Fachdidaktik Informatik

Semesterbegleitende Übung

Breitensuche

DRAFT

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

HS 2017

Inhaltsverzeichnis

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.
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 und sind für den Informatikunterricht im Ergänzungsfach Informatik geeignet, idealerweise für Schüler mit mathematisch/naturwissenschaftlichem Schwerpunkt. Es wird davon ausgegangen, dass die Schüler bereits einige Grundlagen der Informatik kennengelernt haben, insbesondere den Begriff Algorithmus. Zum Vorwissen der Schüler sollte ausserdem der Begriff des Graphen gehören. Zusätzlich sollen die Schüler wissen, wie Programme modular aufgebaut werden können und bereits in der Lage sein, einfache Programme in TigerJython zu schreiben. Grundlegende Kenntnisse von elementaren Datentypen und Arrays sowie von Kontrollstrukturen werden für das Lösen der Programmieraufgaben vorausgesetzt. Die vorliegenden Unterlagen sollten also im zweiten Quartal des Ergänzungsfaches Informatik eingesetzt werden können.

1.2.2 Vorwissen

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

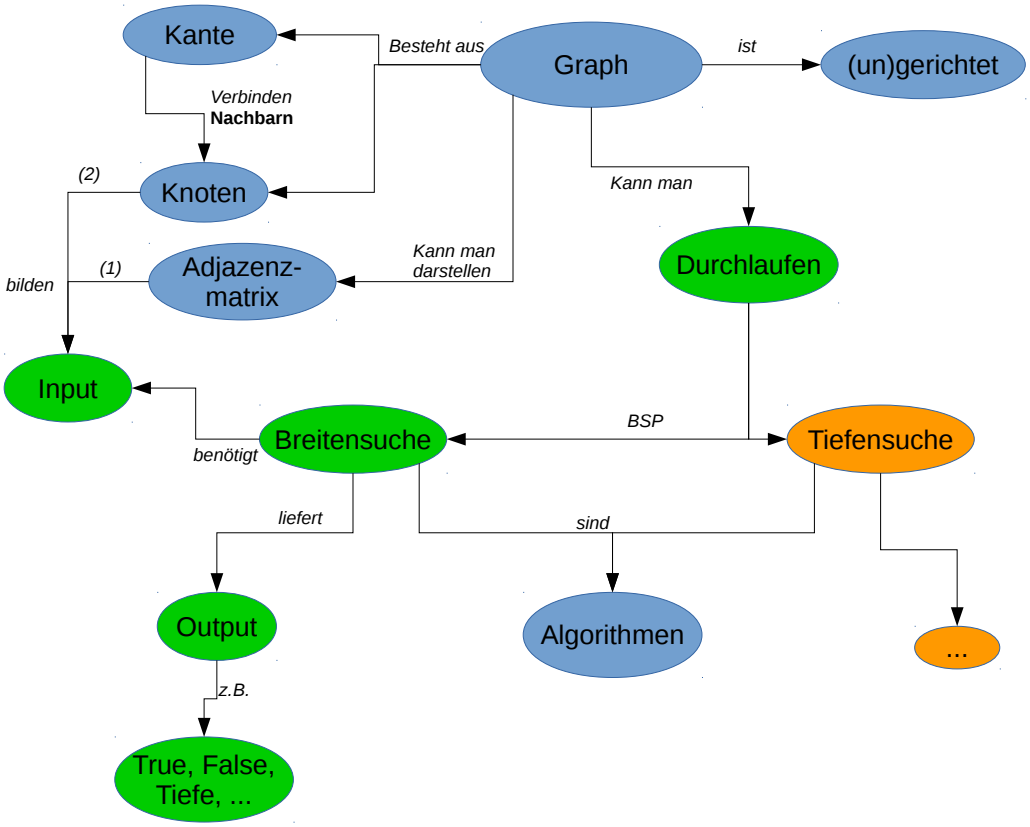


Abbildung 1.1: Concept-Map zum Thema Breitensuche. Blaue Konzepte und grüne Konzept werden in dieser Arbeit behandelt, wobei die Blauen das Vorwissen abbilden und Grüne neue Konzepte sind. Orangene Konzepte bilden einen Ausblick auf Konzepte, welche im Rahmen dieser Arbeit nicht mehr behandelt werden.

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. Wir nehmen an, dass die Schüler auch schon ein grundlegendes Verständnis von Komplexitätsanalyse besitzen.
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.
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 sich auf Graphen bewegen kann. Eine Möglichkeit bietet die Breitensuche, welche einen naiven Einstieg bildet.

1.3.2 Dispositionsziel

Die SuS wissen, dass man gewisse Probleme mit Hilfe von Graphen modellieren und mit Graphenalgorithmen lösen kann. Sie können Probleme analysieren und beurteilen, ob sie mit Hilfe von einer 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 Knoten 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 (in Bezug auf Anzahl Knoten) von einem Knoten zu einem anderen Knoten finden
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 Inhalt

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

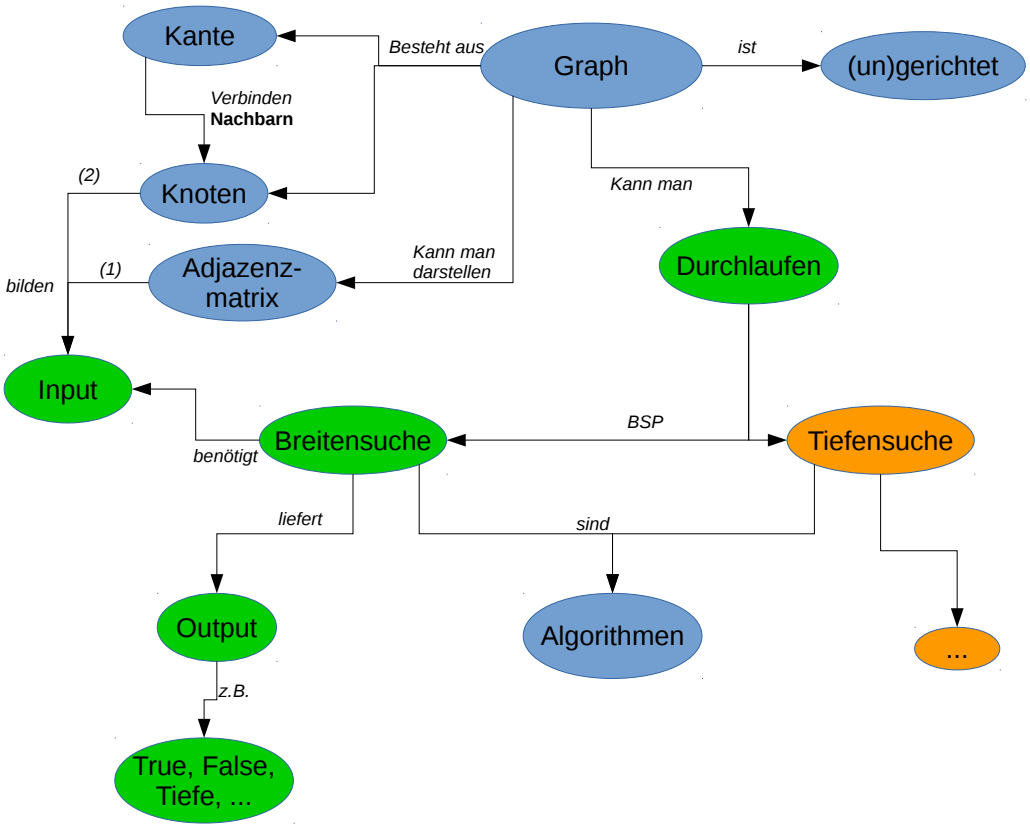


Abbildung 2.1: Concept-Map zum Thema Breitensuche. Blaue Konzepte bilden das Vorwissen ab. Grüne Konzepte werden in dieser Arbeit eingeführt und behandelt. Orangene Konzepte bilden einen Ausblick auf Konzepte, welche im Rahmen dieser Arbeit nicht mehr behandelt werden.

Den Begriff Graph haben Sie schon einmal gehört. Mit einem Graphen kann man verschiedene Objekte mit einander verbinden. 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.
Der Aufbau dieses Kapitels ist so gehalten, dass Sie im ersten Abschnitt nochmal die Grundlagen zu Graphen wiederholen und festigen. Dabei geht es vor allem um verschiedene Darstellungen von Graphen und das Konzept von Nachbarn. Im zweiten Abschnitt sollen dann Grundlagen des Modellierens von Problemen mit Hilfe von Graphen und das Lösen der Probleme mit Hilfe von Algorithmen auf Graphen vermittelt werden. Insbesondere werden Problemstellungen betrachtet, die sich mit Hilfe eines neuen Algorithmus (*Breitensuche*), lösen lassen. Der Algorithmus für die Breitensuche wird dabei Schritt für Schritt erarbeitet, erweitert und implementiert.
In der Concept-Map (s. Abb. 2.1) können Sie die verschiedenen Konzepte und ihren Zusammenhang für das folgende Kapitel überblicken. Mit der Farbe Blau markierte Konzepte sollten schon bekannt sein, aber werden nochmal wiederholt. Grün markiert sind neue Konzepte, die in dieser Einheit behandelt werden und orange sind Konzepte, die einen Ausblick bilden und nicht mehr behandelt werden.

2.1 Graphen

2.1.1 Begriffe

Ein Graph ist eine abstrakte Struktur, die eine Menge von Objekten zusammen mit den zwischen diesen Objekten bestehenden Verbindungen 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).
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.

2.1.2 Defintionen

Definition 2.1. Mathematisch besteht ein **Graph** G 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: $\{v_1, v_2\} = \{v_2, v_1\}$.

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: $(v_1, v_2) \neq (v_2, v_1)$.

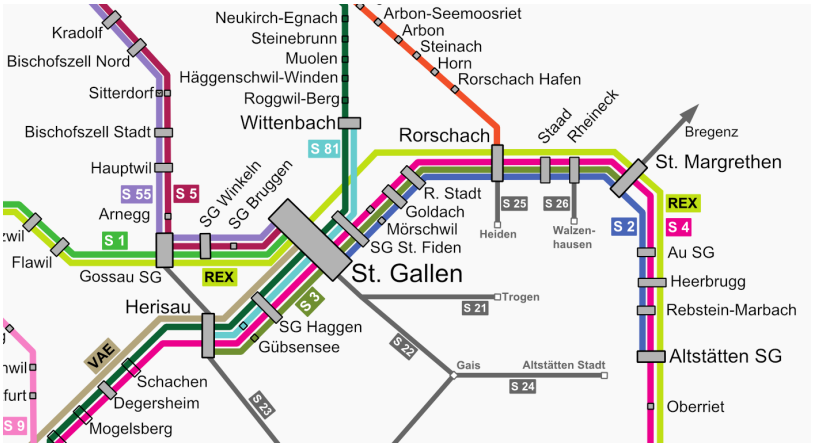


Abbildung 2.2: Darstellung eines Ausschnitts des Sankt Galler S-Bahn-Netzes als (ungerichteter) Graph. Jede Station bildet dabei einen Knoten und die Verbindung zwischen den Knoten ist eine Kante.

Beispiel 2.2. Die Abblidung 2.3 zeigt einen ungerichteten Graphen mit 5 Knoten und einen gerichteten Graphen mit 6 Knoten.

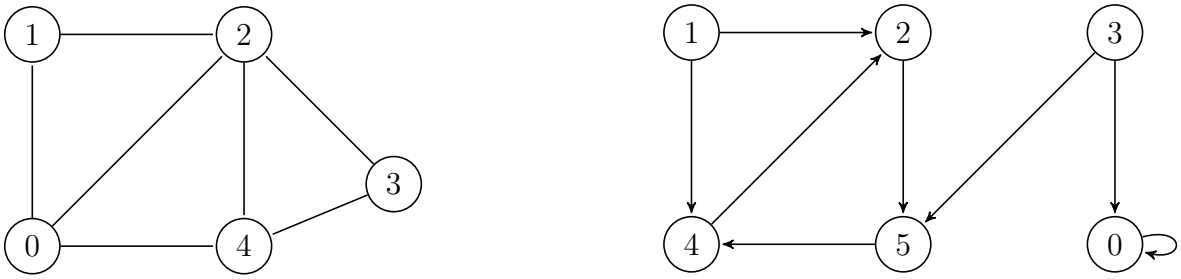


Abbildung 2.3: *Links:* Ein ungerichteter Graph mit 5 Knoten. *Rechts:* Ein gerichteter Graph mit 6 Knoten.

Die mathematische Darstellung der zwei Graphen lautet:

$$\text{Graph links:} \quad G_l = (V_l, E_l); \quad V_l = \{0,1,2,3,4\};$$
$$E_l = \{\{1,2\},\{1,0\},\{2,0\},\{2,4\},\{2,3\},\{3,4\},\{4,0\}.\}$$

$$\text{Graph rechts:} \quad G_r = (V_r, E_r); \quad V_r = \{0,1,2,3,4,5\};$$
$$E_r = \{(1,2),(1,4),(2,5),(3,5),(3,0),(4,2),(5,4),(0,0)\}.$$

Aufgabe 2.1. Bilden Sie für folgenden Graph (Abb. 2.4) die entsprechende mathematische Darstellung $G = (V, E)$.

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

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

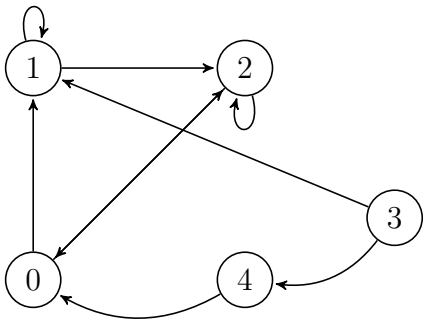


Abbildung 2.4: Ein gerichteter Graph.

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 $\{i, j\}$.

Beispiel 2.3. Die Adjazenz-Matrizen (A_l und A_r) zu den zwei Graphen in Abb. 2.3 lauten:

$$A_l = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}; \quad A_r = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}.$$

Aufgabe 2.3. Bilden Sie für den Graphen aus Abbildung 2.4 die entsprechende Adjazenzmatrix $G = (V, E)$.

Aufgabe 2.4. Zeichnen Sie zu folgender Adjazenzmatrix den entsprechenden Graphen.

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

Aufgabe 2.5. Damit man einen Graphen in einer Textdatei speichern und lesen kann, benutzt man häufig ';' (Semicolon) um die Zahlen einer Zeile zu trennen. Mit einem Absatz wird eine neue Zeile der Matrix bestimmt. Schreiben Sie ein Programm NicePrint(*graph*), das die Adjazenzliste eines Graphen auf den Bildschirm ausgibt. Benutzen Sie dafür das vorgegebene Programm, welches Graphen aus einer Textdatei einlesen kann und überprüfen Sie die Ausgabe mit den Textdateien.

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

Beispiel 2.4. Im Graphen der Abbildung 2.3 links hat der Knoten 2 die Nachbarn 1,0,4 und 3. Dies kann man auch aus der Adjazenzmatrix A_l ablesen: Die zweite Reihe hat dort bei den Spalten für Knoten 1,3,4 und 0 eine eins stehen.
Im Gegensatz dazu hat der Knoten 2 im Graphen der Abbildung 2.3 rechts nur den Nachbarknoten 5. Auch dies kann man wieder aus der zweiten Reihe der Adjazenzmatrix A_r ablesen.

Aufgabe 2.6. Welche Nachbarn hat der Knoten 3, der Knoten 1 und der Knoten 0 des Graphen in Abbildung 2.4.

Aufgabe 2.7. Implementieren Sie eine Funktion Nachbarknoten(*graph*, *knoten*): Sie hat als Input einen Graphen und einen Knoten und gibt als Output eine Liste von Nachbarknoten des Knotens im Graphen 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 Unterschied zwischen gerichteten und ungerichteten Graphen und können die Nachbarn eines Knoten in einem Graphen 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. 2.5) und bestimmen Sie seine Knoten- und Kantemenge und bestimmen Sie zusätzlich die Adjazenzmatrix.

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

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

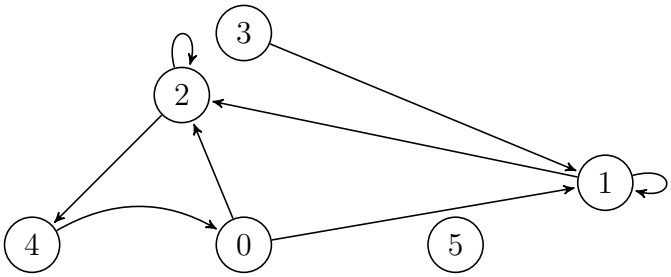


Abbildung 2.5: Ein weiterer Graph.

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: Viele Probleme heutzutage können mit einem Graphen modelliert und mit Hilfe entsprechender Graphenalgorithmen gelöst werden.
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.

Graphenalgorithmus: Ein konkretes Problem kann also mit Hilfe von einem Graphen abstrahiert werden. Ein Graphenalgorithmus kann auf einem Graphen angewendet werden, und das Resultat kann wieder auf das ursprüngliche, konkrete Problem übertragen werden. Durch diese Abstraktion kann also der gleiche Graphenalgorithmus eine Menge von verschiedenen konkreten Problemen lösen.

Traversieren: Es gibt viele verschiedene Algorithmen für Graphen, und eine wichtige Klasse von Algorithmen ist das Durchsuchen bzw. Traversieren von Graphen. Beim Traversieren von einem 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.

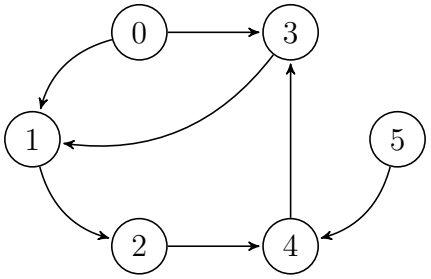


Abbildung 2.6: Ein Beispiels-Graph

Erreichbarkeit: Betrachten Sie den Graphen in Abb. 2.6. 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.

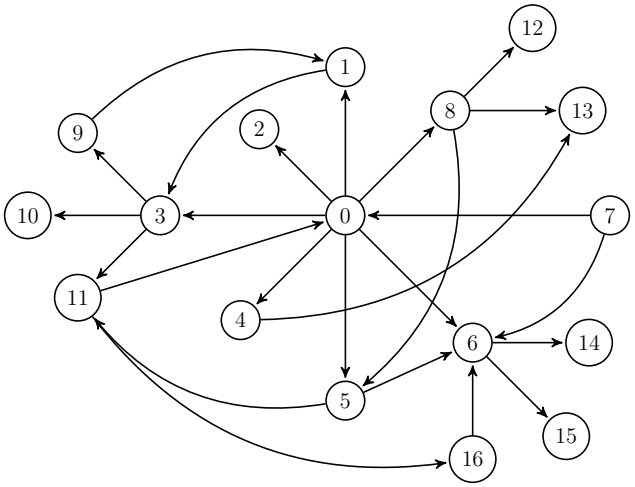


Abbildung 2.7: Ein etwas komplexerer Graph

Betrachten Sie nun den Graphen in Abb.2.7 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. Wie wir gesehen haben, können Graphen mit Hilfe von Matrizen dargestellt werden. Ein Computer kann aber nicht auf eine visuelle Beurteilung zurückgreifen und braucht zusätzlich noch Instruktionen, die ihm mitteilen, wie das Problem überhaupt gelöst werden kann.

Algorithmus entwickeln: Wir möchten also einen Algorithmus entwickeln, der für einen Startknoten in einem Graphen beurteilen kann, ob ein anderer Knoten von diesem

Startknoten erreichbar ist oder nicht. Bei der Beantwortung der Frage zu den Graphen in Abb. 2.6 und 2.7 haben Sie wahrscheinlich intuitiv begonnen, ein paar Wege auszuprobieren und je nachdem wieder zu verwerfen. 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 entwickeln überlegen wir uns noch zusätzlich, ob wir ausser der Erreichbarkeit sonst noch Informationen haben wollen. Wenn wir herausgefunden haben, dass ein Knoten von einem Startknoten aus erreichbar ist, möchten wir oftmals auch gerne wissen, wie weit entfernt er ist (in Anzahl Kanten, die traversiert werden müssen, um den Knoten zu erreichen). Falls es mehrere Wege zum Knoten gibt sind wir häufig an der kürzesten Distanz interessiert. Nicht zuletzt interessiert uns dann auch der konkrete Weg, der vom Startknoten zum Zielknoten führt.

Kürzester Weg: Im Graphen aus Abb. 2.6 ist Knoten 4 von Knoten 0 über Knoten 1 und 2 erreichbar, aber auch von Knoten 0 über Knoten 3, 1, und 2. Der erste Weg ist jedoch der Kürzere.

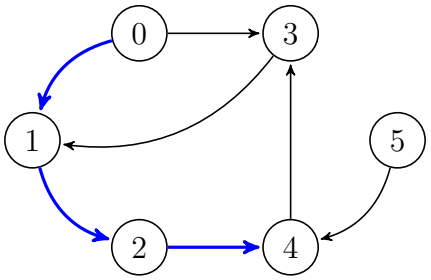


Abbildung 2.8: Weg von Knoten 0 zu Knoten 4 über Knoten 1 und 2

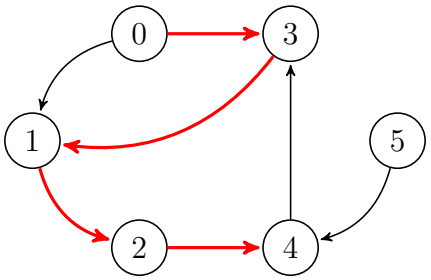


Abbildung 2.9: Weg von Knoten 0 zu Knoten 4 über Knoten 3, 1 und 2

Aufgabe 2.8. Wir möchten 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.
Beachten Sie dabei Folgendes: wenn wir beim Startknoten beginnen, möchten wir zuerst alle Knoten absuchen, welche mit minimaler Distanz vom Startknoten aus erreichbar sind. Welche Knoten sind das? 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 nähmlich, dass wir ihn auf dem kürzesten möglichen Weg gefunden haben.

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 solchen 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. Wir haben gesehen, dass vom Startknoten aus direkt all seine Nachbarn erreicht werden können. In Abb. 2.10 sehen Sie den Graphen aus Abb. **??**. Wir betrachten Knoten 0 als Startknoten (rot eingefärbt). Alle Nachbarn, die von diesem Startknoten aus erreichbar sind, sind grau eingefärbt.

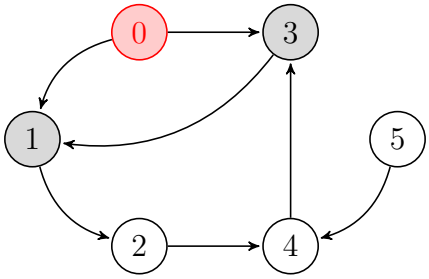


Abbildung 2.10: Startknoten und seine Nachbarn

Wir möchten nun wissen, ob Knoten 2 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 den ersten Nachbarn, Knoten 1, und betrachten dessen Nachbarn.

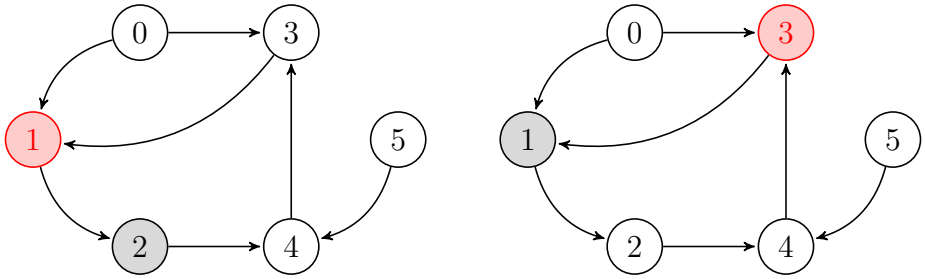


Abbildung 2.11: Aktuelle Knoten und ihre Nachbarn

In Abb. 2.11 sehen wir, dass der Knoten 2 ein Nachbarsknoten von Knoten 1 ist. Nun wählen wir den 2. Nachbarn unseres Startknotens, Knoten 3, und betrachten dessen Nachbarn. Der einzige Nachbarsknoten 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. 2.12 zeigt den Graphen, nachdem der Startknoten (Knoten 0) betrachtet wurde (links), nachdem sein erster Nachbar (Knoten 1) betrachtet wurde (rechts) und nachdem sein zweiter Nachbar (Knoten 3) betrachtet wurde.

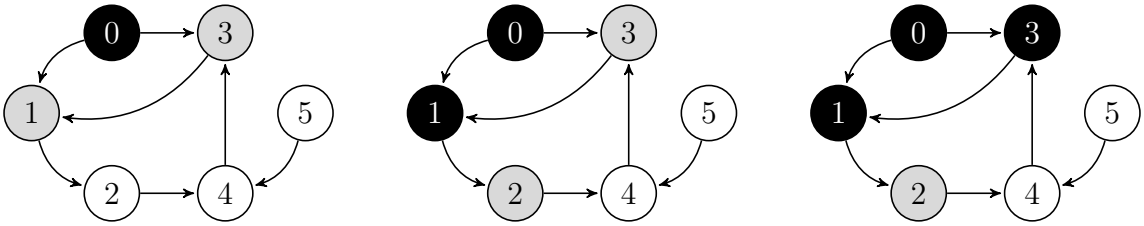


Abbildung 2.12: Verlauf der Einfärbungen

Im nächsten Durchlauf wird der nächste graue Knoten, also Knoten 2 betrachtet. Knoten 2 war der gesuchte Knoten - das heisst wir haben den Knoten gefunden!

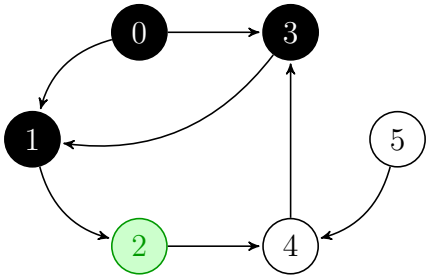


Abbildung 2.13: Zielknoten wurde gefunden

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. Am Schluss färben wir den betrachteten Knoten selbst 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 säter eingefärbt werden. 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.

Algorithmus 1 Breitensuche

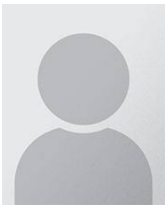
```
1:  procedure BREITENSUCHE(graph, start, ziel)
2:      start grau einfärben
3:      graueKnoten = [start]                    ▷ Warteschlange für graue Knoten
4:      while graueKnoten ≠ ∅ do
5:          current ← vorderster Knoten aus graueKnoten
6:          if current = ziel then return true          ▷ Zielknoten gefunden
7:          for all narchbar in NACHBARKNOTEN(graph, current) do
8:              if nachbar nicht schwarz oder grau then   ▷ Neu entdeckter Knoten
9:                  nachbar grau einfärben
10:                 nachbar zuhinterst in graueKnoten einreihen
11:         current schwarz einfärben                 ▷ Knoten fertig bearbeitet
12:         current aus graueKnoten entfernen
13:     return false                                  ▷ Zielknoten nicht gefunden
```

Aufgabe 2.9. 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 einem Array für jeden Knoten seine Farbe speichern. Farben können Sie einfach als Zahlen darstellen (z.B. weiss=0, grau=1, schwarz=2). Für die Umsetzung der Warteschlange der grauen Knoten können Sie auch ein Array verwenden. Mit dem Befehl $pop(x)$ kann ein Element an Stelle x im Array ausgelesen und entfernt werden.

2.2.3 Anwendung und Erweiterung

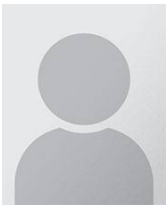
Wir werden nun den entwickelten Algorithmus anhand eines konkreten Beispiels betrachten und weiterentwickeln.

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. 2.14 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.



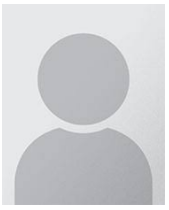
Tom

Freunde:
Karin
Tina



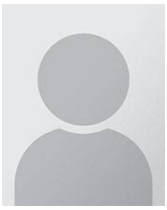
John

Freunde:
Tina
Karin



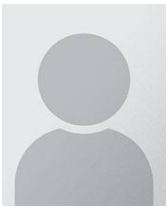
Jimmy

Freunde:
Lars
Laura



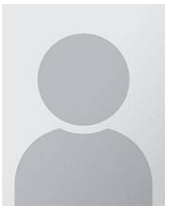
Karin

Freunde:
Tom
Tina
John



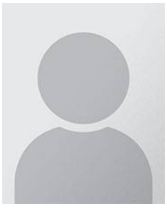
Tina

Freunde:
John
Lars
Anna
Laura
Karin
Tom



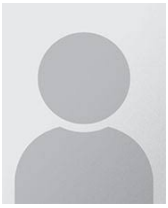
Lars

Freunde:
Jimmy
Laura
Tina



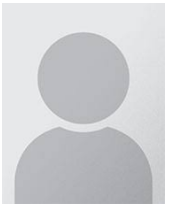
Anna

Freunde:
Tina
Simon



Simon

Freunde:
Anna
Lars



Laura

Freunde:
Tina
Lars
Jimmy

Abbildung 2.14: Ein kleines Personennetzwerk

Dieses Netzwerk kann mit einem 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. 2.15

Aufgabe 2.10. Modellieren Sie diesen Graphen als Adjazenzmatrix und lesen Sie ihn in ihr bisheriges Programm ein.

Aufgabe 2.11. Testen Sie mit ihrem Programm, ob es irgendeine Verbindung zwischen Anna und John gibt.

Aufgabe 2.12. 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 2.13. 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?

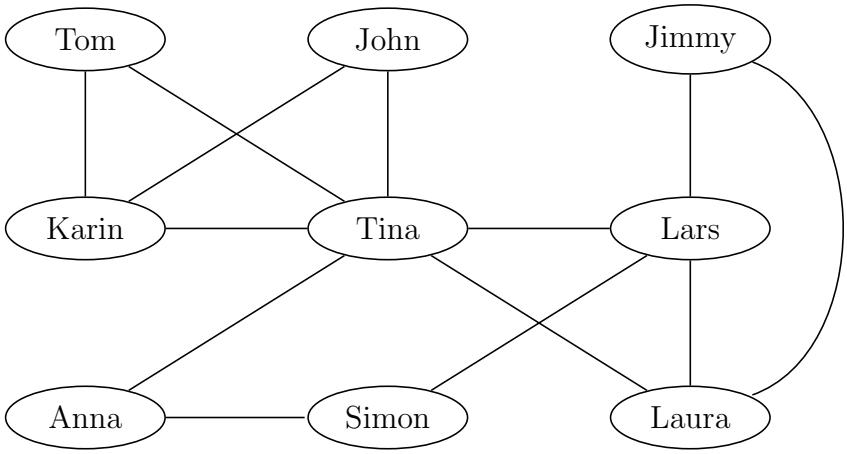


Abbildung 2.15: Modellierung des Personennetzwerks als Graph

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, 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.

Algorithmus 2 Breitensuche

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

Aufgabe 2.14. Ergänzen Sie Ihr Programm so, dass Sie am Schluss die kürzeste Distanz

von einem Startknoten zu einem Zielknoten ausgelesen werden kann, 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 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 2.5. In unserem Netzwerk aus Abb. 2.14 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. In Algorithmus 3 sehen Sie diese Erweiterung in den rot markierten Zeilen.

Algorithmus 3 Breitensuche

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

Aufgabe 2.15. Ergänzen Sie Ihr Programm nun auch so, dass am Schluss der Weg ausgelesen werden kann, der vom Startknoten zum Zielknoten führt, falls dieser vom Startknoten aus erreichbar ist. **Hinweis**: Beachten Sie, dass der Weg zuerst zurückverfolgt wird. Der Weg soll aber in der Reihenfolge vom Startknoten aus ausgegeben werden.

2.2.4 Beispiele

Betrachten Sie den Ausschnitt des ZVV-Netzes in Abb. 2.16.
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 von 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.

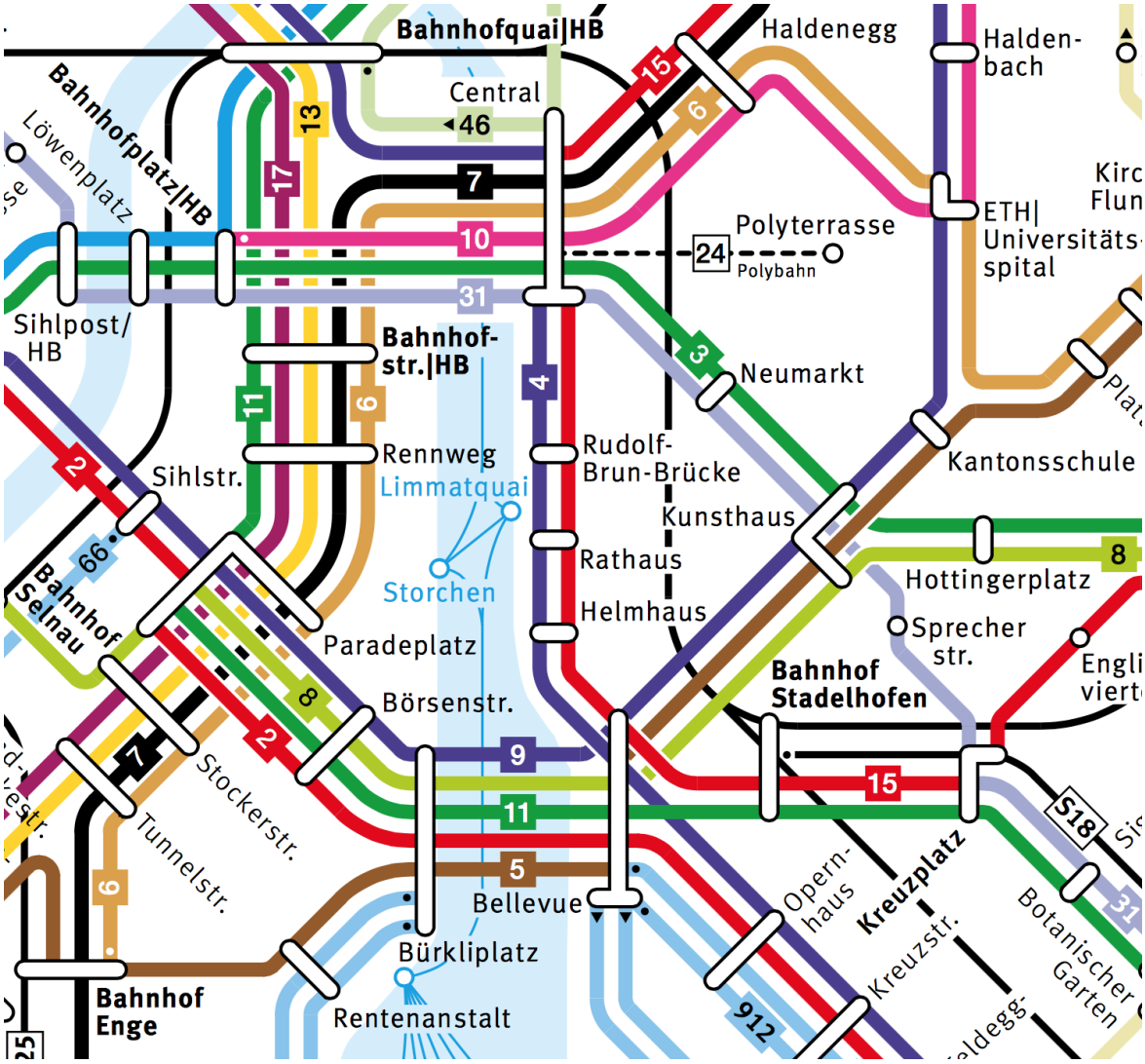


Abbildung 2.16: Ausschnitt aus dem ZVV Liniennetzplan

Aufgabe 2.16. Überlegen Sie sich, wie Sie dieses Problem mit Hilfe von einem Graphen modellieren können implementieren Sie den entsprechenden Graphen. Betrachten Sie dabei aussschliesslich Tramlinien und -haltestellen und nur solche, die im Kartenausschnitt komplett sichtbar sind.
Handelt es sich um einen gerichteten oder ungerichteteten Graphen?

Aufgabe 2.17. Stellen Sie sich vor, am Bellevue werden Sanierungsarbeiten durchgeführt und das Bellevue muss temportär für den Trambetrieb geschlossen werden. Erreichen Sie

von der ETH aus dennoch den Bahnhof Enge? Benutzen Sie Ihr Programm, um dies zu überprüfen.

Aufgabe 2.18. Wieviele Tramstationen müssen mindestens gefahren werden, um von der ETH aus zum Bahnhof Enge zu gelangen?

(a) Falls das Bellevue gesperrt ist

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

Aufgabe 2.19. Auf welchem Weg kommen Sie am schnellsten von der ETH zum Bahnhof Enge?

(a) Falls das Bellevue gesperrt ist

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

Aufgabe 2.20. Beurteilen Sie die Laufzeit des Algorithmus in Bezug auf die Anzahl Knoten und Kanten.

Aufgabe 2.21. Beurteilen Sie den Speicherbedarf des Algorithmus in Bezug auf die Anzahl Knoten und Kanten.

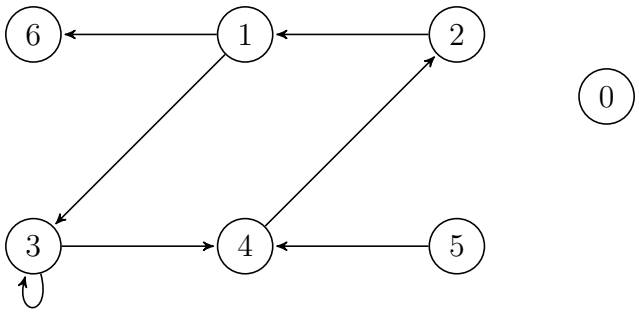
2.3 Lösungen

Lösung zu Aufgabe 2.1.

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

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

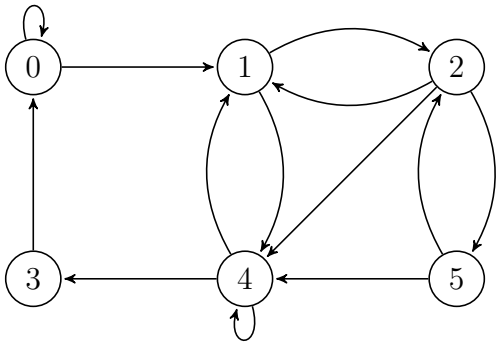
Lösung zu Aufgabe 2.2.



Lösung zu Aufgabe 2.3.

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

Lösung zu Aufgabe 2.4.



Lösung zu Aufgabe 2.5.

```
def nice_print(graph):
    n = len(graph[0])
    for i in range(n):
        for j in range(n-1):
            print graph[i][j], ";"
        print graph[i][n-1], "\n"
```

Lösung zu Aufgabe 2.6.

$$3: \{1\}; \quad 1: \{2\} \text{ und sich selber}; \quad 0: \{1,2\}$$

Lösung zu Aufgabe 2.7.

```
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 2.8. *Das Vorgehen und der Algorithmus wird in Kapitel 2.2.2* erläutert.

Lösung zu Aufgabe 2.9.

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

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

Lösung zu Aufgabe 2.10.

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

Lösung zu Aufgabe 2.11. Es gibt eine Verbindung: das Programm sollte $True$ zurückgeben.

Lösung zu Aufgabe 2.12. Es gäbe trotzdem noch eine Verbindung zwischen Anna und Jimmy, jedoch nicht zwischen Simon und Tom.

Lösung zu Aufgabe 2.13. 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 2.14.

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

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

Lösung zu Aufgabe 2.15.

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

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

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

Lösung zu Aufgabe 2.16. Die Stationen werden als Knoten abgebildet, die Linien als Kanten. Da im Kartenausschnitt alle Tramlinien in beide Richtungen befahren werden können, können wir das Problem als ungerichteten Graphen darstellen.

Lösung zu Aufgabe 2.17. Der Bahnhof Enge ist von der ETH aus auch erreichbar wenn das Bellevue gesperrt ist.

Lösung zu Aufgabe 2.18.

(a) 8

(b) 6

Lösung zu Aufgabe 2.19.

(a) ETH - Haldenegg - Central - Bahnhofstr. - Rennweg - Paradeplatz - Stockerstr. - Tunnelstr. - Bahnhof Enge

(b) ETH - Kantonsschule - Kunsthaus - Bellevue - Bürkliplatz - Rentenanstalt - Bahnhof Enge

Lösung zu Aufgabe 2.20. Bei der Breitensuche kann es sein, dass der ganze Graph traversiert werden muss, um einen Knoten zu finden bzw. um festzustellen, dass er nicht vorhanden ist. Im schlechtesten Fall müssen also alle Knoten und Kanten besucht werden, womit sich die Laufzeit der Breitensuche konzeptionell mit $\mathcal{O}(|V|+|E|)$ beschreiben lässt. In unserer Implementation müssen wir jedoch um die Nachbarn eines Knotens zu finden in der Adjazenzmatrix alle anderen Knoten auf eine Verbindung überprüfen, was zu einer Laufzeit von $\mathcal{O}(|V|^2)$ führt.

Lösung zu Aufgabe 2.21. Da wir Informationen zu allen Knoten speichern (wie zum Beispiel die Farbe, die Distanz, der Vorgängerknoten) beläuft sich der Speicherbedarf für die Breitensuche auf $\mathcal{O}(|V|)$. Zusätzlich dazu braucht das Speichern des Graphen mit einer Adjazenzmatrix $\mathcal{O}(|V|^2)$ Speicherplatz.

Literaturverzeichnis

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

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