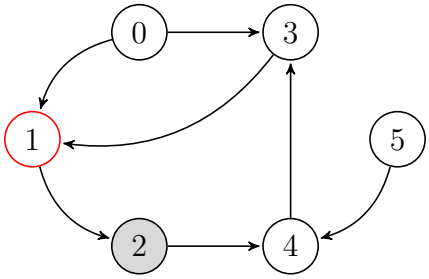


Abbildung 2.16: Aktuell betrachteter Knoten (rot markiert) und sein Nachbar (grau)

In Abb. 2.16 sehen wir, dass der Knoten 2 der einzige Nachbarknoten von Knoten 1 ist. Als Nächstes betrachten wir den 2. Nachbarn unseres Startknotens, nämlich Knoten 3, und betrachten dessen Nachbarn.

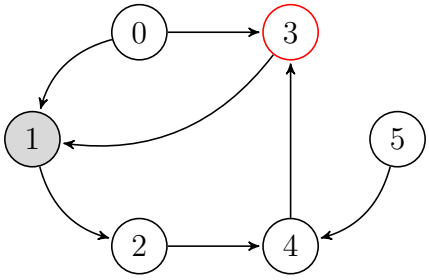


Abbildung 2.17: Aktuell betrachtete Knoten (rot markiert) und sein Nachbar (grau)

Der einzige Nachbarknoten von Knoten 3 ist Knoten 1, welchen wir aber gerade eben schon betrachtet haben und deshalb nicht mehr betrachten müssen. Wir müssen uns also irgendwie merken, welche Knoten wir schon betrachtet haben, damit wir sie nicht nochmals bearbeiten wenn wir später auf einem anderen Weg nochmals darauf stossen.

Wir erreichen dies, indem wir einen betrachteten Knoten speziell markieren, beispielsweise indem wir ihn schwarz einfärben. Gleichzeitig färben wir Knoten, die wir als Nachbarn eines Knoten schon entdeckt haben, aber noch nicht bearbeiten haben, grau ein. Abb. 2.18 zeigt den Graphen, nachdem der Startknoten (Knoten 0) betrachtet wurde (links), nachdem sein erster Nachbar (Knoten 1) betrachtet wurde (Mitte) und nachdem sein zweiter Nachbar (Knoten 3) betrachtet wurde (rechts).

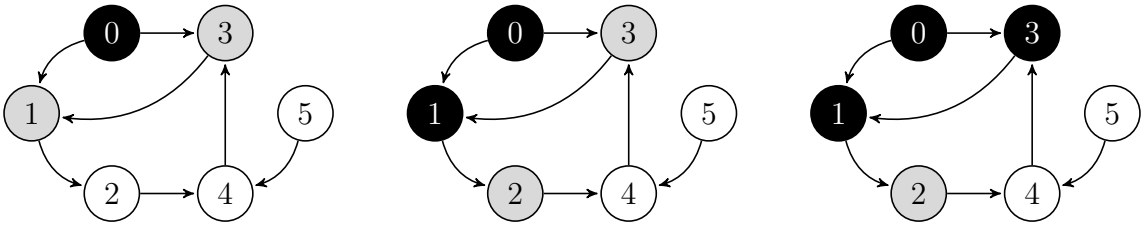


Abbildung 2.18: Verlauf der Einfärbungen

Im nächsten Durchlauf wird der nächste graue Knoten, also Knoten 2, betrachtet. Der einzige Nachbarknoten von Knoten 2 ist Knoten 4. Knoten 4 war der gesuchte Knoten - das heisst wir haben den Knoten gefunden!

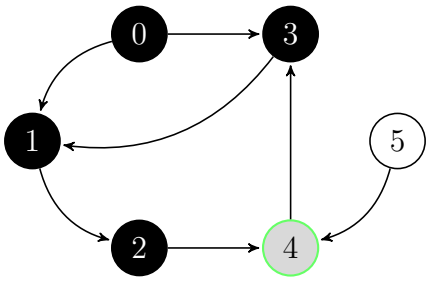


Abbildung 2.19: Zielknoten (grün markiert) wurde gefunden

Zusammenfassung: Wir möchten nun das Vorgehen zusammenfassen und in einem Algorithmus formulieren.
Wir beginnen mit einem Startknoten und färben ihn sogleich grau ein. Der nächste zu betrachtende Knoten ist jeweils der nächste graue Knoten. Da anfangs nur den Startknoten grau ist, beginnen wir gleich mit diesem Knoten. Falls der Knoten der gesuchte Knoten ist, haben wir ihn gefunden und können das Programm beenden. Andernfalls färben wir alle Nachbarn des Knotens grau ein. Den betrachteten Knoten selbst färben wir schwarz ein, um zu markieren, dass wir diesen nun bearbeitet haben. Wir fahren fort mit dem grauen Knoten, der als erstes grau gefärbt wurde und wiederholen das beschriebene Vorgehen.
Es ist wichtig, dass die grauen Knoten in der Reihenfolge betrachtet werden, in der sie eingefärbt werden. Damit stellen wir sicher, dass zuerst alle Knoten von einer kleineren Distanz zum Startknoten bearbeitet werden als die weiter entfernten, da jene auch erst später eingefärbt wurden. Man kann sich dies also wie eine Warteschlange vorstellen, in der die grauen Knoten eingereiht werden: die Knoten die zuerst eingereiht wurden, werden auch zuerst wieder von der Warteschlage entfernt. Dies ist das sogenannte FIFO Prinzip: First In, First Out.

Wenn wir die Suche beginnen, dann reihen wir den Startknoten in die Warteschlange (welche initial leer ist) ein.

Abbildung 2.20: Warteschlange nur mit dem Startknoten

Da der Startknoten der einzige Knoten in der Warteschlange ist, ist es der Knoten, der als nächstes genauer betrachtet wird. Wir entfernen ihn also aus der Warteschlange. Diese Operation wird dequeue genannt.

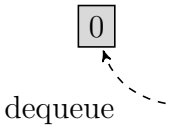


Abbildung 2.21: Entfernen des Startknotens aus der Warteschlange

Wenn wir nun den Startknoten betrachten, sehen wir seine Nachbarn und fügen diese nacheinander in die Warteschlange ein. In Abb 2.21 ist der Verlauf der Warteschlange nach diesen Schritten zu sehen.

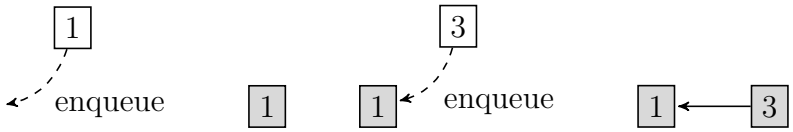


Abbildung 2.22: Verlauf der Warteschlange

Am Schluss wird der Startknoten schwarz eingefärbt. Der aktuell eingefärbte Graph und die Warteschlange sind in Abb. 2.23 zu sehen.
Als Nächstes wird der vorderste Knoten in der Warteschlange betrachtet. Er wird aus der Warteschlange entfernt, seine Nachbarn werden in die Warteschlange eingereiht (sofern sie nicht dem gesuchten Knoten entsprechen!) und schlussendlich wird der aktuell betrachtete Knoten schwarz eingefärbt. Dies wiederholen wir so lange, bis wir keine Knoten mehr in der Warteschlange haben.
Dieses Vorgehen führt uns nun zur Formulierung des Algorithmus für die Breitensuche. Das Ziel des Algorithmus ist es, herauszufinden, ob ein Knoten von einem Startknoten aus erreichbar ist.

Aufgabe 2.10. Benutzen Sie ihre Funktion NACHBARKNOTEN, welche Sie bereits im vorhergehenden Kapitel geschrieben haben, um Algorithmus 1 zu implementieren. Testen Sie Ihr Programm indem sie die Breitensuche für verschiedene Start- und Endknoten aufrufen.
Hinweis: Sie können das Einfärben der Knoten so umsetzen, dass sie in einer Liste für jeden Knoten seine Farbe speichern. Für die Umsetzung der Warteschlange der grauen Knoten können Sie ein Array verwenden. Mit dem Befehl $pop(x)$ kann ein Element an Stelle x im Array ausgelesen und entfernt werden. Der Befehl $pop(0)$ entspricht also der oben beschriebenen *dequeue* Operation.

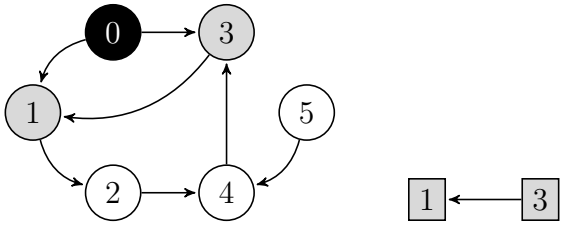


Abbildung 2.23: Graph mit der Warteschlange

Algorithmus 1 Breitensuche

```
1: procedure BREITENSUCHE(graph, start, ziel)
2:     if start == ziel then return true        ▷ Startknoten ist schon der Zielknoten
3:     start grau einfärben
4:     graueKnoten = [start]                    ▷ Warteschlange für graue Knoten
5:     while graueKnoten ≠ ∅ do
6:         current ← graueKnoten.dequeue()
7:         for all nachbar in NACHBARKNOTEN(graph, current) do
8:             if nachbar == ziel then return true          ▷ Zielknoten gefunden
9:             if nachbar nicht schwarz und nicht grau then ▷ Neu entdeckter Knoten
10:                nachbar grau einfärben
11:                graueKnoten.enqueue(nachbar)
12:        current schwarz einfärben                ▷ Knoten fertig bearbeitet
13:    return false                                 ▷ Zielknoten nicht gefunden
```

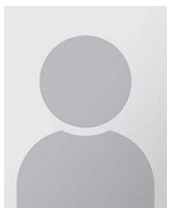
2.2.3 Anwendung und Erweiterung

Wir werden nun den entwickelten Algorithmus anhand eines konkreten Beispiels betrachten und weiterentwickeln.

Soziale Netzwerke: In der heutigen Gesellschaft sind soziale Netzwerke zu einem wichtigen Werkzeug geworden, um Beziehungen zu pflegen, z.B. über Berufsnetzwerke oder Freundesnetzwerke. Solche Netzwerke können wir als Graphen modellieren. Mit Hilfe von Graphenalgorithmen können gewisse Eigenschaften und Strukturen der Netzwerke erforscht werden.
In Abb. 2.24 sehen Sie einige Personen und ihre Vernetzungen zu anderen Personen. Wir möchten dieses kleine Personennetzwerk etwas genauer untersuchen, z.B. um herauszufinden, ob bestimmte Personen (auch über andere Personen) miteinander vernetzt sind. Dieses Netzwerk kann mit einem ungerichteten Graphen modelliert werden, bei dem die Knoten die Personen repräsentieren und die Kanten die Freundschaftsbeziehungen zwischen den Personen. Den daraus resultierenden Graphen sehen Sie in Abb. 2.25

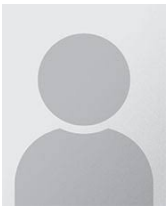
Aufgabe 2.11. Modellieren Sie diesen Graphen als Adjazenzmatrix und lesen Sie ihn in ihr bisheriges Programm ein. Benutzen Sie dafür folgende Reihenfolge: Tom für Zeile/-Spalte 0, John für Zeile/Spalte 1 und so weiter.

Aufgabe 2.12. Testen Sie mit ihrem Programm, ob es irgendeine Verbindung zwischen Anna und John gibt.



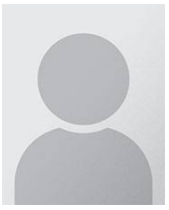
Freunde:
Karin
Tina

Tom



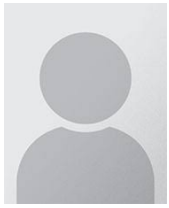
Freunde:
Tina
Karin

John



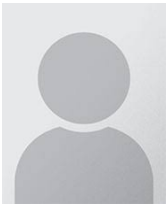
Freunde:
Lars
Laura

Jimmy



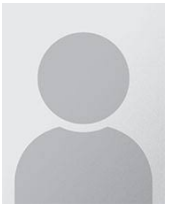
Freunde:
Tom
Tina
John

Karin



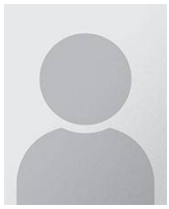
Freunde:
John
Lars
Anna
Laura
Karin
Tom

Tina



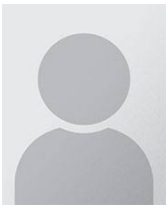
Freunde:
Jimmy
Laura
Tina

Lars



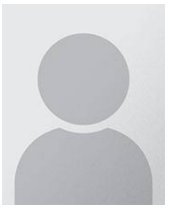
Freunde:
Tina
Simon

Anna



Freunde:
Anna
Lars

Simon



Freunde:
Tina
Lars
Jimmy

Laura

Abbildung 2.24: Ein kleines Personennetzwerk

Aufgabe 2.13. Tina überlegt sich schon seit einiger Zeit, das Netzwerk wieder zu verlassen. Falls sie tatsächlich das Netzwerk verlassen würde, gäbe es dann trotzdem noch eine Verbindung von Anna zu Jimmy? Und von Simon zu Tom?

Wir können mit unserem Programm jetzt also einfach beurteilen, ob es eine Verbindung zwischen zwei Personen gibt. Nun möchten wir aber gerne wissen, über wieviele Personen zwei Personen sozusagen miteinander verbunden sind. Wir müssen also unseren Algorithmus so anpassen, dass wir am Ende sagen können, wieviele Verbindungen eine Person von einer anderen Person mindestens entfernt ist. Beispiel: Wenn Anna mit Tina befreundet ist und Tina mit Lars, dann ist Lars über zwei Freundschaftsbeziehungen mit Anna verbunden.

Aufgabe 2.14. Überlegen Sie sich, wie der Algorithmus 1 angepasst werden könnte, sodass die Distanz zum Zielknoten gespeichert wird. **Hinweis:** Es reicht nicht, die Distanz in einer einzigen Variablen zu Speichern. Wieso nicht?

Wenn wir beim Startknoten starten, dann sind alle seine Nachbarn mit einer Distanz von 1 erreichbar. Da wir bei der Breitensuche immer zuerst alle Knoten einer gewissen Distanz abarbeiten, bevor wir die Distanz erhöhen, wissen wir, dass wenn wir einen Knoten neu entdecken, dass dieser nicht über irgendeinen Weg schneller erreichbar ist als über den Knoten, von dem aus wir ihn entdeckt haben. Das heisst also, ein Knoten kann mit einer Distanz vom Vorgängerknoten + 1 erreicht werden. Da wir in unserem Algorithmus beim Abarbeiten der grauen Knoten nicht mehr wissen, wie weit diese vom Startknoten entfernt sind, können wir nicht einfach in einer Variable die Distanz festhalten, auf der wir gerade suchen. Wir können aber zu jedem neu entdeckten Knoten seine Distanz speichern,

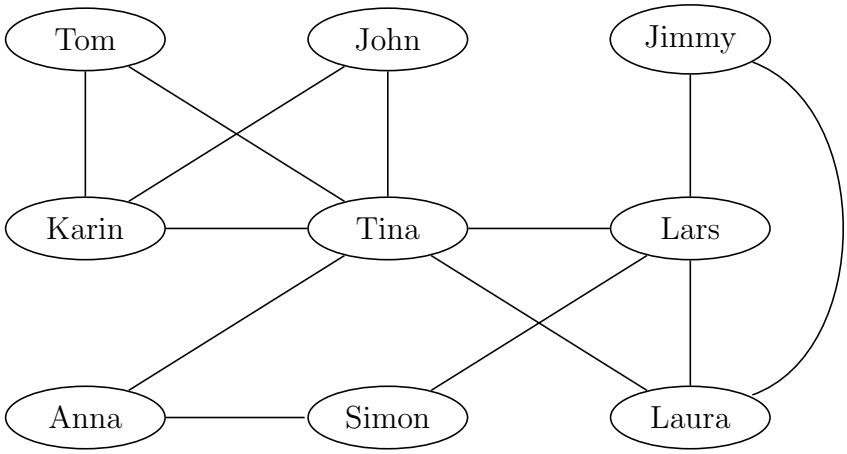


Abbildung 2.25: Modellierung des Personennetzwerks als Graph

damit wir beim Bearbeiten des Knotens wissen, wie weit entfernt vom Startknoten wir uns befinden. Wenn wir schlussendlich den Zielknoten erreichen, können wir ganz einfach seine Distanz auslesen.

Beispiel 2.8. In unserem Netzwerk aus Abb 2.25 möchten die minimale Distanz von Tom zu John herausfinden. Wir initialisieren dazu die Distanzen für jeden Knoten auf -1, um zu symbolisieren, dass wir noch keine konkreten Werte haben. (Beachten Sie die Nummerierung: Tom ist bei Index 0, John bei Index 1, ..., Laura bei Index 8.)

$$Distanzen = \Big[-1,\ \ -1,\ \ -1,\ \ -1,\ \ -1,\ \ -1,\ \ -1,\ \ -1,\ \ -1\Big]$$

Wenn wir jetzt bei Tom als Startknoten beginnen, können wir seine Distanz auf 0 setzen. Seine Nachbarn (Karin und Tina) sind dann jeweils mit einer Distanz von 1 von Tom entfernt:

$$Distanzen = \Big[0,\ \ -1,\ \ -1,\ \ 1,\ \ 1,\ \ -1,\ \ -1,\ \ -1,\ \ -1\Big]$$

Als Nächstes würden wir die Nachbarn von Karin betrachten. Hier ist John der einzige noch nicht betrachtete Nachbar und bekommt die Distanz von Karin + 1, also 2. Dies ist die gesuchte Distanz von Tom zu John.

$$Distanzen = \Big[0,\ \ 2,\ \ -1,\ \ 1,\ \ 1,\ \ -1,\ \ -1,\ \ -1,\ \ -1\Big]$$

Wir können nun also den Algorithmus 1 erweitern mit dieser Speicherung der Distanzen. In Algorithmus 2 sehen Sie den erweiterten Algorithmus, wobei die neu dazugekommenen Schritte rot markiert sind.

Aufgabe 2.15. Ergänzen Sie Ihr Programm so, dass am Schluss die kürzeste Distanz von einem Startknoten zu einem Zielknoten ausgegeben wird, falls der Zielknoten vom Startknoten aus erreichbar ist.

Wir haben nun also ein Programm, das uns sagen kann, wie weit entfernt ein Startknoten von einem Zielknoten mindestens liegt (falls der Zielknoten überhaupt vom Startknoten aus erreichbar ist). Wir möchten unseren Algorithmus nun noch etwas weiter ausbauen, so dass wir am Schluss, falls wir den Zielknoten gefunden haben, herausfinden können, über

Algorithmus 2 Breitensuche

```
1:  procedure BREITENSUCHE(graph, start, ziel)
2:      if start == ziel then return true          ▷ Startknoten ist schon der Zielknoten
3:      start grau einfärben
4:      graueKnoten ← [start]                      ▷ Warteschlange für graue Knoten
5:      distanzen = array[Anzahl Knoten]
6:      while graueKnoten ≠ ∅ do
7:          current ← graueKnoten.dequeue()
8:          for all nachbar in NACHBARKNOTEN(graph, current) do
9:              if nachbar == ziel then return true          ▷ Zielknoten gefunden
10:             if nachbar nicht schwarz und nicht grau then  ▷ Neu entdeckter Knoten
11:                 nachbar grau einfärben
12:                 graueKnoten.enqueue(nachbar)
13:                 distanzen[nachbar] = distanzen[current] + 1
14:         current schwarz einfärben                  ▷ Knoten fertig bearbeitet
15:     return false                                   ▷ Zielknoten nicht gefunden
```

welche Knoten der kürzeste Weg vom Startknoten zum Zielknoten führt. Wir müssen uns also während dem Traversieren irgendwie merken, wie wir zu einem aktuellen Knoten gelangt sind. Es ist möglich, dass es mehrere Wege von einem Startknoten zu einem Zielknoten gibt, aber wir sind stets am kürzesten Weg interessiert. Das heisst, wir können während der Suche, beim Abarbeiten eines Knotens, einfach speichern, von welchem Knoten aus wir zu diesem Knoten gelangt sind. Dann können wir vom Schlussknoten aus Knoten für Knoten zurückverfolgen, auf welchem Weg wir zum Ziel gelangt sind.

Beispiel 2.9. In unserem Netzwerk aus Abb. 2.24 ist Tom mit Karin befreundet und Karin mit John. Tom ist also über Karin mit John verbunden. Ein möglicher Pfad wäre also Tom - Karin - John. Einen kürzeren Pfad gibt es nicht, da Tom nicht direkt mit John befreundet ist. Den Pfad können wir konstruieren, indem wir uns während der Breitensuche merken, dass wir von Tom aus auf Karin gestossen sind und von Karin auf John.

Für unseren Algorithmus bedeutet das, dass wir für jeden Knoten, den wir entdecken, mitspeichern, von welchem Knoten aus wir ihn entdeckt haben.

Beispiel 2.10. Wir starten wiederum in unserem Netzwerk aus Abb 2.25 und möchten den Weg, welcher mit minimaler Distanz von Tom zu John führt, herausfinden. Wir initialisieren dazu, analog zur Matrix mit den Distanzen, die Vorgänger für jeden Knoten auf -1, um zu symbolisieren, dass wir noch keine konkreten Werte haben.

$$Vorgänger = \begin{bmatrix} -1, & -1, & -1, & -1, & -1, & -1, & -1, & -1, & -1 \end{bmatrix}$$

Da Tom als Startknoten keinen Vorgänger hat, lassen wir dort die -1. Für Toms Nachbarn (Karin und Tina) jedoch tragen wir Tom bzw. den Index von Toms Knoten in die Matrix ein.

$$Vorgänger = \begin{bmatrix} -1, & -1, & -1, & 0, & 0, & -1, & -1, & -1, & -1 \end{bmatrix}$$

Der einzige noch nicht betrachtete Nachbar von Karin ist John, also tragen wir bei John den Index von Karins Knoten ein, nämlich 3.

$$Vorgänger = \Big[-1,\ \ 3,\ \ -1,\ \ 0,\ \ 0,\ \ -1,\ \ -1,\ \ -1,\ \ -1\Big]$$

Um nun den Weg vom Startknoten zum Zielknoten zu berechnen, können wir vom Zielknoten her jeweils die Vorgängerknoten verfolgen. Bei John (Index 1) hatten wir also also Vorgängerknoten Karin (3), bei Karin (Index 3) hatten wir Tom (0), das heisst der Weg vom Startknoten zum Zielknoten ist Tom, Karin, John.

In Algorithmus 3 sehen Sie die Erweiterung mit den Vorgängerknoten in den rot markierten Zeilen.

Algorithmus 3 Breitensuche

```
1:  procedure BREITENSUCHE(graph, start, ziel)
2:      if start == ziel then return true          ▷ Startknoten ist schon der Zielknoten
3:      start grau einfärben
4:      graueKnoten ← [start]                      ▷ Warteschlange für graue Knoten
5:      distanzen = array[Anzahl Knoten]
6:      vorgaenger = array[Anzahl Knoten]
7:      while graueKnoten ≠ ∅ do
8:          current ← graueKnoten.dequeue()
9:          for all nachbar in NACHBARKNOTEN(graph, current) do
10:             if current = ziel then return true          ▷ Zielknoten gefunden
11:             if nachbar nicht schwarz und nicht grau then   ▷ Neu entdeckter Knoten
12:                 nachbar grau einfärben
13:                 graueKnoten.enqueue(nachbar)
14:                 distanzen[nachbar] = distanzen[current] + 1
15:                 vorgaenger[nachbar] = current   ▷ Von wo der Knoten entdeckt wurde
16:         current schwarz einfärben                  ▷ Knoten fertig bearbeitet
17:     return false                                   ▷ Zielknoten nicht gefunden
```

Aufgabe 2.16. Schreiben Sie eine Funktion WEG_ZUM_ZIEL, welche als Input den Zielknoten und einen Array von Vorgängerknoten hat, und als Output eine Liste der Knoten ausgibt, über welche der Weg vom Startknoten zum Zielknoten führt, und zwar in der Reihenfolge in der sie vom Startknoten aus traversiert werden.

Aufgabe 2.17. Benutzen Sie Ihre Funktion WEG_ZUM_ZIEL und ergänzen Sie Ihr Programm so, dass am Schluss zusätzlich zur Distanz der Weg ausgegeben wird, welcher vom Startknoten zum Zielknoten führt, falls dieser vom Startknoten aus erreichbar ist.

2.2.4 Beispiele

Betrachten Sie den Ausschnitt des ZVV-Netzes in Abb. 2.26.
Sie möchten wissen, wie man am schnellsten nur mit Tramfahren von der ETH Zürich zum Bahnhof Enge gelangt. Sie nehmen an, dass es immer ungefähr gleich lange dauert um von einer Tramstation zur nächsten zu fahren, aber dass der grösste Teil der Zeit an den Stationen selbst verloren geht, weil immer viele Leute ein- und aussteigen wollen. Sie möchten also einen Weg finden, bei welchem man an möglichst wenigen Tramstationen vorbeifahren muss, damit man möglichst schnell am Ziel ankommt.

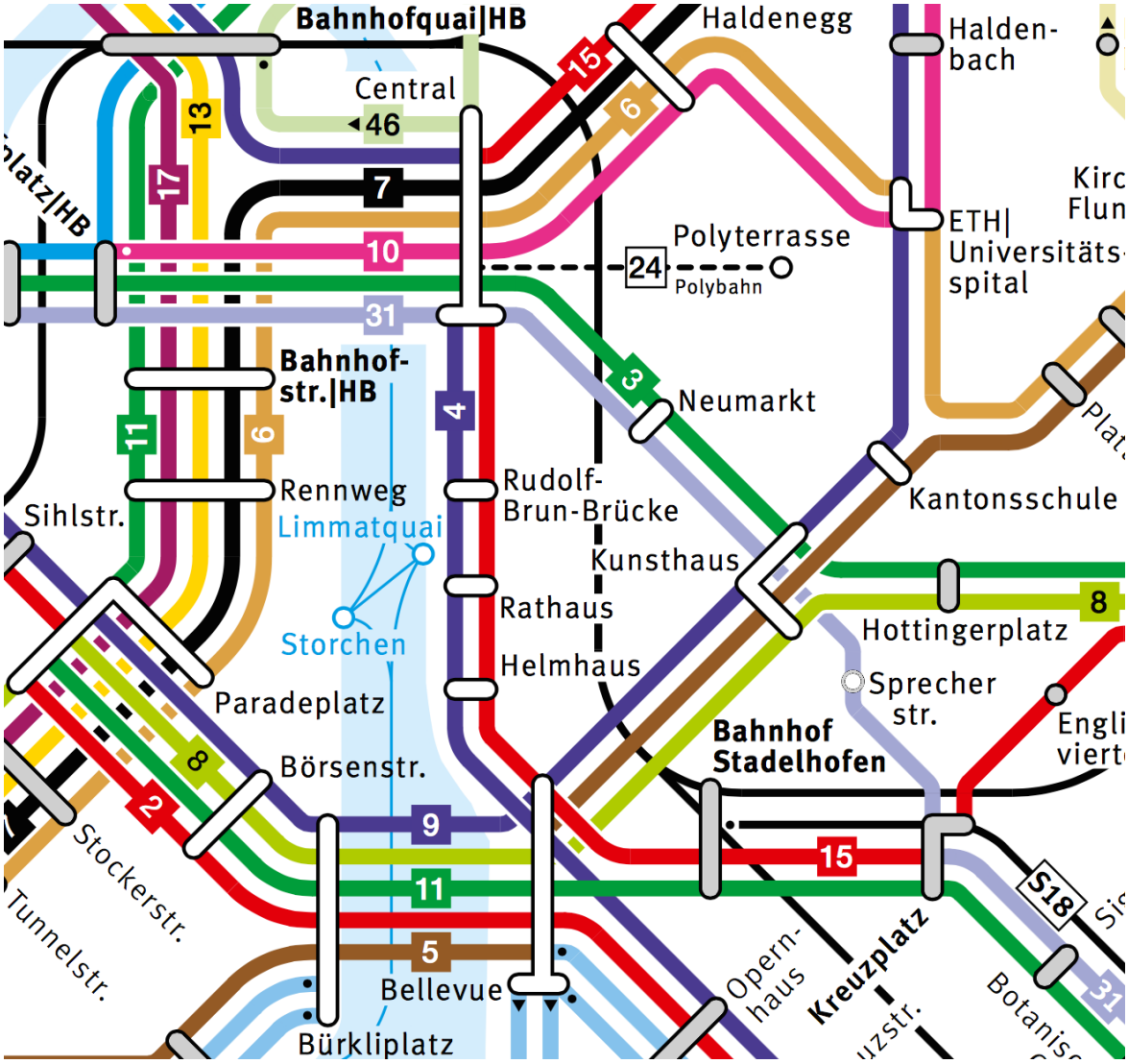


Abbildung 2.26: Ausschnitt aus dem ZVV Liniennetzplan

Aufgabe 2.18. Überlegen Sie sich, wie Sie dieses Problem mit Hilfe von einem Graphen modellieren können und implementieren Sie den entsprechenden Graphen als Adjazenzmatrix. Betrachten Sie dabei der Einfachheit halber nur Tramlinien und nur weisse Stationen.
Handelt es sich um einen gerichteten oder ungerichteteten Graphen?

Aufgabe 2.19. Stellen Sie sich vor, am Central werden Bauarbeiten durchgeführt und das Bellevue muss temportär für den Trambetrieb geschlossen werden. Erreichen Sie von der ETH aus dennoch den Paradeplatz? Um diese Frage zu beantworten, können Sie den Netzplan studieren und auf ein visuelles Urteil zurückgreifen. Wir möchten dies aber nun mit unserem Programm beurteilen, welches zum Beispiel in einer Farhplan-App eingesetzt werden könnte. Sie haben gelernt, dass ein Computer nicht auf eine visuelle Beurteilung zurückgreifen kann, sondern dass er dazu eine Repräsentation wie die Adjazenzmatrix benutzen muss. Wie verändert sich die Adjazentmatrix, wenn das Central geschlossen wird? Verwenden Sie nun ihr Programm, um zu beurteilen, ob der Paradeplatz auch bei geschlossenem Central von der ETH aus erreichbar ist.

Aufgabe 2.20. Wieviele Tramstationen müssen mindestens gefahren werden, um von

der ETH aus zum Paradeplatz zu gelangen? Verwenden Sie dazu Ihr Programm.

(a) Falls das Central gesperrt ist.

(b) Falls das Central wieder ohne Einschränkungen befahrbar ist.

Aufgabe 2.21. Auf welchem Weg kommen Sie am schnellsten von der ETH zum Paradeplatz? Verwenden Sie dazu Ihr Programm.

(a) Falls das Central gesperrt ist.

(b) Falls das Central wieder ohne Einschränkungen befahrbar ist.

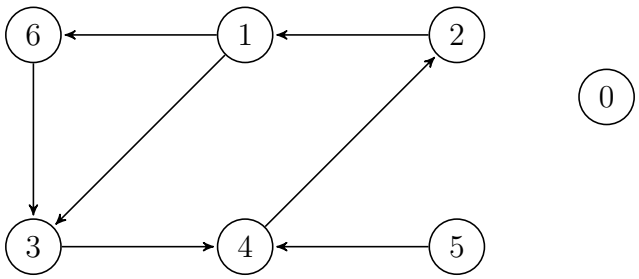
2.3 Lösungen

Lösung zu Aufgabe 2.1.

$$G = (V, E) \text{ mit: } V = \{0, 1, 2, 3, 4\};$$

$$E = \{(0,1), (0,2), (1,2), (2,0), (3,1), (3,4), (4,0)\}.$$

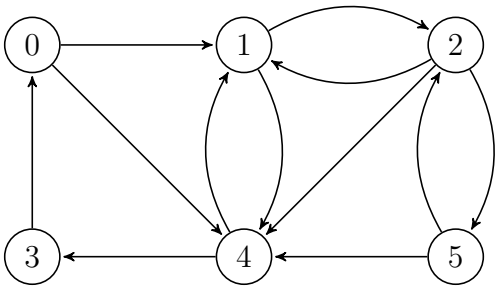
Lösung zu Aufgabe 2.2.



Lösung zu Aufgabe 2.3.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Lösung zu Aufgabe 2.4.



Lösung zu Aufgabe 2.5.

```
A = [ [0, 1, 0, 0, 1], [1, 0, 1, 0, 0], [1, 1, 0, 1, 1],
      [0, 0, 1, 0, 1], [1, 0, 1, 1, 0] ]
print A[1]
print A[2][3]
```

Die Ausgabe gibt:

```
[1, 0, 1, 0, 0]
1
```

Lösung zu Aufgabe 2.6. Der Knoten 3 hat den Nachbarknoten: 1.
Der Knoten 1 hat den Nachbarknoten: 2.
Der Knoten 0 hat die Nachbarknoten: 1 und 2.

Lösung zu Aufgabe 2.7. Der Knoten 0 hat die Nachbarknoten: 1 und 4.
Der Knoten 2 hat die Nachbarknoten: 0,1 und 3.
Der Knoten 4 hat die Nachbarknoten: 0,2 und 3.

Lösung zu Aufgabe 2.8.

```
def nachbarsknoten(graph, node):
    n = len(graph[0])
    neighbors = []
    for i in range(n):
        if graph[node][i] == 1:
            neighbors.append(i)
    return neighbors
```

Lösung zu Aufgabe 2.9. Wenn wir beim Startknoten beginnen, möchten wir zuerst alle Knoten absuchen, welche mit minimaler Distanz vom Startknoten aus erreichbar sind, also die direkten Nachbarsknoten vom Startknoten. Wenn wir in kürzester Distanz den gesuchten Knoten nicht gefunden haben müssen wir die nächste grössere Distanz in Kauf nehmen, in der Hoffnung, den Knoten dort zu finden. Wenn wir so vorgehen und den Knoten finden wissen wir nämlich, dass wir ihn auf dem kürzesten möglichen Weg gefunden haben. *Das Vorgehen und der Algorithmus wird in Kapitel 2.2.2* detaillierter erläutert.

Lösung zu Aufgabe 2.10.

```
def breitensuche (graph, start, ziel):
    if start == ziel:
        return True
    n = len(graph[0])
    graueKnoten = [start]
    farben = ['weiss'] * n
    farben[start] = 'grau'
    while graueKnoten != []:
        current = graueKnoten[0]
        for nachbar in nachbarsknoten(graph, current):
            if nachbar == ziel:
                return True
            if farben[nachbar] != 'grau' and farben[nachbar] != 'schwarz':
```

```
            farben[nachbar] = 'grau'
            graueKnoten += [nachbar]
        farben[current] = 'schwarz'
        graueKnoten = graueKnoten[1:]
    return False
```

Lösung zu Aufgabe 2.11.

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Lösung zu Aufgabe 2.12. Es gibt eine Verbindung: das Programm sollte $True$ zurückgeben.

Lösung zu Aufgabe 2.13. Es gäbe trotzdem noch eine Verbindung zwischen Anna und Jimmy, jedoch nicht zwischen Simon und Tom.

Lösung zu Aufgabe 2.14. Wir speichern die Distanz zu jedem Knoten. Das heisst, der Startknoten bekommt die Distanz 0. Jeder weitere Knoten, der vom aktuellen Knoten aus entdeckt wird (und grau gefärbt wird) bekommt die Distanz des aktuellen Knotens plus 1.

Lösung zu Aufgabe 2.15.

```
def breitensuche (graph, start, ziel):
    if start == ziel:
        print "Distanz : ", 0
        return True
    n = len(graph[0])
    graueKnoten = [start]
    farben = ['weiss'] * n
    farben[start] = 'grau'
    distanzen = [-1] * n
    while graueKnoten != []:
        current = graueKnoten.pop(0)
        for nachbar in nachbarsknoten(graph, current):
            if farben[nachbar] != 'grau' and farben[nachbar] != 'schwarz':
                distanzen[nachbar] = distanzen[current] + 1
                if nachbar == ziel:
                    print "Distanz: ", distanzen[ziel]
                    return True
```

```
            farben[nachbar] = 'grau'
            graueKnoten += [nachbar]
        farben[current] = 'schwarz'
    return False
```

Lösung zu Aufgabe 2.16.

```
def weg_zum_ziel(vorgaenger, ziel):
    knoten = ziel
    weg = [ziel]
    while vorgaenger[knoten] != -1:
        weg = [vorgaenger[knoten]] + weg
        knoten = vorgaenger[knoten]
    return weg
```

Lösung zu Aufgabe 2.17.

```
def breitensuche (graph, start, ziel):
    if start == ziel:
        print "Distanz: ", 0
        print "Weg zum Ziel: schon am Ziel!"
        return True
    n = len(graph[0])
    graueKnoten = [start]
    farben = ['weiss'] * n
    farben[start] = 'grau'
    distanzen = [-1] * n
    vorgaenger = [-1] * n
    while graueKnoten != []:
        current = graueKnoten[0]
        for nachbar in nachbarsknoten(graph, current):
            if nachbar == ziel:
                print "Distanz: ", distanzen[ziel]+1
                print "Weg zum Ziel: ", weg_zum_ziel(vorgaenger, ziel)
                return True
            if farben[nachbar] != 'grau' and farben[nachbar] != 'schwarz':
                distanzen[nachbar] = distanzen[current] + 1
                vorgaenger[nachbar] = current
                farben[nachbar] = 'grau'
                graueKnoten += [nachbar]
        farben[current] = 'schwarz'
        graueKnoten = graueKnoten[1:]
    return False
```

Lösung zu Aufgabe 2.18. Die Stationen werden als Knoten abgebildet, die Linien als Kanten. Da im Bild alle Tramlinien in beide Richtungen befahren werden können, können wir das Problem als ungerichteten Graphen darstellen, mittels folgender Adjazenzmatrix:

	ETH/Universitätsspital	Haldenegg	Kantonsschule	Kunsthaus	Neumarkt	Central	Rudolf-Brun-Brücke	Rathaus	Helmhaus	Bellevue	Bürkliplatz	Bahnhofstr./HB	Rennweg	Börsenstrasse	Paradeplatz
ETH/Universitätsspital	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Haldenegg	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
Kantonsschule	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Kunsthaus	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0
Neumarkt	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
Central	0	1	0	0	1	0	1	0	0	0	0	1	0	0	0
Rudolf-Brun-Brücke	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0
Rathaus	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0
Helmhaus	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0
Bellevue	0	0	0	1	0	0	0	0	1	0	1	0	0	0	0
Bürkliplatz	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
Bahnhofstr./HB	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0
Rennweg	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
Börsenstrasse	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
Paradeplatz	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0

Lösung zu Aufgabe 2.19. Die Adjazenzmatrix ändert sich so, dass alle Einträge in der Zeile und in der Spalte vom Central 0 werden.
Der Paradeplatz ist von der ETH aus auch erreichbar wenn das Central gesperrt ist, und zwar mit folgendem Weg: ETH - Kantonsschule - Kunsthaus - Bellevue - Börsenstrasse - Bürkliplatz - Paradeplatz.

Lösung zu Aufgabe 2.20.

(a) 7

(b) 6

Lösung zu Aufgabe 2.21.

(a) ETH - Kantonsschule - Kunsthaus - Bellevue - Bürkliplatz - Börsenstrasse - Paradeplatz

(b) ETH - Haldenegg - Central - Bahnhofstrasse/HB - Rennweg - Paradeplatz

Literaturverzeichnis

[CLR04] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L.: *Algorithmen - Eine Einführung*. Oldenbourg Wissensch.Vlg, 2004. – ISBN 3486275151

[Hro14] Hromkovič, J.: *Theoretische Informatik*. Springer Fachmedien Wiesbaden, 2014. – ISBN 978–3–658–06432–7

[OW12] Ottmann, T. ; Widmayer, P.: *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag Heidelberg 2012, 2012. – ISBN 978–3–8274–2803–5