Mathematik & Informatik II

...

Sommersemester 2016

Prof. Dr. Christoph Minnameier

Vorlesungsinhalte Mathematik & Informatik I (Jakob)

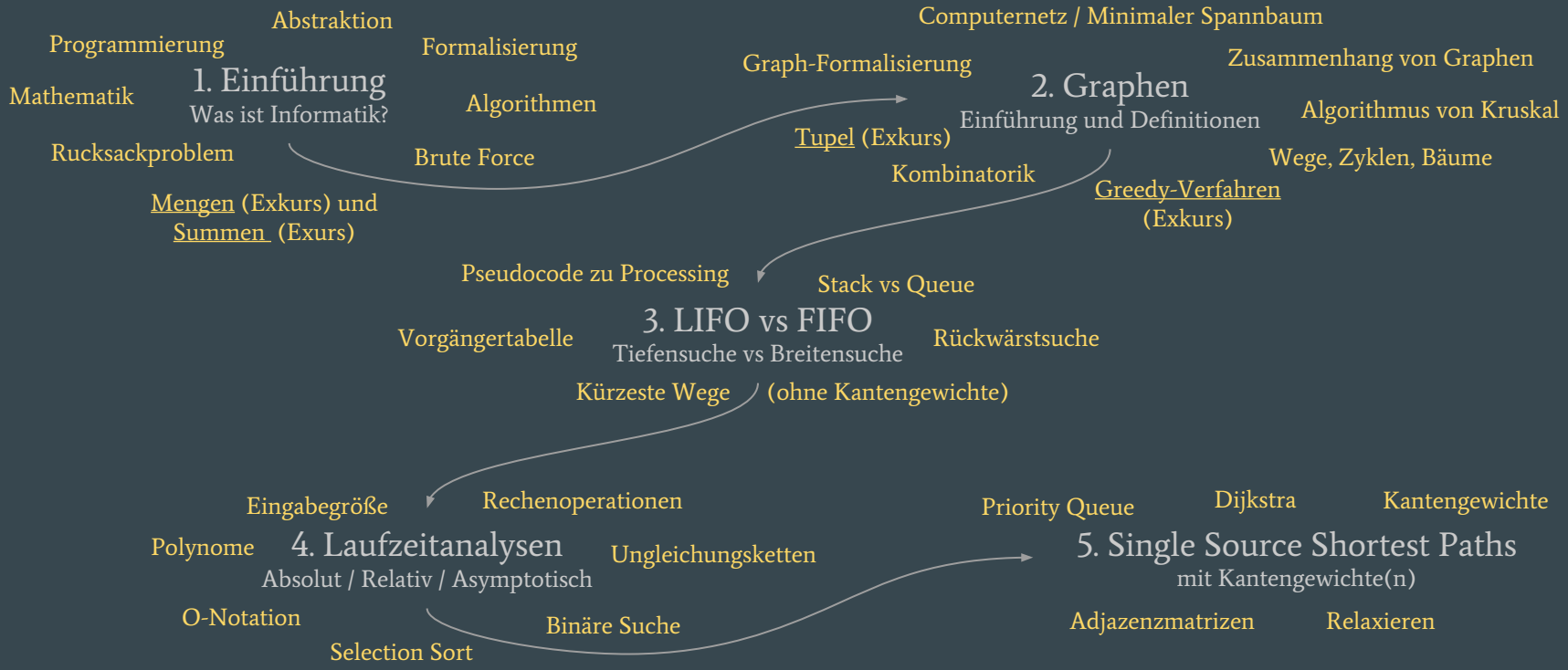
- Potenzen, Wurzeln, Logarithmen*
- Zinsrechnung (Folgen+Reihen)
- Wdh. Ableitungen
- Wdh. Grenzwerte
- Matrizen & Vektoren (Operatoren anwenden)
- Trigonometrie
- Binärdarstellung
- Logik

*kurz wiederholen

Vorlesungsinhalte Mathematik & Informatik II

- Graphentheorie
- Algorithmen & Datenstrukturen
- Kombinatorik
- Komplexität / Laufzeit

Inhaltsverzeichnis / Road Map



1. Einführung: Was ist Informatik?

Informatik in Abgrenzung zur Programmierung

- Betrachtung von spezifischen Fragestellungen bzw. Problemen
- Verallgemeinerung/Abstraktion von Problemen
- Formulierung vonPseudocode-Algorithmen zur Lösung der Probleme
- Laufzeitanalyse der Algorithmen
- Einordnung von Problemen in Komplexitätsklassen (die den Schwierigkeitsgrad der Probleme angeben) auf Basis von Beweisen.

Informatik in Abgrenzung zur Mathematik

- Anschauliche Fragestellungen.
- Oft auch umgangssprachlich formuliert.
- Algorithmen (dynamisch) statt Gleichungen (statisch).
- Trotzdem präzise Notation erforderlich, z.B.
  $G = (V, E), V \neq \emptyset, \ E \subseteq \{ (u,v) \mid u,v \in V, u \neq v \}$
- Um formale Notationen zu verstehen, ist es extrem hilfreich, vorher die umgangssprachliche Problemstellung zu durchdringen.

Beispiele für Fragestellungen bzw. Probleme

Probleme:

- Gegeben sei ein Array von Zahlen, sortiere es. (Sortieren)
- Gegeben sei ein sortiertes Array von Namen. Prüfe, ob es den Namen x enthält. (Suchen)
- Gegeben sei ein Text *t* und ein Muster *m*. Bestimme, wie oft *m* in *t* vorkommt. (Mustererkennung)
- Gegeben sei eine Landkarte (mit Städten und Straßen) und zwei Städte *s* und *t*. Finde einen kürzesten Weg von s nach t. (Wegsuche)
- Gegeben sei eine Menge von Objekten, die ein Gewicht und einen Verkaufspreis besitzen. Außerdem gegeben sei eine maximale Traglast. Gesucht ist eine Auswahl an Gegenständen, deren Gesamtgewicht die maximale Traglast nicht übersteigt und für die die Summe der Verkaufspreise (verglichen mit allen anderen erlaubten Auswahlen) maximal ist. (Rucksackproblem)

Überlegungen:

- Sind die Probleme überhaupt lösbar? Versuche jeweils eine brute-force Lösung zu formulieren.
- Gibt es einen Algorithmus, der das Problem effizienter löst?
- Welches der genannten Probleme ist am schwersten?
- Wie kann man beweisen, dass ein Problem ‘einfach’ ist? Wie, dass es ‘schwer’ ist?

Abstraktion und Formalisierung

Betrachte folgendes Problem:
Gegeben sei eine Menge von Pokémons. Außerdem gegeben sei eine maximale Erfahrungspunkt-Summe. Triff eine Auswahl von Pokémons, so dass ihre Erfahrungspunkte insgesamt nicht größer als die vorgegebene Summe sind und dabei die Summe ihrer Freundschaftswerte möglichst groß wird.

- Es fällt auf, dass das Problem dem Rucksackproblem entspricht, wenn man die *Pokémons* als *Objekte*, ihre *Erfahrungspunkte* als *Gewicht* und die *Freundschaftswerte* als *Verkaufspreis* ansieht.
- Ein Algorithmus, der das eine Problem löst, löst also auch das andere.
- Um Probleme nicht immer aufs Neue zu lösen, wollen wir die Fragestellungen *abstrahieren*. Statt von *Verkaufspreisen* und *Freundschaftswerten* sprechen wir daher einfach von einem *Nutzwert*, der *maximiert* werden soll. Auf diese Weise können wir Probleme, die bereits gelöst sind, wiedererkennen.
- Weil umgangssprachliche Fragestellungen erstens *mehrdeutig* formuliert sein können, und zweitens nicht als *Eingabe* für ein Computerprogramm geeignet sind, wollen wir die Fragestellungen formalisieren. Die Formalisierung ermöglicht uns außerdem die präzisere Formulierung von Pseudocode zur Problemlösung.
- Das Verständnis der Formalisierung wird enorm erleichtert, wenn wir die umgangssprachliche Problemstellung bereits verstanden haben (siehe nächste Folie).

Rucksackproblem: Abstraktion und Formalisierung

Gegeben sei eine Menge von Objekten, die ein Gewicht und einen Verkaufspreis besitzen. Außerdem gegeben sei eine maximale Traglast. Gesucht ist eine Auswahl, die die maximale Traglast nicht übersteigt und dabei einen möglichst großen Gesamterlös hat.

Abstraktion

Gegeben sei eine Menge von Objekten, die ein Gewicht und einen Nutzwert besitzen. Außerdem gegeben sei eine maximale Traglast. Gesucht ist eine Auswahl an Gegenständen, deren Gesamtgewicht die maximale Traglast nicht übersteigt und für die die Summe der Nutzwerte maximal ist.

Umgangssprachliche Formalisierung

Gegeben sei eine maximale Traglast T und eine Menge M von Objekten. Jedem Objekt $x \in M$ sei außerdem ein Gewicht $w(x)$ und einen Nutzwert $v(x)$ zugewiesen. Gesucht ist eine Auswahl $A \subseteq M$, deren Gesamtgewicht höchstens T ist und für die die Summe der Nutzwerte maximal ist.

Exkurs: Mengen

Mathematische Formalisierung

Gegeben sei eine endliche Menge von Objekten M, eine Gewichtsfunktion $w\colon M \to \mathbb{R}$, eine Nutzfunktion $v\colon M \to \mathbb{R}$ und eine Gewichtsschranke $T \in \mathbb{R}$.
Gesucht ist eine Teilmenge $A \subseteq M$, für die $\sum_{x \in A} w(x) \leq T$ gilt und $\sum_{x \in A} v(x)$ maximal ist.

Exkurs: Summen

Rucksackproblem - Brute Force Algorithmus

Problemstellung:
Gegeben sei eine endliche Menge von Objekten M, eine Gewichtsfunktion $w: M \to \mathbb{R}$, eine Nutzfunktion $v: M \to \mathbb{R}$ und eine Gewichtsschranke $B \in \mathbb{R}$.
Gesucht ist eine Teilmenge $A \subseteq M$, für die $\sum_{x \in A} w(x) \leq T$ gilt und $\sum_{x \in A} v(x)$ maximal ist.

Brute-Force Knapsack (Pseudocode)

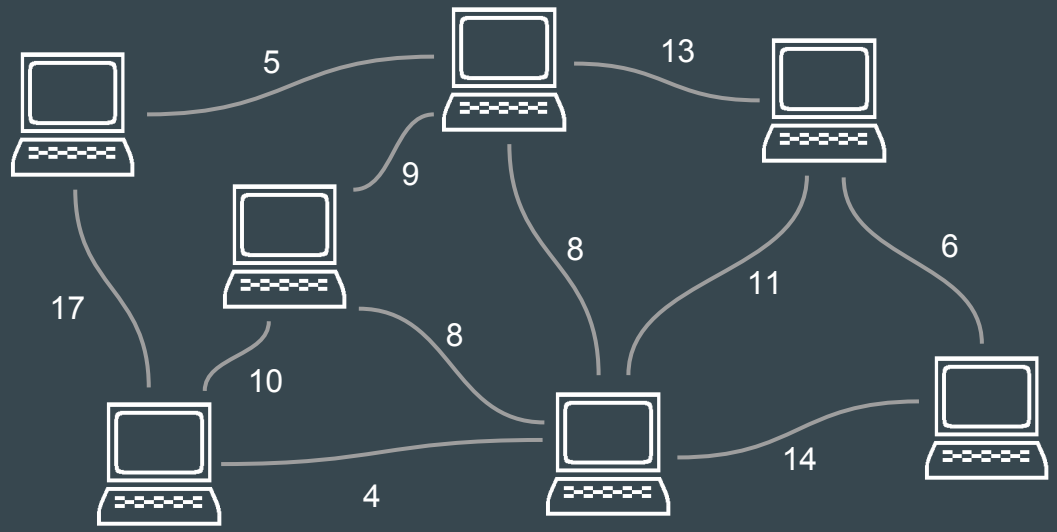
```
BestSelectionSoFar = ∅;
BestValueSoFar = 0;
foreach A ⊆ M
    if (∑x ∈ A w(x) ≤ T)
        valueOfA = ∑x ∈ A v(x);
        if (valueOfA  > BestValueSoFar)
            BestValueSoFar = valueOfA;
            BestSelectionSoFar = A;
```

- Der angegebene Algorithmus findet immer eine optimale Auswahl.
- Er iteriert dabei aber über alle $2^{|M|}$ Teilmengen von M und benötigt daher exponentiell viele Rechenschritte.
- Bereits für $|M| = 100$ ist die Berechnung mit Computern unmöglich.
- Es ist kein nicht-exponentieller Algorithmus zur Lösung des Rucksack-Problems bekannt!

2. Graphentheorie

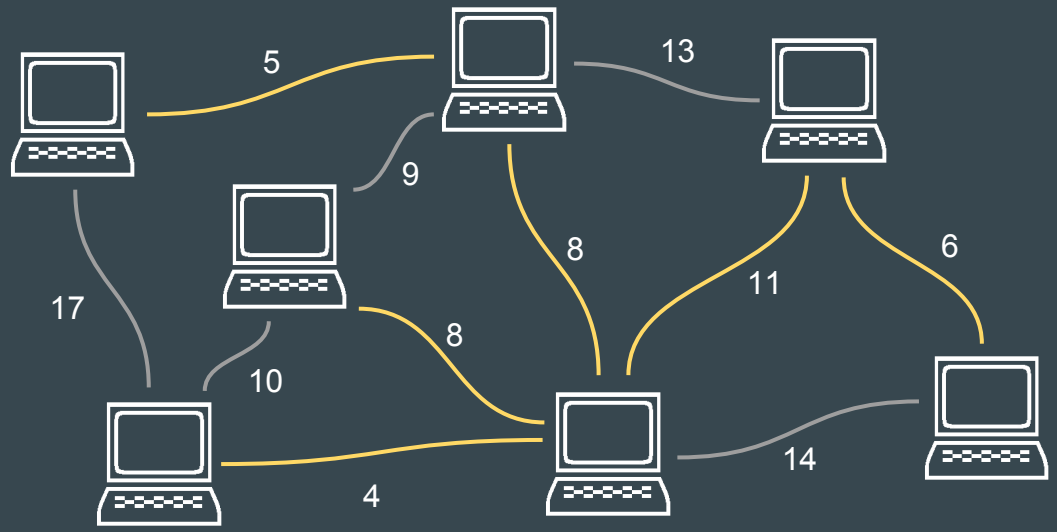
Einführungsbeispiel: Computernetzwerk

Gegeben sei eine Menge von Computern, die durch Kabel miteinander verbunden sind. Triff eine Auswahl an Kabeln, die ein zusammenhängendes Gesamtnetz gewährleisten und dabei eine möglichst geringe Gesamtlänge haben. (Die Kabellänge ist dem Schaubild zu entnehmen.)



Einführungsbeispiel: Computernetzwerk

- Die farblich markierten Kabel bilden eine optimale Auswahl.
- Wie viele (Anzahl) Kabel werden benutzt? Ist das Zufall?
- Gib einen Algorithmus an, der das Problem allgemein löst! (Tipp: Sei ‘gierig’.)



Lösung mittels Greedy-Verfahren

Greedy-Algorithmus für das Computernetz

`Auswahl = ∅;`
`Wiederhole (Anzahl Computer - 1) mal`
`  Füge das kürzeste Kabel zur Auswahl hinzu, das nicht ‘überflüssig’ ist.`

- Im Gegensatz zum offenbar sehr schwierigen Rucksack-Problem, das wir mit einem brute-force Verfahren lösen mussten, führt für dieses Problem ein greedy (‘gieriges’) Verfahren zur Lösung.
- Das liegt in der Natur des (leichteren) Problems!
- Wieder könnte man andere Probleme formulieren (z.B. Städte, die mit möglichst geringem Aufwand durch Straßen verbunden werden sollen) die inhaltlich dem von uns betrachteten Problem entsprechen.
- In Analogie zur vorangegangenen Formalisierung am Beispiel des Rucksackproblems, wollen daher wir wieder die Problemstellung (und anschließend unseren Algorithmus) formalisieren.

Computernetzwerk: Abstraktion und Formalisierung

Gegeben sei eine Menge von Computern, die durch Kabel miteinander verbunden sind. Triff eine Auswahl an Kabeln, die ein zusammenhängendes Gesamtnetz gewährleisten und dabei eine möglichst geringe Gesamtlänge haben.

Abstraktion

Gegeben sei eine Menge von Objekten und Verbindungen. Die Verbindungen besitzen jeweils ein Gewicht. Gesucht ist eine Auswahl an Verbindungen, die den Gesamt-Zusammenhang der Objekte gewährleistet und für die die Summe der Gewichte minimal ist.

Umgangssprachliche Formalisierung

Gegeben sei eine Menge V (Vertices) von Knoten und eine Menge E (Edges) von paarweisen Verbindungen. Jeder Verbindung $e \in E$ sei außerdem ein Gewicht $w(e)$ zugeordnet. Gesucht ist eine Auswahl $A \subseteq E$, die den Zusammenhang von V gewährleistet und für die die Summe der Gewichte minimal ist.

Mathematische Formalisierung

Gegeben sei eine Menge V von Knoten, eine Menge $E \subseteq \{ \{u,v\} \mid u,v \in V, u \neq v \}$ von Kanten und eine Gewichtsfunktion $w: E \to \mathbb{R}$. Gesucht ist eine Teilmenge $E' \subseteq E$ der Kanten, die den Zusammenhang von V erhält und für die $\sum_{e \in E'} w(x)$ minimal ist.

Minimum-Spanning-Tree

Gegeben sei eine Menge V von Knoten, eine Menge $E \subseteq \{ \{u,v\} \mid u,v \in V, u \neq v \}$ von Kanten, so dass V (durch E) zusammenhängt und eine Gewichtsfunktion $w: E \rightarrow \mathbb{R}$. Gesucht ist eine Teilmenge $E' \subseteq E$ der Kanten, die den Zusammenhang von V (durch E') gewährleistet und für die $\sum_{e \in E'} w(x)$ minimal ist.

Algorithmus von Kruskal

```
E' = ∅;
E = [e_1, e_2, …, e_|E|] // aufsteigend sortiert
index = 1;
Wiederhole (|V| - 1) mal
     Solange E' ∪ { e_index } einen Kreis hat
          index++;
     E' = E' ∪ { e_index };
```

- Wir dürfen nicht einfach annehmen, dass die Kanten E sortiert vorliegen.
- Stattdessen sollten wir uns bewusst machen, dass wir diese Voraussetzung durch Sortieren des Arrays selbst schaffen können. ('Preprocessing.') (z.B. in $|E|^2$ Schritten mit Selection Sort)
- Der Algorithmus benötigt offenbar genau |V|-1 Schleifendurchläufe.
- Wie prüfen wir E' auf Kreise?

Formalisierung von Graphen

- Wir haben soeben das Computernetz-Problem formalisiert. Die Formalisierung besteht einerseits aus den Anforderungen an die Lösung (minimale Gewichtsumme...) und andererseits aus der Formalisierung unseres Schaubilds (in Knoten- und Kantenmenge).
- In der Informatik existiert eine Vielzahl von Problemen, deren Fragestellung die Angabe eines solchen Schaubilds (bzw. Graphs) beinhaltet. Das Teilgebiet der Informatik, das alle solchen Probleme umfasst, nennt man Graphentheorie.
- Um Fragestellungen (und Algorithmen) in der Graphentheorie formal eindeutig angeben zu können greifen wir auf die folgende Defintion von ungerichteten Graphen zurück:

Ein ungerichter Graph $G = (V, E)$ ist ein Paar, bestehend aus einer Menge $V \neq \emptyset$, von Knoten und einer Menge E von Kanten, wobei $E \subseteq \{ \{u,v\} \mid u,v \in V, u \neq v \}$.

Exkurs: Tupel

Formalisierung von Graphen

- Ein ungerichter Graph ist definiert als Paar
  $G = (V, E)$ mit $V \neq \emptyset$ und $E \subseteq \{ \{u,v\} \mid u,v \in V, u \neq v \}$. *
- Ein ungerichter, kantengewichteter Graph ist definiert als Tripel
  $G = (V, E, w)$ mit V, E, wie oben und $w: E \rightarrow \mathbb{R}$.
- Ein gerichteter Graph ist definiert als
  $G = (V, E)$ mit $V \neq \emptyset$ und $E \subseteq \{ (u,v) \mid u,v \in V, u \neq v\}$.
- Ein gerichteter, kantengewichteter Graph ist definiert als
  $G = (V, E, w)$ mit wie oben und $w: E \rightarrow \mathbb{R}$.

*siehe auch Folie ‘Informatik in Abgrenzung zur Mathematik’

Aufgaben

1. Ist $G = (\{a,b,c\}, \{(a,b),(a,c), (b,a)\})$ gerichtet oder ungerichtet? Zeichne G.
2. Nummeriere die Computer in unserem Computernetz-Beispiel, so dass du sie als Menge V angeben kannst. Anschließend gib auf dieser Basis die Kantenmenge E an. Abschließend gib die Gewichtsfunktion w (in ‘expliziter Notation’) an.
3. Welche Bedeutung hat die Bedingung $u \neq v$ in der Definition von Graphen? Definiere einen Graphen der dieser Bedingung widerspricht und zeichne ihn.
4. Gegeben sei ein (un)gerichteter Graph $G = (V,E)$. Wie viele Kanten hat G maximal?
5. Gegeben sei eine Knotenmenge V. Wie viele (un)gerichtete Graphen gibt es auf V?

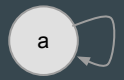
Lösungen

1. Ist G = ({a,b,c}, {(a,b),(a,c), (b,a)}) gerichtet oder ungerichtet? Zeichne G.
   G ist gerichtet (weil die Kanten Tupel sind).
   G:

2. Nummeriere die Computer in unserem Computernetz-Beispiel, so dass du sie als Menge V angeben kannst. Anschließend gib auf dieser Basis die Kantenmenge E an. Abschließend gib die Gewichtsfunktion w (in ‘expliziter Notation’) an.

   G = (V, E, w) mit
   V = {1,...,7}, E = {{1,2},{2,3},{1,4},{2,4},{2,7},{3,7},{3,5},{4,6},{4,7}, {6,7},{7,5}} und
   w({1,2}) = 5, w({2,3}) = 13, w({1,4}) = 17, w({2,4}) = 9, w({2,7}) = 8, w({3,7}) = 11, w({3,5}) = 6, w({4,6}) = 10, w({4,7}) = 8, w({6,7}) = 4, w({7,5}) =14.

Lösung

3. Welche Bedeutung hat die Bedingung $u \neq v$ in der Definition von Graphen?
   Definiere einen Graphen der dieser Bedingung widerspricht und zeichne ihn.
   Antwort: $u \neq v$ impliziert, dass ein Knoten keine Kante zu sich selbst haben darf.
   $G = (V,E)$ mit $V = \{a\}$, $E = \{(a,a)\}$ G: 

4. Gegeben sei ein (un)gerichteter Graph $G = (V,E)$. Wie viele Kanten hat G maximal?
   Gerichtet: $|E_{max}| = |V| \bullet (|V|-1)$ (vergleiche Aufgabe 3 im Exkurs Tupel)
   Ungerichtet: $|E_{max}| = |V| \bullet (|V|-1)/2$ (halb so viele wie im gerichteten Fall oder Gauß)

5. Gegeben sei eine Knotenmenge V. Wie viele (un)gerichtete Graphen gibt es auf V?
   Gerichtet: $2^{|V| \bullet (|V|-1)}$
   Ungerichtet: $2^{|V| \bullet (|V|-1)/2}$
   Begründung: Die Anzahl möglicher Kantenmengen $E \subseteq E_{max}$ also $|\mathbb{P}(E_{max})|$

Graphdefinitionen: Pfade, Zyklen, Wege

Gegeben sei ein ungerichteter Graph $G= (V,E)$

Umgangssprachlich	Formal
Ein Pfad in G ist eine Folge von Knoten, von denen jeder mit seinem Nachfolger durch eine Kante verbunden ist.	Ein Pfad von einem Starknoten v_1 zu einem Endknoten v_k in G ist ein Tupel $(v_1,...,v_k)$ von Knoten aus V, so dass für alle $i \in \{1,...,k-1\}$: $\{v_i,v_{i+1}\} \in E$.
In Graphen ohne Kantengewichte sei die Länge eines Pfads die Anzahl der Kanten auf dem Pfad. In Graphen mit Kantengewichten die Summe der Gewichte der Kanten.	In Graphen ohne Kantengewichte sei die Länge $l(p)$ eines Pfads $p = (v_1,...,v_k)$ die Anzahl der benutzen Kanten $k-1$. In Graphen mit Kantengewichten sei $l(p) = \sum_{i \in \{1,...,k-1\}} w(\{i,i+1\})$.
Ein Weg in G ist ein Pfad, der keine Kante mehrfach benutzt.	Ein Weg in G ist ein Pfad $p = (v_1,...,v_k)$, so dass für alle $i,j \in \{1,...,k-1\}$ gilt $i \neq j \Rightarrow \{v_i, v_{i+1}\} \neq \{v_j, v_{j+1}\}$.
Ein Weg in G dessen Start- und Endknoten identisch sind, heißt Zyklus.	Ein Weg $p = (v_1,...,v_k)$ in G heißt Zyklus, falls $v_1 = v_k$.

Graphdefinitionen: Grad, Zusammenhang, Bäume

Gegeben sei ein ungerichteter Graph $G = (V,E)$

Umgangssprachlich	Formal
Für zwei Knoten $u,v \in V$ sei die Distanz $dist(u,v)$ die Länge eines kürzesten Wegs von u nach v.	Für zwei Knoten $u,v \in V$ sei $dist(u,v) = \min \{ l(p) \mid p \text{ ist Weg von } u \text{ nach } v \}$
Der Grad eines Knoten $v \in V$ (schreibe: $deg(v)$) ist die Anzahl seiner Kanten.	Für einen Knoten $v \in V$ ist $deg(v) = \lvert\{\{u,v\} \mid \{u,v\} \in E\}\rvert$
G ist zusammenhängend, wenn für jedes Paar $s, t \in V$ ein Weg von s nach t existiert.	G ist zusammenhängend $\Leftrightarrow$ $\forall s,t \in V \; \exists \; p = (s, v1, ..., vk, t)$ mit p ist Weg in G.
Wenn G zusammenhängend und in G kein Zyklus existiert, bezeichnen wir G als Baum.	[aufgrund unserer vorangegangenen Definitionen ist die umgangssprachliche Definition bereits exakt]
In einem Baum G bezeichnen wir jeden Knoten mit nur einer Kante als Blatt.	In einem Baum G bezeichen wir $v \in V$ mit $deg(v) = 1$ als Blatt.

Vorteile unserer Definitionen

Nachdem wir nun exakt definiert haben, was ein ungerichter, kantengewichter Graph ist und wann ein Graph zusammenhängend ist, können wir diese Definitionen verwenden, um Probleme knapp und eindeutig zu formulieren:

Beispiel: Minimum Spanning Tree	
alt	*Gegeben sei eine Menge V von Knoten, eine Menge* $E \subseteq \{ \{u,v\} \mid u,v \in V, u \neq v \}$ *von Kanten, so dass V (durch E) zusammenhängt und eine Gewichtsfunktion* $w: E \to \mathbb{R}$. *Gesucht ist eine Teilmenge* $E' \subseteq E$ *der Kanten, die den Zusammenhang von V (durch E') gewährleistet und für die* $\sum_{e \in E'} w(x)$ *minimal ist.*
neu	Gegeben sei ein zusammenhängender, *ungerichteter, kantengewichteter* Graph $G = (V,E,w)$. *Gesucht ist eine Teilmenge* $E' \subseteq E$ *der Kanten, so dass auch* $G' = (V, E')$ *zusammenhängend ist und für die* $\sum_{e \in E'} w(e)$ *minimal ist.*

Genau genommen war die Definition vorher (obwohl sie länger war) auch nicht eindeutig, weil wir gar nicht definiert hatten, was "V hängt (durch E) zusammen" bedeuten soll.

Aufgaben

1. Warum haben wir einen Weg nicht wie folgt definiert, bzw. wo liegt der Unterschied zu unserer Definition? Gib ein entsprechendes Beispiel an.
   Ein Weg in G ist ein Pfad $(v_1,...,v_k)$, so dass für alle $i,j \in \{1,...,k-1\}$ gilt $i \neq j \Rightarrow v_i \neq v_j$.
2. Beweise: Existiert in einem ungerichteten Graphen G ein Pfad von v_1 nach v_k, so existiert auch ein Weg von v_1 nach v_k.
3. Gegeben sei ein beliebiger ungerichteter Graph $G = (V, E)$.
   Beweise:
   Die Summe der Grade aller Knoten $\sum_{v \in V} \deg(v)$ ist gerade.
4. Gegeben sei ein beliebiger ungerichteter Graph $G = (V, E)$, sowie drei Knoten $u,v,w \in V$.
   Beweise (Dreiecksungleichung):
   $\text{dist}(u,w) \leq \text{dist}(u,v) + \text{dist}(v,w)$

Lösung

1. (1,2,3,1,4) ist nach unserer Definition ein Weg, weil keine Kante doppelt benutzt wird, wäre aber nach der alternativen Defintion kein Weg, weil ein Knoten (1) doppelt besucht wird.
2. Sei $p = (v_1,..., v_k)$ ein Pfad in G, der nicht bereits ein Weg ist.
   Dann wird eine Kante e in p mehrfach verwendet, d.h. es existieren zwei Indizes $i,j \in \{1,...,k-1\}$ mit $i \neq j$ (o.b.d.A. sei $i < j$) und $\{v_i, v_{i+1}\} = \{v_j, v_{j+1}\}$. (Umkehrung der Bedingung für einen Weg).
   Wir bezeichnen $\{v_i, v_{i+1}\} = \{v_j, v_{j+1}\}$ mit e. Unterscheide zwei Fälle:
   a) $v_i = v_j$ und $v_{i+1} = v_{j+1}$ (e wird auf p zweimal in der gleichen Richtung benutzt)
   b) $v_i = v_{j+1}$ und $v_{i+1} = v_j$ (e wird auf p erst in einer, dann der anderen Richtung benutzt)
   In beiden Fällen können wir $p = (v_1,v_2,v_3,v_4, ..., v_k)$ wie folgt durch Entfernen des Teils, der zwischen den beiden Benutzungen von e liegt zu einem echt kürzeren Pfad p' modifizieren:
   a) $p' = (v_1,...,v_i,v_{j+1},...,v_k)$
   b) $p' = (v_1,...,v_i,v_{j+2},...,v_k)$
   Dieses Verfahren wiederholen wir, solange p' kein Weg ist, d.h. bis keine Kante mehrfach genutzt wird. Da sich in jeder Wiederholung die Pfadlänge echt verkürzt, terminiert das Verfahren.

Lösung

3. Betachte $G' = (V, \varnothing)$. Hier gilt offensichtlich $\sum_{v \in V} \deg(v) = 0$.
Füge die Kanten aus E nach und nach in G' ein.
Mit jeder neuen Kante steigt $\sum_{v \in V} \deg(v)$ um 2.
Folglich ist $\sum_{v \in V} \deg(v) = 2 \cdot |E|$ und damit gerade.

4. Sei $p_{(u,v)} = (u, x_1, ..., x_k, v)$ ein kürzester Weg von u nach v (d.h. $l(p_{(u,v)}) = dist(u,v)$).
Sei $p_{(v,w)} = (v, y_1, ..., y_l, w)$ ein kürzester Weg von v nach w (d.h. $l(p_{(v,w)}) = dist(v,w)$).
Dann ist offensichtlich $p_{(u,w)} = (u, x_1, ..., x_k, v, y_1, ..., y_l, w)$ ein Pfad von u nach w.
Die Anzahl benutzter Kanten in $p_{(u,w)}$ ist $l(p_{(u,w)}) = l(p_{(u,v)}) + l(p_{(v,w)}) = dist(u,v) + dist(v,w)$.
Somit ist die Existenz eines Pfads $p_{(u,w)}$ mit $l(p_{(u,w)}) = dist(u,v) + dist(v,w)$ erwiesen.
Gemäß Aufgabe 2 lässt sich aus $p_{(u,w)}$ ein Weg $p'_{(u,w)}$ mit $l(p'_{(u,w)}) \leq l(p_{(u,w)})$ erzeugen.
Somit gilt $dist(u,w) = \min \{ l(p) \mid p \text{ ist Weg von u nach v} \} \leq l(p'_{(u,w)}) \leq l(p_{(u,w)}) \leq dist(u,v) + dist(v,w)$.

Anmerkung: Diese (triviale) Aussage gilt nicht in Graphen mit negativen Kantengewichten.

Weitere Fragestellungen bzw. Graphprobleme

- Gegeben sei ein ungerichteter Graph $G = (V, E)$.
  - Ist G zusammenhängend?
- Gegeben sei ein (un)gerichteter Graph $G = (V,E)$.
  - Enthält G einen Zyklus?
- Gegeben sei ein (un)gerichteter Graph $G = (V,E)$ mit $s, t \in V$.
  - Exisitiert ein Weg von s nach t?
  - Finde einen kürzesten Weg von s nach t.
- Gegeben sei ein (un)gerichteter, kantengewichteter Graph $G = (V, E, w)$ mit $s, t \in V$.
  - Finde einen kürzesten Weg von s nach t.
- Gegeben sei ein (un)gerichteter Graph $G = (V,E)$.
  - Existiert ein Zyklus in G, der jeden Knoten genau einmal besucht?
    Hamiltonkreisproblem (Keine nicht-exponentielle Lösung bekannt!)
  - Existiert ein Zyklus in G, der jede Kante genau einmal besucht?
    Eulerkreisproblem (Sehr effiziente Lösung mit linearem Aufwand bekannt!)

Zusammenhang von Graphen

Gegeben sei ein ungerichteter Graph $G = (V, E)$. Ist G zusammenhängend?

Lösungsansatz:

1. Beginne in einem beliebigen Knoten $s \in V$ und markiere ihn als ‘erreicht’.
2. Markiere seine Nachbarn ebenfalls als erreicht und nimm sie in eine Todo-Liste auf.
3. Wiederhole dies für die Knoten in der Todo-Liste bis die Todo-Liste leer ist.
4. Nachdem der Algorithmus beendet ist, prüfe, ob alle Knoten erreicht wurden.

Frage: Welches Problem tritt hier auf?

Momentan terminiert der Algorithmus nicht.
Stelle sicher, dass jeder Knoten höchstens einmal in die Todo-Liste aufgenommen wird.

Zusammenhang von Graphen

Gegeben sei ein ungerichteter Graph G = (V, E). Ist G zusammenhängend?

Zusammenhangs-Algorithmus

```
Reached = {s}; // s ∈ V beliebig
Todo = {s};
Solange (Todo ≠ ∅)
     Entferne einen Knoten u aus Todo
     Für jeden Nachbarn n von u
          Falls n ∉ Reached
               Reached = Reached ∪ {n}
               Todo = Todo ∪ {n}
connected = (|Reached| == |V|);
```

- In Pseudocode-Algorithmen können wir mit Mengen arbeiten, sollten uns aber bewusst sein, dass andere Datenstrukturen meist besser geeignet sind.
- Wenn wir Reached nicht als Menge sondern als boolean Array der Größe |V| anlegen, können wir `n ∈ Reached` in konstanter Zeit prüfen bzw. setzen.
- Auch Todo sollten wir nicht als Menge anlegen, sondern eine der Datenstukturen Stapel (Stack) bzw. Warteschlange (Queue) verwenden. Die Wahl wird sich auch auf das Suchverhalten auswirken.

3. LIFO vs FIFO

Stapel (Stack) vs Warteschlange (Queue)

Betrachte die folgende Zeile in unserem Zusammenhangs-Algorithmus:

```
Entferne einen Knoten u aus Todo
```

Für die Korrektheit unseres Zusammenhangs-Algorithmus spielt es keine Rolle, welchen Knoten u wir in jeder Ausführung dieser Zeile aus Todo entfernen. Trotzdem wollen wir nun zwei unterschiedliche Prinzipien betrachten, nach denen wir beim Einfügen und Entfernen von Knoten in/aus Todo verfahren können:

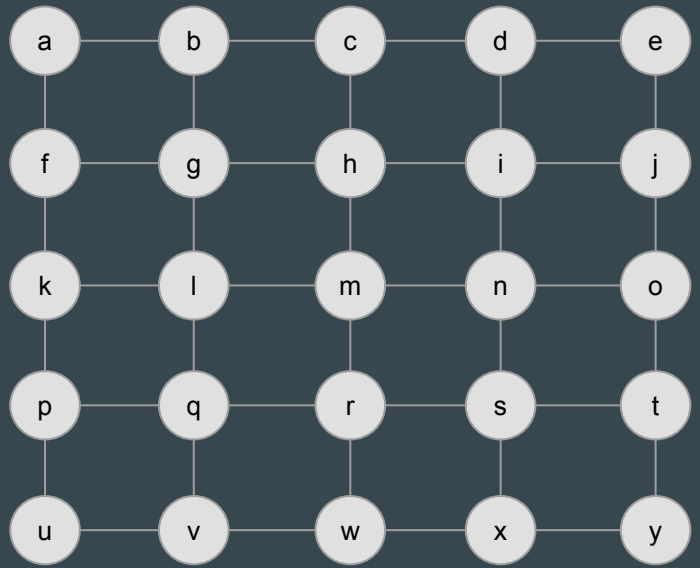
A. LIFO (last in first out) - Prinzip
   Stellen wir uns vor, wir legen neue Knoten (z.B. in Form von Zetteln) auf einen Todo-Stapel und entfernen in der obigen Codezeile den obersten (zuletzt hinzugefügten) Zettel vom Stapel.
B. FIFO (first in first out) - Prinzip
   Stellen wir uns vor wir legen die Knoten als Todo-Warteschlange auf den Tisch, an die wir neue Knoten links anlegen und in der obigen Codezeile den rechtesten (‘ältesten’) Knoten der Warteschlange entfernen.

Beachte, dass wir damit den “Grad der Beliebigkeit” (Nichtdeterminismus) in unserem Algorithmus reduzieren. Beachte aber auch, dass die Reihenfolge, in der wir die Nachbarn von u in Todo einfügen, noch immer beliebig ist!

Aufgabe

Führe den Zusammenhangs-Algorithmus am folgenden Graphen G, ausgehend vom Konten m aus.
Benutze als Todo-Liste einmal einen Stapel und einmal eine Warteschlange.
Notiere jeweils die Reihenfolge, in der die Knoten (abhängig von der gewählten Datenstruktur) als Knoten u (in "`Entferne einen Knoten u aus Todo`") betrachtet werden.

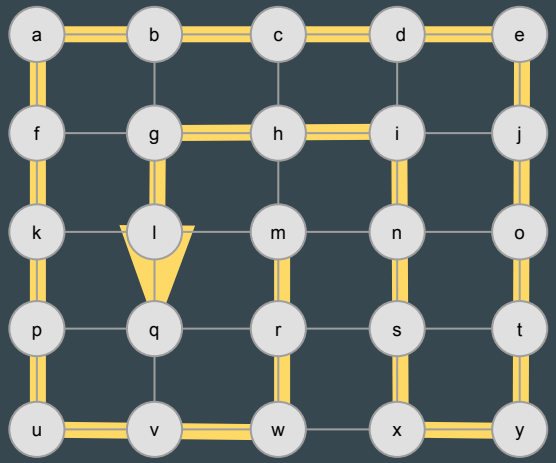
G:



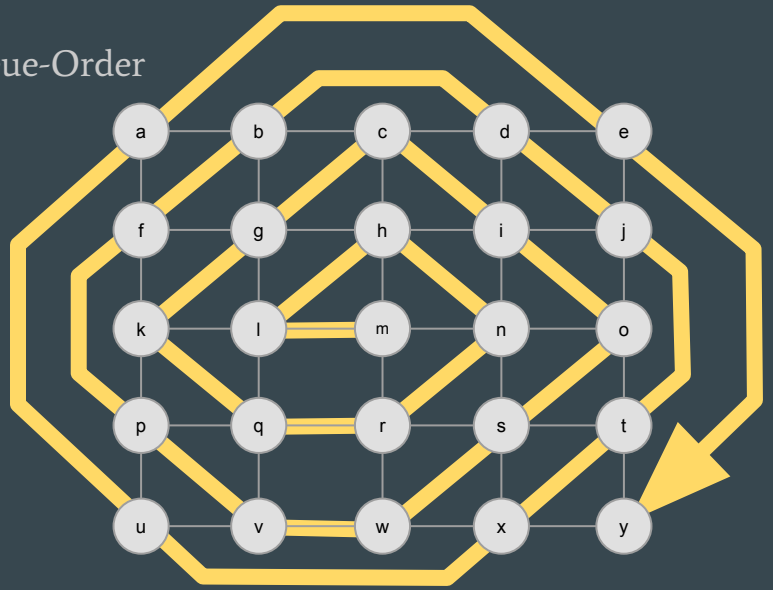
Lösung

Die folgenden Reihenfolgen sind beispielhaft. Andere Reihenfolgen sind (wegen des im Algorithmus noch immer vorhandenen Nichtdeterminisums bei der Aufnahme der Nachbarn n von u in Reached) möglich.
Stack-Order = (m, r, w, v, u, p, k, f, a, b, c, d, e, j, o, t, y, x, s, n, i, h, g, l, q)
Queue-Order = (m, l, h, n, r, q, k, g, c, i, o, s, w, v, p, f, b, d, j, t, x, u, a, e, y)

Stack-Order



Queue-Order



Tiefen- und Breitensuche

- Es ist von maßgeblicher Bedeutung, nach welchem Prinzip (LIFO/FIFO) wir unsere Todo-Liste verwalten.
    - Wählen wir als Todo-Liste einen Stapel (engl. Stack) (LIFO) und suchen somit jeweils vom zuletzt entdeckten Knoten aus weiter, so arbeiten wir uns ‘in die Tiefe’ des Graphen vor.
      Wir sprechen daher von Tiefensuche (engl. Depth-First-Search, abgek. DFS).
    - Wählen wir als Todo-Liste eine Warteschlange (engl. Queue) (FIFO) und priorisieren damit bei der weiteren Suche den jeweils zuerst entdeckten Knoten, so decken wir den Graphen ringförmig bzw. ‘in die Breite’ auf. Wir sprechen daher von Breitensuche (engl. Breadth-First-Search, abgek. BFS).
- Für das weitere Verständnis ist die Beobachtung entscheidend, dass die Tiefensuche den Graphen aufgrund der ‘chaotischen’ Reihenfolge ohne großen Mehrwert durchläuft. Sie ist damit nur für die Probleme Erreichbarkeit bzw. Zusammenhang nutzbar.
  Die Breitensuche (ausgehend von $s \in V$) hingegen durchläuft den Graphen auf sehr strukturierte Weise:
    - Für $u, v \in V$ gilt: steht u vor v in Queue-Order dann ist $dist(s, u) \leq dist(s,v)$.
      D.h. in der Breitensuche werden die Knoten gemäß ihrer Distanz zu s traversiert.
      (Dies gilt nicht für die Tiefensuche!)

Tiefen- und Breitensuche in Processing

- Download Processing-Projekt GraphTraversal
  Ziel: Einführung einer (naiven) Datenstruktur für Graphen und Veranschaulichung von LIFO- vs FIFO-Suche.
- Erläuterungen zum Code (siehe verschiedene Tabs im Processing-Projekt)
  - GraphVertex (Klasse zur Repräsentation eines Knoten)
    Beachte, dass alle Variablen/Funktionen ausschließlich für Darstellung/UI benötitgt werden.
  - GraphEdge (Klasse zur Repräsentation einer Kante)
    Die Funktion Edge(Vertex _u, Vertex _v) ist ein Konstruktor und hat daher keinen Rückgabetyp.
  - Graph (Klasse zur Repräsentation eines Graphen)
    Knoten und Kanten werden in ArrayLists (verfügbar in C#, C++, Java,...) gespeichert. Diese Datenstruktur erlaubt wie ein Array den direkten Zugriff auf ein Element mittels Index i, hat aber im Gegensatz zum Array keine fixe Größe. D.h. es können dynamisch zur Laufzeit Elemente angehängt/gelöscht werden.
    Beachte, dass diese 'naive' Weise, einen Graphen zu speichern, zu ineffizientem Laufzeitverhalten führt!
  - TodoList (Klasse zur Repräsentation unserer Todo-Liste)
    Da Processing (im Gegensatz zu C#, C++, Java,...) keine Datenstruktur Stack bzw. Queue zur Verfügung stellt, wrappen/encapsulaten wir die Klasse ArrayList, um das gewünschte Verhalten (LIFO bzw. FIFO) zu erzeugen.
  - Search (Funktionen zur Implementierung unseres Such-Algorithmus und Visualisierung der Traversierung)
    Beachte, dass die Implementierung von Search(...) gemäß unseres Zusammenhangs-Algorithmus noch aussteht.
  - Input (Funktionen zur Implementierung des User-Interface)

Aufgabe

Implementiere die im Processing-Projekt GraphTraversal fehlende Programmlogik der Funktion Search() (im Tab Search), indem du den unseren Zusammenhangs-Algorithmus (unter Berücksichitung der vorgegebenen Datenstrukturen) von Pseudcode nach Processing übersetzt.

Zusammenhangs-Algorithmus

```
Reached = {s}; // s ∈ V beliebig
Todo = {s};
Solange (Todo ≠ ∅)
     Entferne einen Knoten u aus Todo
     Für jeden Nachbarn n von u
          Falls n ∉ Reached
               Reached = Reached ∪ {n}
               Todo = Todo ∪ {n}
connected = (|Reached| == |V|);
```

- Durch Drücken der Taste “i” im Processing-Fenster kann ein vorgegebener Test-Graph erzeugt werden.
- Durch Mouse-Over (z.B. über dem Knoten in der Mitte) und Drücken der Taste “D” (bzw. “B”) wird eine Tiefensuche (bzw. Breitensuche) ausgeführt.
- Teste damit deine Implementierung.
- Die Zahlen neben den Knoten entsprechen der Reihenfolge ihrer Aufnahme in Reached. (Im Gegensatz zu unserer vorigen Betrachtung der Reihenfolge des Entfernens aus Todo.)

Lösung

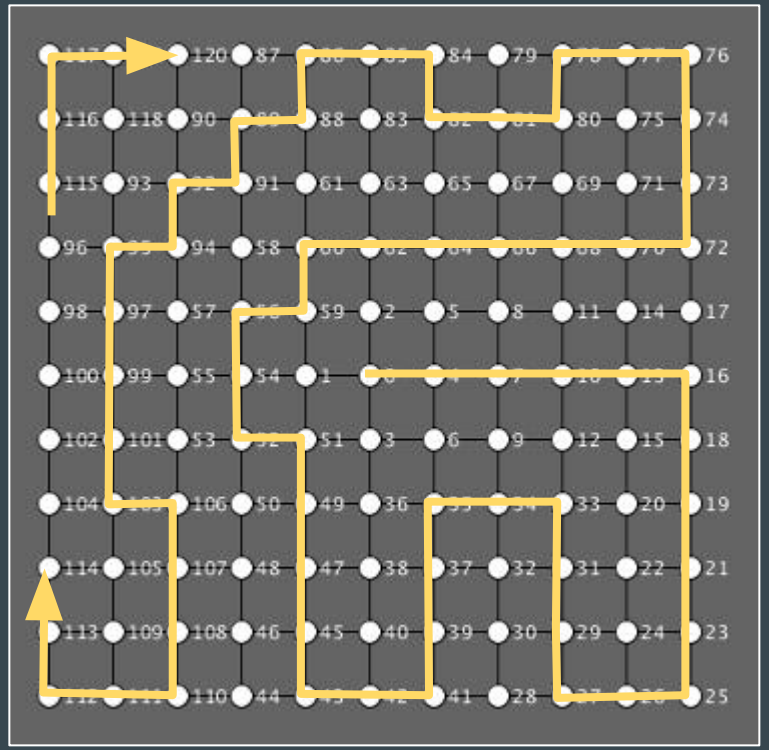
```
ArrayList<Vertex> Search(Vertex _Start, EOperationMode _Mode) {
	// we use an array list as reached set for simplicity. note that this is very inefficient.
	// (normal graph implementations use integers as vertices and a boolean array (of size |V|) as reached set)
	ArrayList<Vertex> Reached = new ArrayList<Vertex>();
	TodoList Todo = new TodoList(_Mode);

	Reached.add(_Start);
	Todo.Add(_Start);
	while (Todo.GetSize() > 0) {
		Vertex u = Todo.Remove();
		for (Edge e : Edges) // to iterate over all neighbours of u we have to iterate over all edges
			if (e.Contains(u)) { // and restrict ourselves to those that contain u (this causes bad performance!)
				Vertex n = e.GetNeighbour(u);
				if (Contains(Reached, n) == false) {
					Reached.add(n);
					Todo.Add(n);
				}
			}
	}

	return Reached;
}
```

Download GraphTraversalSolution

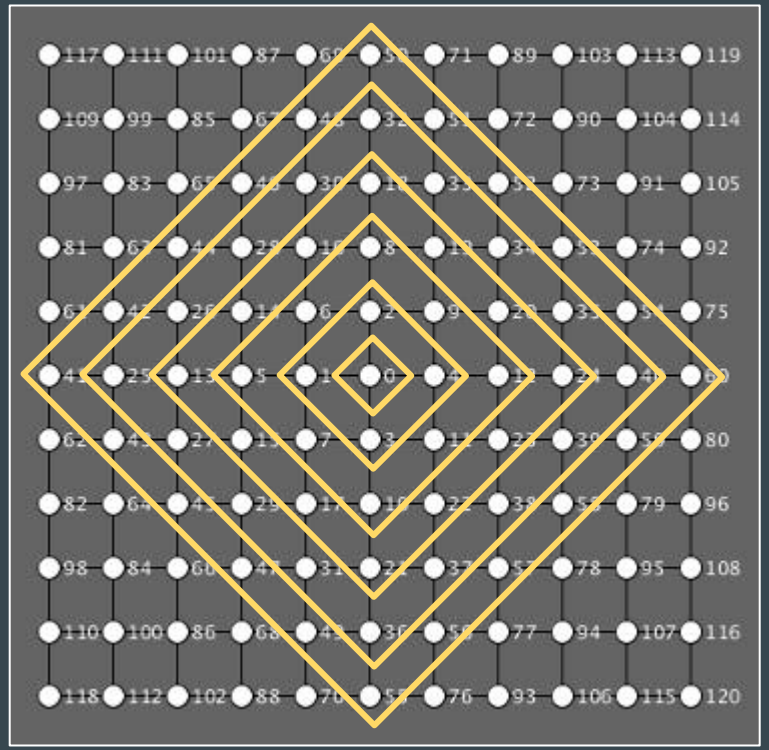
Tiefen- und Breitensuche in Processing

Visualisierung der Reihenfolge, in der wir jeden Knoten aus Todo entfernen (und als Knoten u betrachten).

DFS



BFS



Aufgabe

Die Nummerierung weicht vom gelben Pfad ab, weil wir die Knoten momentan gemäß ihrer Aufnahme in Reached nummerieren. (Im Gegensatz zu unserer vorigen Betrachtung der Reihenfolge des Entfernens aus Todo.)

1. Sorge dafür, dass die Knoten gemäß des Entfernens aus Todo nummeriert werden. Erstelle dafür in Search (...) eine weitere ArrayList ‘Visited’, füge u bei Betrachtung dort ein und gib diese Liste (statt Reached) zurück. Beachte die neue Nummerierung!
2. Verändere die Reihenfolge der Kantenmenge nach Erzeugen des Testgraphen (durch InitSampleGraph()) zufällig. Beachte die Auswirkungen auf die Traversierungsreihenfolge der Tiefensuche bzw. Breitensuche.

Lösung

1.
```
ArrayList<Vertex> Search(Vertex _Start, EOperationMode _Mode) {
      ArrayList<Vertex> Reached = new ArrayList<Vertex>();
      ArrayList<Vertex> Visited = new ArrayList<Vertex>();
      …
            Vertex u = Todo.Remove();
            Visited.add(u);
            …
      return Visited;
}
```
2.
```
void InitSampleGraph() {
      …
      ArrayList<Edge> ShuffledEdges = new ArrayList<Edge>();
      while (Edges.size() > 0) {
            int iRandomIndex = (int)random(0, Edges.size()-1);
            ShuffledEdges.add(Edges.get(iRandomIndex));
            Edges.remove(iRandomIndex);
      }
      Edges = ShuffledEdges;
}
```

Download GraphTraversalSolutionRandomized

Weitere Fragestellungen bzw. Graphprobleme

Die ersten bereits genannten Problemstellungen lassen sich mit Tiefen- oder Breitensuche lösen!

- Gegeben sei ein ungerichteter Graph $G = (V, E)$.
  - Ist G zusammenhängend?
- Gegeben sei ein (un)gerichteter Graph $G = (V,E)$.
  - Enthält G einen Zyklus?
- Gegeben sei ein (un)gerichteter Graph $G = (V,E)$ mit $s, t \in V$.
  - Exisitiert ein Weg von s nach t?

Wir wollen uns nun der nächsten, etwas komplexeren Frage zuwenden:

- Gegeben sei ein (un)gerichteter Graph $G = (V,E)$ mit $s, t \in V$.
  - Finde einen kürzesten Weg von s nach t.

Wie können wir uns die strukturierte Traversierung durch die Breitensuche zu Nutze machen?

Kürzester Weg von s nach t

Gegeben sei ein (un)gerichteter Graph $G = (V,E)$ mit $s, t \in V$. Finde einen kürzesten Weg von s nach t.

Folgende Beobachtungen (zur Breitensuche mit Start in s) führen zur Lösung:

- Wir wissen, dass $dist(s,s) = 0$ (d.h. s ist von s aus trivialerweise in 0 Schritten erreichbar)
- Für alle Nachbarn n von s gilt $dist(s,n) = 1$, und $p = (s, n)$ ist ein kürzester Weg von s nach n.
- Für alle Nachbarn n' (die nicht bereits zuvor erreicht wurden) dieser Nachbarn n gilt $dist(s, n') = 2$, und $p = (s, n, n')$ ist ein kürzester Weg von s nach n'.
- usw.

Wird also $n \in V$ als Nachbar von $u \in V$ in Reached aufgenommen, so gilt (verallgemeinert)

- $dist(s, n) = dist(s, u) + 1$, bzw.
- Ist $p = (s, v_1, ..., v_k, u)$ ein kürzester Weg von s nach u, so ist $p' = (s, v_1, ..., v_k, u, n)$ (also einfach p erweitert um n) ein kürzester Weg von s nach n.

Diesen Umstand können wir ausnutzen, um alle kürzesten Wege von s aus (zu von dort erreichbaren Knoten) zu berechnen, indem wir uns für jeden Knoten n seinen Vorgänger u speichern, von dem aus n erreicht wurde!

Kürzester Weg von s nach t (bzw. zu allen v ∈ V)

Wegsuche-Algorithmus (von s nach t)

```
// (1) Berechne Vorgängertabelle
Vorgängertabelle = [-1, …, -1]; // |V| Einträge
Vorgängertabelle[s] = s; // markiere s als besucht
Todo = (s); // Warteschlange
Solange (Todo.length > 0)
      u = Erster Knoten aus TodoQueue
      Für jeden Nachbarn n von u
            Falls Vorgängertabelle[n] == -1
                  Vorgängertabelle[n] = u
                  Hänge n an Todo an
// (2) Erstelle Pfad durch Rückwärts-Suche
Pfad = ()
Falls (Vorgängertabelle[t] != -1)
      PfadKnoten = t;
      Solange (PfadKnoten != s)
            Hänge PfadKnoten an Pfad an
            PfadKnoten = Vorgängertabelle[PfadKnoten]
      Hänge s an Pfad an
      Drehe Pfad um.
```

- Indem wir für n ∈ V auf Vorgängertabelle[n] zugreifen, nehmen wir implizit V = {0,…,|V|-1} an.
  Dies lässt sich aber leicht realisieren (siehe Processing).
- Eigentlich waren wir nur an einem kürzesten Weg von s zu einem speziellen Zielknoten t interessiert.
- Nach einmaliger Breitensuche (1) von s aus genügt aber (2) der Zugriff auf die bestehende Vorgängertabelle um für beliebige Zielknoten t' einen kürzesten Weg von s nacht t' in nur dist(s,t') Schleifendurchläufen zu berechnen!
- Da wir somit beim Berechnen eines kürzesten Wegs schon die Hauptarbeit für alle von s ausgehenden kürzesten Wege leisten, sprechen wir hier von einem single source shortest path(s) - Algorithmus.
- Statt Vorgängertabelle[] kann in (1) auch ein Array Distanz[] verwendet werden, um für alle Knoten v ∈ V die minimale Distanz dist(s,v) von s zu berechnen.

Wegsuche in Processing

- Download Processing-Projekt PathFinding
  Ziel: Wegfindung durch Berechnung der Vorgänger-Knoten und anschließende Rückwärtssuche von t zu s.
- Erläuterungen zum Code (siehe verschiedene Tabs im Processing-Projekt)
  - Search (Funktionen zur Implementierung unseres Wegsuche-Algorithmus und Visualisierung des Pfads)
    - Da unsere Knoten V als ArrayList 'Vertices' vorliegen, können wir jeden Knoten V mit seiner Position i in Vertices identifizieren, um auf Predecessor zuzugreifen. Den Index i von v in Vertices gibt GetIndex() zurück.
    - Dies ermöglicht es uns, bereits entdeckte Knoten über das Array Predecessor zu identifizieren.
    - Search() gibt nun nicht mehr die ArrayList Reached zurück, sondern ein neues Array Predecessor.
      Beachte, dass die Implementierung von Search noch aussteht.
    - Backtracking() gibt eine IntList (ArrayList vom Typ int) zurück, die einen Pfad von s nach t (in Form der Indizes der besuchten Knoten) enthält.
      Beachte, dass die Implementierung von Backtracking noch aussteht.
  - Input (Funktionen zur Implementierung des User-Interface, angepasst für die Pfadsuche)
    - Drücken der Taste 'S' triggert nun die Breitensuche ausgehend vom mouseOver-Knoten
    - Drücken der Taste 'T' triggert die Rückwärtssuche basierend auf Predecessor hin zum mouseOver-Knoten

Aufgabe

- Passe den Processing-Code in Search() so an, dass das Predecessor Array für einen Knoten n der erstmals über u ereicht wird, an der Stelle ‘Index von n in Vertices’ den Eintrag ‘Index von u in Vertices’ erhält.
- Passe den Processing-Code in Backtracking() so an, dass ein Pfad (bestehend aus den Indizes der besuchten Knoten in Vertices) berechnet wird.
- Teste deinen Code am (mit Taste “I” erzeugten) Test-Graphen.
- Entferne einige Knoten aus dem Graphen (entspricht blockierten Feldern in einem Labyrinth) und teste erneut.
- Füge einige neue Kanten zwischen weit entfernten Knoten ein (entspricht Teleport-Feldern) und teste erneut.

Lösung

```
while (Todo.GetSize() > 0) {
        Vertex u = Todo.Remove();
        for (Edge e : Edges) // to iterate over all neighbours of u we have to iterate over all edges
          if (e.Contains(u)) { // and restrict ourselves to those that contain u (this causes bad performance!)
              Vertex n = e.GetNeighbour(u);
              int iIndexOfN =  GetIndex(Vertices,  n);
              if (Predecessors[iIndexOfN] == -1) {
                int iIndexOfU =  GetIndex(Vertices,  u);
                Predecessors[iIndexOfN] = iIndexOfU;
                Todo.Add(n);
              }
          }
    }
```

```
IntList Backtracking(Vertex _Target, int[] _Predecessor) {
  IntList Path = new IntList();
  // TODO: IMPLEMENT THE ABOVE PSEUDOCODE ALGORITHM
  int iTargetIndex = GetIndex(Vertices, _Target);
  if (_Predecessor[iTargetIndex] != -1) {
      int iTravelIndex = iTargetIndex;
      while (_Predecessor[iTravelIndex] != iTravelIndex) {
         Path.append(iTravelIndex);
         iTravelIndex = _Predecessor[iTravelIndex];
      }
      Path.append(iTravelIndex); // << source
      Path.reverse();
  }
  return Path;
}
```

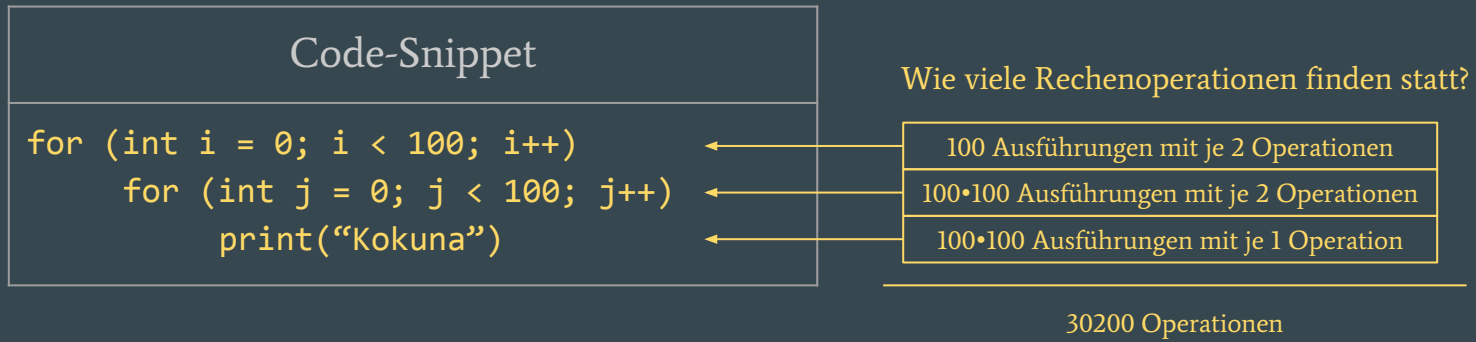
Download PathfindingSolution

4. Laufzeitanalysen

Laufzeitanalysen

Wir haben bereits bei unserem Zusammenhangs-Algorithmus angemerkt, dass die Verwendung von Mengen im Pseudocode zwar inhaltlich korrekt ist, bei der Implementierung aber Fragen zur Laufzeit aufwirft.
Da wir natürlich unter allen Algorithmen die ein Problem allgemeingültig lösen, an jenen mit der besten Laufzeit interessiert sind (sonst könnten wir für alles einen Brute Force Ansatz wählen) wollen wir uns nun der Laufzeitanalyse widmen.

Hierfür betrachten wir eingangs das folgende Code Snippet.
Wie viele Rechenschritte werden insgesamt ausgeführt?



Aufgaben

1. Bestimme die Anzahl der ausgeführten Operationen im folgenden Code Snippet:

Code-Snippet 1

```
for (int i = 1; i <= 10; i++)
      for (int j = 1; j <= i; j++)
            print(“Hornliu”)
```

2. Bestimme die Anzahl der ausgeführten Operationen im folgenden Code Snippet:

Code-Snippet 2

```
int i = 100000;
while (i > 1)
      i = i/2;
```

Lösung

1. Bestimme die Anzahl der ausgeführten Operationen im folgenden Code Snippet:

Code-Snippet 1

```
for (int i = 1; i <= 10; i++)
     for (int j = 1; j <= i; j++)
          print(“Hornliu”)
```

- 10 Ausführungen mit je 2 Operationen
- (1+…+10) Ausführungen mit je 2 Operationen
- (1+…+10) Ausführungen mit je 1 Operation

$20 + 3 \cdot (10 \cdot 11)/2 = 185$ Operationen

2. Bestimme die Anzahl der ausgeführten Operationen im folgenden Code Snippet:

Code-Snippet 2

```
int i = 100000;
while (i > 1)
     i = i/2;
```

- 1 Ausführungen mit 1 Operation
- Diese beiden Zeilen werden so oft ausgeführt, wie man 100.000 halbieren muss, um bei 2 zu landen, also $\log_2 100000 = 17$ mal.

$1 + 2 \cdot 17 + 1 = 36$ Operationen

Absolute vs Relative Laufzeit

Wir haben bisher die absolute Laufzeit einiger Code-Snippets bestimmt. Unsere Algorithmen (wie z.B. Breitensuche) haben jedoch den Anspruch für beliebige Probleminstanzen (und damit für beliebige Eingabegrößen) gültige Lösungen zu ermitteln. Wir sind daher daran interessiert die Laufzeit in Abhängigkeit der Größe der zu lösenden Probleminstanz (also der Größe des zu sortierenden Arrays, oder der Größe des Graphen dessen Zusammenhang überprüft werden soll, etc.) anzugeben.

Betrachte die folgende Funktion SumOfProducts(), die ein Array der Größe n als Parameter erhält.
Wie viele Rechenschritte werden insgesamt ausgeführt?

SumOfProducts

```
int SumOfProducts(int[] _Array) {
      int iSum = 0;
      for (int i = 0; i < _Array.length; i++)
            for (int j = 0; j < _Array.length; j++)
                  iSum += _Array[i]*_Array[j];
      return iSum;
}
```

Wie viele Rechenoperationen finden statt?

- 1 Ausführungen mit 1 Operation
- n Ausführungen mit je 2 Operationen
- n^2 Ausführungen mit je 2 Operationen
- n^2 Ausführungen mit je 2 Operationen
- 1 Ausführungen mit 1 Operation

$4n^2 + 2n + 2$ Operationen

Aufgaben

Der folgende Algorithmus sortiert ein Array aufsteigend, indem er jeweils den kleinsten Wert nach vorne zieht. Bestimme die relative Laufzeit (abhängig von der Eingabegröße _Array.length = n) im Best/Worst Case.

Selection Sort

```
void SelectionSort(int[] _Array) {
      for (int i = 0; i < _Array.length; i++) {
            int iMinIndex = i;
            for (int j = i; j < _Array.length; j++)
                  if (_Array[j] < _Array[iMinIndex])
                        iMinIndex = j;
            int iTemp = _Array[i];
            _Array[i] = _Array[iMinIndex];
            _Array[iMinIndex] = iTemp;
      }
}
```

Lösung

Der folgende Algorithmus sortiert ein Array aufsteigend, indem er jeweils den kleinsten Wert nach vorne zieht. Bestimme die relative Laufzeit (abhängig von der Eingabegröße _Array.length = n) im Best/Worst Case.

Selection Sort

```
void SelectionSort(int[] _Array) {
      for (int i = 0; i < _Array.length; i++) {
            int iMinIndex = i;
            for (int j = i; j < _Array.length; j++)
                  if (_Array[j] < _Array[iMinIndex])
                        iMinIndex = j;
            int iTemp = _Array[i];
            _Array[i] = _Array[iMinIndex];
            _Array[iMinIndex] = iTemp;
      }
}
```

- n Ausführungen mit 2 Operationen
- n Ausführungen mit 1 Operation
- n•(n+1)/2 Ausführungen mit 2 Operation
- n•(n+1)/2 Ausführungen mit 1 Operation
- n•(n+1)/2 (bzw. 0*) Ausführungen mit 1 Operation
- n Ausführungen mit 1 Operation
- n Ausführungen mit 1 Operation
- n Ausführungen mit 1 Operation

$2n^2 + 8n$ Operationen (im worst case)
bzw.
$1{,}5n^2 + 7{,}5n$ Operationen (im best case*)

Aufgaben

Der folgende Algorithmus sucht eine Zahl in einem sortierten Array und gibt (sofern er sie findet) ihren Index zurück. Dafür halbiert der Algorithmus in jedem Durchlauf der while-Schleife den ‘Suchabschnitt’ im Array.
Bestimme die relative Laufzeit (abhängig von der Eingabegröße _Array.length = n) im Best Case bzw. im Worst Case.

Binary Search

```
int BinarySearch(int[] _SortedArray, int _iNumber) {
	int iLeftSearchBound = 0;
	int iRightSearchBound = _SortedArray.length-1;
	while (iLeftSearchBound <= iRightSearchBound ) {
		int iMiddleIndex = (iLeftSearchBound + iRightSearchBound) / 2;
		if (_SortedArray[iMiddleIndex] == _iNumber)
			return iMiddleIndex; // i.e. we found _iNumber at iMiddleIndex
		if (_SortedArray[iMiddleIndex] > _iNumber)
			iRightSearchBound = iMiddleIndex - 1;
		else
			iLeftSearchBound = iMiddleIndex + 1;
	}
	return -1; // i.e. ‘could not find number in array’
}
```

Lösung

- Best Case:
  Die gesuchte Zahl steht beim ersten `iMiddleIndex` und wird sofort gefunden.
  Es ergeben sich ~10 Rechen-Operationen.
- Worst Case:
  Die gesuchte Zahl ist nicht im Array enthalten.
  Dies wird erst festgestellt, nachdem der zu durchsuchende Restbereich die Größe 1 hat (woraufhin die BoundsIndizes aneinander ‘vorbei laufen’ und die while-Schleife abgebrochen wird).
  Wie oft muss _Array.length = n halbiert werden, damit dies geschieht? $\Rightarrow \log_2(n)$ mal.
  (Für Arraylängen n, die keine Zweierpotenzen sind, runden wir $\log_2(n)$ einfach auf.)
  Für die while-Schleife werden 8 Operationen ausgeführt, außerahlb der Schleife 4 Operationen, es ergeben sich also insgesamt: $8 \log_2(n) + 4$ Rechen-Operationen.

Aufgabe

1. Teste Selection Sort in Processing.
   Füge dann eine globale Variable `int iCounter` in deinen Code ein.
   Füge nach jeder Programmzeile einen Aufruf iCounter++ in deinen Code ein.
   (Auf diese Weise zählen wir die Anzahl der tatsächlich ausgeführten Programmzeilen mit.)
   Gib am Ende iCounter aus und gleiche die Ausgabe mit deinen Erwartungen ab.
2. Teste BinarySearch in Processing und verfahre wie in Aufgabe 1.
3. Implementiere (zum Vergleich zu 2) einen Algorithmus LinearSearch, der das Eingabe-Array linear durchsucht und vergleiche die benötigten Rechenschritte für sehr große Arrays.
4. Implementiere eine Funktion `void Test(int n)`, die genau $2n^2 + n + 5$ mal `print('Glurak')` aufruft, ohne Multiplikation oder Addition zu verwenden!

Exakte vs. Asymptotische relative Laufzeit

Wir haben bisher die exakte relative Laufzeit (z.B. $2n^2 + 8n$ im Falle von Selection Sort) bestimmt.
Nun sind allerdings für kleine oder mittlere Eingabelängen n sowieso keine Performane-Probleme zu erwarten.
In einem Polynom $T(n) = c_k \cdot n^k + c_{k-1} \cdot n^{k-1} + ... + c_2 \cdot n^2 + c_1 \cdot n + c_0$ (mit Grad k) verlieren jedoch für sehr große Eingabelängen n alle Summanden mit nicht-maximalem Exponenten zunehmend Einfluss auf den Gesamtwert.

Weiter sind für zwei Algorithmen A und A' mit Laufzeit $T(n) = a \cdot n^x$ und $T'(n) = b \cdot n^y$ (sofern $x \neq y$) für sehr große n nur die Exponenten x und y relevant und die Faktoren a und b vernachlässigbar.

Beispiel: Sei $T(n) = 1/2 \cdot n^5$ und $T'(n) = 30 \cdot n^4$.
Dann gilt zwar für kleine n, bspw. $n = 6$ noch $T(n) = 3888$ und $T'(n) = 38880$ und damit $T(n) < T'(n)$.
Für hinreichend große n, bspw. schon für $n = 1000$ gilt aber: $T(n) = 0.5 \cdot 10^{15}$ und $T'(n) = 30 \cdot 10^{12}$, also $T(n) > T'(n)$.

Deshalb begnügen wir uns meist damit die asymptotische Laufzeit von Algorithmen anzugeben. D.h. wir verzichten auf die Angabe eines exakten Terms T(n) und sprechen lediglich von Algorithmen mit logarithmischer, linearer, quadratischer, kubischer... Laufzeit.

Diese umgangssprachliche Einordnung basiert auf der im Folgenden erklärten mathematischen Formalisierung.

Aufgabe

Gib die Anzahl der print-Aufrufe (abhängig von n) für folgende Codefragemente an.

a)
```
for (int i = 0; i < n; i++)
  for (int j = 0; j < 30; j++)
    print (‘a’);
```

b)
```
for (int i = 0; i <= n; i++)
  for (int j = 0; j <= n; j++)
    for (int k = 0; k <= n; k++)
      print (‘b’);
```

c)
```
for (int i = 0; i < 50; i++)
  for (int j = 0; j < 50; j++)
    print (‘c’);
```

d)
```
for (int i = 0; i < n; i++)
  for (int j = i+1; j < n; j++)
    print (‘d’);
```

Vergleiche anschließend alle diese Laufzeiten für die Eingagebrößen n = 10 bzw. n = 5000.

Lösung

a) $T(n) = 30n$
b) $T(n) = (n+1)^3$
c) $T(n) = 2500$
d) $T(n) = n(n-1)/2$

O-Notation

Die O-Notation (als eine der Landau-Notationen) zielt darauf ab, Algorithmen (bzw. deren Laufzeit, die, wie wir gesehen haben, als Funktion T(n) ermittelt werden kann) in Klassen (bzw. Ordnungen) einzuordnen.
Sie wurde also mit dem Ziel formuliert, zwei Laufzeitangaben (und damit Funktionen) die sich 'geringfügig' unterscheiden (z.B. Polynome des gleichen Grads) der selben Klasse zuzuweisen, zwei Laufzeitangaben, die sich 'maßgeblich' unterscheiden (z.B. Polynome verschiedenen Grads) aber verschiedenen Klassen.

Für eine Funktion f(n) ist ihre Ordnung O(f(n)) definiert als:
$O(f(n)) = \{ g(n) \mid \exists\ n_0 \in \mathbb{N}, c \in \mathbb{R}$, so dass $\forall\, n \geq n_0$ gilt $g(n) \leq c \cdot f(n) \}$

Für eine Funktion f(n) ist also O(f(n)) eine Menge von Funktionen. Eine andere Funktion g(n) gehört zu dieser Menge gdw. es einen konstanten Faktor c gibt, und eine Zahl n_0, so dass für alle Werte 'ab n_0' der Wert von g(n) kleiner gleich c mal f(n) ist.

Wir betrachten zunächst ein paar Beispiele, um diese formale Definition zu verstehen.

O-Notation Beispiele

1. Sei $f(n) = 2n^2$ und $g(n) = n$.
   Hier gilt offenbar grundsätzlich: $g(n) \leq f(n)$, oder formaler: $\forall n \in N\ g(n) = n \leq 2n^2 = f(n)$
   Wir 'benötigen' hier weder n_0 noch c, es gilt aber insbesondere mit $n_0 = 0$ und $c = 1$ $\forall n \geq n_0$ ist $g(n) \leq c \cdot f(n)$. Damit gilt also $g(n) \in O(f(n))$.
2. Sei $f(n) = 2n^2$ und $g(n) = 5n$.
   Dann gilt nicht grundsätzlich: $g(n) \leq f(n)$. (z.B. für $n = 2$: $g(n) = 10 \not\leq 8 = f(n)$.)
   Allerdings gilt bereits für alle $n \geq 3$: $g(n) \leq f(n)$.
   Wir benötigen diesmal also n_0 aber nicht c und stellen fest: mit $n_0 = 3$ und $c = 1$ gilt $\forall n \geq n_0$ ist $g(n) \leq c \cdot f(n)$.
   Damit gilt also $g(n) \in O(f(n))$.
3. Sei $f(n) = n^2$ und $g(n) = 2n^2 + 3n + 1$.
   Dann gilt nicht grundsätzlich: $g(n) \leq f(n)$. Die Ungleichung gilt sogar niemals (d.h. für kein $n \in N$).
   Wenn wir aber z.B. $c = 3$ (oder größer) und $n_0 = 4$ wählen, gilt $\forall n \geq n_0$ ist $g(n) \leq c \cdot f(n)$.
   Damit gilt also $g(n) \in O(f(n))$, d.h. selbst die 'größere' Funktion $g(n)$ ist in $O(f(n))$.

   Tatsächlich sind alle Polynome $g(n)$ mit Grad 2 oder kleiner in $O(f(n)) = O(n^2)$.

O-Notation

Wie wir eben (in Beispiel 3) gesehen haben, lässt die O-Notation offenbar alle quadratischen Polynome (darunter $2n^2 + 3n + 1$) in die gleiche Komplexitätsklasse, nämlich $O(n^2)$ fallen.
Polynome höheren Grades (z.B. $g(n) = \frac{1}{2} \cdot n^3$) fallen jedoch nicht in diese Klasse, wie wir im Folgenden sehen:

Sei $f(n) = n^2$ und $g(n) = \frac{1}{2} \cdot n^3$. Überlege, ob $g(n) \in O(n^2)$ gilt.

Existieren ein $n_0 \in \mathbb{N}$ und ein $c \in \mathbb{R}$, so dass $\forall\ n \geq n_0$ gilt: $\frac{1}{2} \cdot n^3 \leq c \cdot n^2$?

Das dies nicht der Fall ist, sieht man schnell, wenn man beide Seiten der Ungleichung durch n^2 dividiert:

In der Ungleichung $\frac{1}{2} \cdot n \leq c$ spielt es keine Rolle, wie (groß auch immer) wir c und n_0 wählen, es wird immer ein $n > n_0$ geben, für das die Ungleichung nicht gilt.
Ein solches n lässt sich sogar allgemeingültig angeben, nämlich $n = \max(2c + 1, n_0)$.
(Für $c = 10$ und $n_0 = 3$ ergibt sich also als Gegenbeispiel $n = \max(21,4) = 21$ in Form von $\frac{1}{2} \cdot 21 \not\leq 10$.)

Es gilt also $n^3 \notin O(n^2)$.

O-Notation: Beweis durch Ungleichungskette

Sei $g(n) = 2n^2 + n + 5$ und $f(n) = n^2$.
Wir wollen beweisen: $g(n) \in O(f(n))$, d.h. $g(n) \in O(n^2)$.

Beweis:

$$
\begin{aligned}
g(n) &= 2n^2 + n + 5 \\
&\quad \wedge \quad \wedge \qquad \text{für } n_0 \geq 3 \\
&\leq 2n^2 + n^2 + n^2 \\
&\leq 4 \cdot n^2 \\
&\leq c \cdot f(n) \qquad \text{für } c = 4
\end{aligned}
$$

Die Ungleichungskette beweist die Behauptung mit $n_0 = 3$ und $c = 4$.

Wir sagen also die Laufzeit $2n^2 + n + 5$ eines Algorithmus (bzw. das dadurch gelöste Problem) ist in $O(n^2)$.
Wir nehmen dabei eine Abschätzung nach oben vor, die beweist, dass das Problem in $O(n^2)$ lösbar ist.
(Beachte aber, dass wir dadurch nicht ausschließen, dass eine performantere Lösung möglich ist.)

Übung

1. Beweise: $2n+8 \in O(n)$
2. Beweise: $5n^3+3n^2+8n+3 \in O(n^4)$
3. Beweise: $n^2 \notin O(n)$
4. Gib die Laufzeiten der Algorithmen aus der letzten Übung
   a. $T(n) = 30n$
   b. $T(n) = (n+1)^3$
   c. $T(n) = 2500$
   d. $T(n) = n(n-1)/2$

   in O-Notation an.

Lösung

1. $g(n) = 2n + 8$
   $\leq 2n + 8n$ (für alle $n \in \mathbb{N}$)
   $\leq 10n$
   $\leq c \cdot n$ (für $c = 10$)
   $\Rightarrow$ Für $n_0 = 1$ und $c = 10$ folgt $\forall\ n \geq n_0$ gilt $g(n) \leq c \cdot f(n)$, demnach ist $g(n) \in O(f(n))$
2. $g(n) = 5n^3 + 3n^2 + 8n + 3$
   $\leq 5n^4 + 3n^4 + 8n^4 + 3n^4$ (für alle $n \in \mathbb{N}$)
   $\leq 19n^4$
   $\leq c \cdot n^4$ (für $c = 19$)
   $\Rightarrow$ Für $n_0 = 1$ und $c = 19$ folgt $\forall\ n \geq n_0$ gilt $g(n) \leq c \cdot f(n)$, demnach ist $g(n) \in O(f(n))$
3. Existieren ein $n_0 \in N$ und ein $c \in R$, so dass $\forall\ n \geq n_0$ gilt: $n^2 \leq c \cdot n$?
   Nein, denn für beliebig gewähltes $n_0 \in N$ und $c \in R$, widerlegt $n = \max(c+1, n_0)$ die Ungleichung.
4. a) $T(n) = 30n \in O(n)$
   b) $T(n) = (n+1)^3 = n^3 + 3n^2 + 3n + 1 \in O(n)$
   c) $T(n) = 2500 \in O(1)$
   d) $T(n) = n(n-1)/2 = \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$

Übersicht: Probleme / Algorithmen / Laufzeiten

Problem	Algorithmus	Eingabegröße(n)	Laufzeit (worst case)
Sortieren	Selection Sort / Bubble Sort / Quicksort	Arraylänge n	$O(n^2)$
Sortieren	Merge Sort / Heapsort	Arraylänge n	$O(n \cdot \log(n))$
Suchen im sortierten Array	Binäre Suche	Arraylänge n	$O(\log(n))$
Rucksackproblem	Brute-Force-Ansatz	Objektmenge M	$O(2^{\|M\|+1})$
Minimaler Spannbaum	Algorithmus von Kruskal	Graph G = (V, E)	$O(\|E\| \cdot \log(\|E\|))$ (Sortieren von E)
Graph-Zusammenhang	Tiefensuche / Breitensuche	Graph G = (V, E)	$O(\|V\|^2)$ (oder ggf. weniger)
Wegsuche (ohne Kantengew.)	Breitensuche + Backtracking	Graph G = (V, E)	vgl. Breitensuche
Wegsuche (mit Kantengew.)	Algorithmus von Dijkstra (+Backtracking)	Graph G = (V, E)	$O(\|V\| \cdot \log(\|V\|) + \|E\|)$*
Euler-Kreis Existenz	Tiefensuche / Breitensuche + Knotengrad-Prüfung	Graph G = (V, E)	vgl. Tiefensuche/Breitensuche
Hamilton-Kreis Existenz	Brute-Force-Ansatz	Graph G = (V, E)	$O(2^{(\|V\|+\|E\|)})$

*mit Heap Priority Queue

Laufzeitanalyse Zusammenhangs-Algorithmus

Wir wollen nun die (worst case) Laufzeit unseres Zusammenhangs-Algorithmus ermitteln.
Da wir an der asymptotischen Laufzeit interessiert sind, vernachlässigen wir Operationen mit konstantem Aufwand (wie z.B. `Vertex n = e.GetNeighbour(u)`).

Zusammenhangs-Algorithmus

```
while (Todo.GetSize() > 0) {
      Vertex u = Todo.Remove();
      for (Edge e : Edges)
            if (e.Contains(u)) {
                  Vertex n = e.GetNeighbour(u);
                  int iIndexOfN =  GetIndex(Vertices,  n);
                  if (Predecessors[iIndexOfN] == -1) {
                        int iIndexOfU =  GetIndex(Vertices,  u);
                        Predecessors[iIndexOfN] = iIndexOfU;
                        Todo.Add(n);
                  }
            }
}
```

Da jeder Knoten maximal einmal in Todo aufgenommen wird:
$|V|$ Ausführungen

Da wir innerhalb der while Schleife über alle Kanten iterieren:
$|V| \cdot |E|$ Ausführungen

GetIndex iteriert über alle Vertices und hat damit Aufwand $|V|$.
Da die Funktion $|V| \cdot |E|$ mal ausgeführt wird entsteht Aufwand $|V|^2 \cdot |E|$

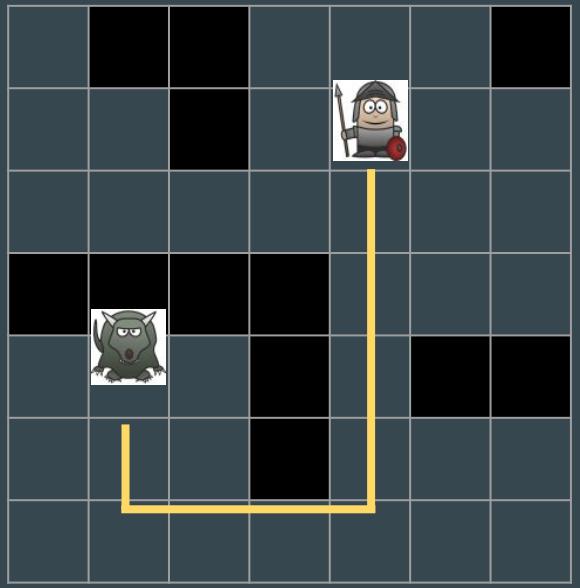
GetIndex iteriert über alle Vertices und hat damit Aufwand $|V|$.
Da die Funktion $|V| \cdot |E|$ mal ausgeführt wird entsteht Aufwand $|V|^2 \cdot |E|$

Der asymptotische Aufwand beträgt momentan $|V|^2 \cdot |E|$
(und wäre ohne den Einsatz des Predecessor-Arrays noch schlechter)

Implikationen der Asymptotischen Laufzeit

Szenario:

- Gegeben sei ein Spiel mit Tile Grid.
  (Vgl. Darstellung rechts - schwarze Felder sind nicht begehbar.)
- Ein KI-Agent sucht einen kürzesten Weg zum Spieler
  (z.B. für einen Nahkampfangriff).
- Bei 60 Frames haben wir pro Frame ca 16ms Rechenzeit.
- Unsere Kacheln entsprechen für die Wegsuche der Knotenmenge V.
- Die Kantenmenge E ist in diesem Fall (nur orthogonale Bewegungen, keine Teleporter etc) relativ klein, $|E| \sim 2|V|$.
- Unsere aktuelle Implementierung der Breitensuche
  (Laufzeit $|V|^2 \cdot |E|$ vgl. letzte Folie) hätte demnach Laufzeit $\sim |V|^3$.
- Wenn wir die Seitenlänge unseres Spielbretts ver-3-fachen, ver-9-fachen wir damit die Anzahl der Kacheln (bzw. Knoten) und ver-729-fachen ($9^3 = 729$) somit die Laufzeit unseres Algorithmus.



Hinweise

- Tatsächlich ist (durch die Implikationen die auf der letzten Folie dargelegt wurden) die asymptotische Komplexität in der Praxis sehr relevant.
- Algorithmen mit kubischer Laufzeit können für Spiele bereits zum Performance-Problem werden.
- Allerdings gibt es noch weitere praktisch relevante Aspekte z.B. (ohne Anspruch auf Vollständigkeit)
  - Werden Objekte ‘by value’ übergeben oder ‘by reference’?
  - Werden (z.B. bei Rückgabewerten von Funktionen) temporäre Objekte erzeugt?
  - Werden zusammenhängende Speicherbereiche (Arrays)

    oder unzusammenhängende Speicherbereiche (LinkedLists) benutzt?
    (Relevant wegen Caching)
  - Werden inline-Funktionen verwendet?
  - Werden virtuelle Methoden aufgerufen?
  - etc

5. Single Source Shortest Paths

Datenstrukturen zur Laufzeitoptimierung

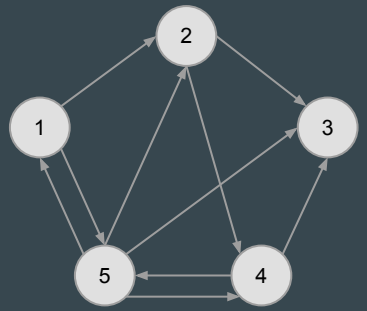
Wir haben soeben für unseren Zusammenhangs-Algorithmus die Laufzeit $|V|^2 \cdot |E|$ ermittelt.
Für einen Graphen mit maximaler Kantenzahl ergibt sich der asymptotische Aufwand $|V|^4$.
Die schlechte Laufzeit ist jedoch auf unsere naive Implementierung zurückzuführen.

Wie können wir die Laufzeit unseres Zusammenhangs-Algorithmus verbessern?

- Die while Schleife, die jeden Knoten einmal betrachtet, lässt sich offenbar nicht vermeiden, denn wir wollen ja von s aus den gesamten Graphen erschließen, um zu prüfen, ob er zusammenhängend ist.
- Es würde innerhalb der while Schleife aber genügen, für jeden Knoten nur die ihm zugehörigen Kanten zu betrachten (anstatt für jeden Knoten über alle Kanten zu iterieren).
- Wir benötigen dafür vorab für jeden Knoten eine explizite Liste seiner Kanten!
- Eine Möglichkeit, diese Voraussetzung zu schaffen, ist (für $V = \{ 1, \dots , n\}$) das Anlegen eines zweidimensionalen boolean Array A der Größe [n,n]. Einen Eintrag A[i,j] == true dieser Matrix interpretieren wir als Kante $\{i,j\} \in E$. Im Umkehrschluss setzen wir beim Erstellen dieser sog. Adjazenzmatrix A[i,j] = true wenn $\{i,j\} \in E$ und A[i,j] = false, falls $\{i,j\} \notin E$.
  In Graphen mit Kantengewichten setzen wir A[i,j] = x für $\{i,j\} \in E$ mit $w(\{i,j\}) = x$ und sonst $A[i,j] = \infty$.

Adjazenzmatrix - Beispiele

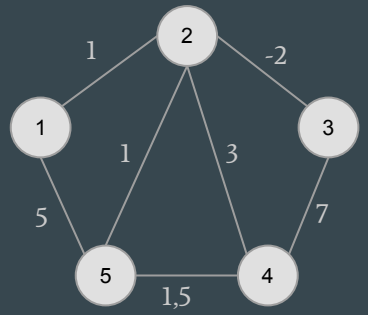
G_1:



A_1:

	1	2	3	4	5
1		t	f	f	t
2	f		t	t	f
3	f	f		f	f
4	f	f	t		t
5	t	t	t	t	

G_2:



A_2:

	1	2	3	4	5
1		1	∞	∞	5
2			-2	3	1
3				7	∞
4					1,5
5					

- Wie man sieht, eignen sich Adjazenzmatrizen zur Darstellung gerichteter, wie auch ungerichteter Graphen mit oder ohne Kantengewichte.
- Für ungerichtete Graphen wird (zur Vermeidung von Redundanz) nur eine Hälfte der Matrix benutzt, dieser Overhead ist jedoch vernachlässigbar.
- Für (sog. 'dünne') Graphen mit sehr wenigen Kanten (d.h. für $|E| \ll |V|^2$) sind i.d.R. wegen ihres geringeren Speicherbedarfs sogenannte Adjazenzlisten zu bevorzugen.

Laufzeitanalyse Zusammenhangs-Algorithmus

Wir ermitteln nun erneut die asymptotische Laufzeit unseres Zusammenhangs-Algorithmus - diesmal unter Verwendung einer Adjazenzmatrix A.

Zusammenhangs-Algorithmus

```
bool[,] A;
int[] Predecessors;
IntList Todo;
while (Todo.GetSize() > 0) {
      int u = Todo.Remove();
      for (int n = 0; n < A[u].length; n++)
            if (A[u,n])
                  if (Predecessors[n] == -1)
                        Predecessors[n] = u;
                        Todo.Add(n);
```

Da jeder Knoten maximal einmal in Todo aufgenommen wird: $|V|$ Ausführungen

Da wir die for-Schleife (über eine Zeile der Matrix mit $|V|$ Einträgen) innerhalb der while Schleife ausführen ergeben sich $|V|^2$ Ausführungen

Wie bereits erwähnt sind die Zugriffe auf das Predecssor-Array und das Hinzufügen/Entfernen für Warteschlangen in konstanter Zeit möglich.

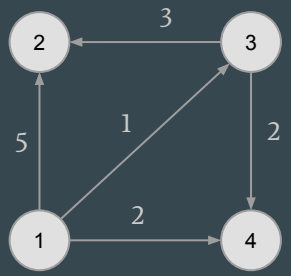
Der asymptotische Aufwand beträgt nun $|V|^2$

Kürzeste Wege in kantengewichteten Graphen

Der Algorithmus Breitensuche (im Gegensatz zur Tiefensuche) gehorcht dem Prinzip, alle Knoten im Graphen gemäß ihrer Distanz von s aus zu erschließen. Für zwei Knoten u_1 und u_2 mit dem gemeinsamen Nachbarn n folgt:

Ist $dist(s, u_1) < dist(s, u_2)$, so wird u_1 im Algorithmus vor u_2 als Knoten u betrachtet werden.
Dadurch ist gewährleistet, dass n (weitere Nachbarn u_k ausgeschlossen) erstmals von u_1 aus erreicht wird, und u_1 als seinen Vorgänger auf einem kürzesten Weg von s aus in die Vorgänger-Tabelle eintragen darf.
(u_2 als Vorgänger für n einzutragen wäre falsch, da jeder Weg $(s, ..., u_2, n)$ echt länger wäre, als unser $(s, ... , u_1, n)$.)

Betrachte den folgenden kantengewichteten Graphen (mit $w: E \to \mathbb{R}^+$) für die Suche kürzester Wege ausgehend von 1:



- Knoten 1 erschließt zunächst seine Nachbarn (2,3,4) (mit den Distanzen 5, 1, bzw. 2).
- Für die Knoten 2 und 4 können wir nun noch keinen Vorgänger eintragen, denn ggf. könnten kürzere Wege (z.B. über 3) als die direkten Wege (1,2) bzw. (1,4) existieren.
- Für den Knoten 3 hingegen lässt sich feststellen:
  Die Kante (1,3) ist unter allen Möglichkeiten, den Knoten 1 überhaupt zu verlassen, die günstigste. Ein Umweg über 2 oder 4 kann (sofern negative Kantengewichte verboten sind) nur länger werden. Wir dürfen daher 1 als Vorgänger von 3 eintragen!
- Die günstigste Weise, die Knotenmenge {1, 3} zu verlassen, führt dann über die Kante (1,4)...

Dijkstra Single Source Shortest Paths Algorithmus

Wir haben soeben eine Methode skizziert, die einen/alle kürzesten Weg(e) von einem Knoten s aus in kantengewichteten Graphen bestimmt. Diese Methode ist als Dijkstra Single Source Shortest Paths Algorithmus bekannt.

Der Algorithmus bzw. die zugrundeliegende Logik kann wie folgt skizziert werden:

- Teile V in finalisierte (anfangs leer) und nicht-finalisierte Knoten (anfangs V) ein.
- Speichere für alle Knoten ihre Distanz zu s (anfangs 0 für s selbst und ∞ für alle anderen Knoten)
- Solange es noch nicht-finalisierte Knoten gibt, wähle aus diesen einen Knoten u mit minimaler Distanz
    - Markiere u als finalisiert (‘günstiger’ können wir die finalisierten Knoten nicht verlassen)
    - Für jeden nicht-finalisierten Nachbarn n von u (verbunden mittels Kante e)
        - berechne Distanz[u] + w(e) (das wären die Kosten, um n von s aus über u zu erreichen)
        - falls diese Summe kleiner ist als (die bisherige) Distanz[n]
            - aktualisiere Distanz[n] = Distanz[u] + w(e) (diesen Vorgang nennt man Relaxieren)

Falls die Distanz zu einem gewissen Knoten t gesucht ist, brich ab, sobald dieser als Knoten u finalisiert wurde.
Falls alle in maximaler Distanz d erreichbaren Knoten gefragt sind, bricht ab, sobald der erste Knoten mit Distanz d’ > d finalisiert wurde.

Algorithmus von Dijkstra

Wir haben soeben eine Methode skizziert, die einen/alle kürzesten Weg(e) von einem Knoten s aus in kantengewichteten Graphen bestimmt. Diese Methode ist als Dijkstra Single Source Shortest Paths Algorithmus

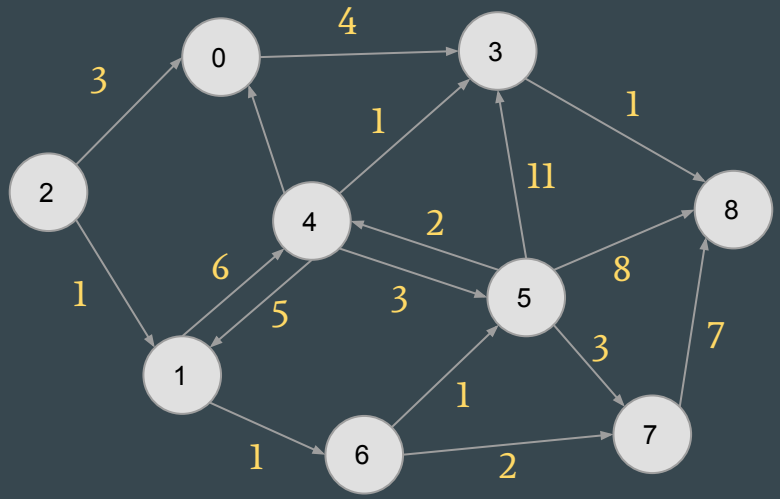
Dijkstra Single Shortest Paths Algorithmus

```
int[,] A; // Kantenmenge E gegeben als Adjazenzmatrix
int[] Distance; // initialisiert mit 0 für s und ∞ sonst
int[] Predecessor; // initialisiert mit -1
Unfinalized = V;
while (es gibt ein u ∈ Unfinalized mit Distance[u] != ∞) {
      int u = Entferne aus Unfinalized mit minimaler Distance[u];
      for (int n = 0; n < A[u].length; n++)
            if (A[u,n] < ∞ /* && n ∈ Unfinalized */)
                  int distOverU = Distance[u] + A[u,n];
                  if (distOverU < Distance[n])
                        Distance[n] = distOverU;
                        Predecessor[n] = u;
```

- Im Gegensatz zur Breitensuche müssen wir damit rechnen, dass ein früh erreichter Knoten erst spät (als u) finalisiert wird.
- Unfinalized kann daher nicht als Warteschlange gehalten werden.
- Wir gehen hier von einer 'naiven' Implementierung von Unfinalized als bool-Array aus, d.h. wir müssen für die Auswahl von u über Unfinalized iterieren.
- Alternativ kann Unfinalized (für eine bessere asymptische Laufzeit) als Priority Queue implementiert werden.
- Nach Ablauf von Dijkstra bestimmen wir einen kürzesten Weg mittels Backtracking wie bei der Breitensuche.

Übung

Wende den Algorithmus von Dijkstra auf den folgenden Graphen mit Startknoten 2 an.



Lösung

step \ dist	0	1	2	3	4	5	6	7	8	Unfinalized
init	∞	∞	0	∞	∞	∞	∞	∞	∞	{ 0,1,2,3,4,5,6,7,8 }
u = 2	3	1	-	-	-	-	-	-	-	{ 0,1,3,4,5,6,7,8 }
u = 1	-	-	-	-	7	-	2	-	-	{ 0,3,4,5,6,7,8 }
u = 6	-	-	-	-	-	3	-	4	-	{ 0,3,4,5,7,8 }
u = 0	-	-	-	7	-	-	-	-	-	{ 3,4,5,7,8 }
u = 5	-	-	-	-	5	-	-	-	11	{ 3,4,7,8 }
u = 7	-	-	-	-	-	-	-	-	-	{ 3,4,8 }
u = 4	-	-	-	6	-	-	-	-	-	{ 3,8 }
u = 3	-	-	-	-	-	-	-	-	-	{ 8 }

← Hier wäre auch u = 5 korrekt

← |Unfinalized| = 1 erlaubt Abbruch

Algorithmus von Dijkstra

Wir ermitteln nun die asymptotische Laufzeit des Dijkstra-Algorithmus unter Verwendung einer Adjazenzmatrix A und einem boolean Array Unfinalized. Beachte, dass die Laufzeit z.B. mittels Einsatz eines Fibonacci-Heap für Unfinalized und Adjazenzlisten für E auf $O(|V| \cdot \log(|V|) + |E|)$ gesenkt werden kann.

Dijkstra Single Shortest Paths Algorithmus

```
int[,] A; // Kantenmenge E gegeben als Adjazenzmatrix
int[] Distance; // initialisiert mit 0 für s und ∞ sonst
int[] Predecessor; // initialisiert mit -1
Unfinalized = V;
while (es gibt ein u ∈ Unfinalized mit Distance[u] != ∞) {
      int u = Entferne aus Unfinalized mit minimaler Distance[u];
      for (int n = 0; n < A[u].length; n++)
            if (A[u,n] < ∞ /* && n ∈ Unfinalized */)
                  int distOverU = Distance[u] + A[u,n];
                  if (distOverU < Distance[n])
                        Distance[n] = distOverU;
                        Predecessor[n] = u;
```

- Da jeder Knoten maximal einmal in finalisiert wird: |V| Ausführungen
- Da ganz Unfinalized durchlaufen werden muss: |V| Ausführungen
- Da eine Zeile der Matrix durchlaufen wird: |V| Ausführungen

Alle weiteren Array-Zugriffe, etc sind in konstanter Zeit möglich.

Der asymptotische Aufwand beträgt $|V|^3$

Fazit

Informatik beinhaltet (unter anderem)

- die Kenntnis diverser (abstrakter) Problemstellungen
- und deren Formalisierung
- sowie optimale Lösungsansätze für diese Problemstellungen
- und deren Komplexität
- außerdem mathematische und kombinatorische Grundlagen als Ausgangspunkt für
- das Vermögen, Probleme zu abstrahieren und einer bekannten Problemstellung zuzuordnen
- bzw. eine eigene Lösung für das Problem zu erarbeiten
- und sich dabei der Laufzeit des eigenen Codes bewusst zu sein
- sowie die Fähigkeit, diese Laufzeit ggf. (z.B. durch geeignete Datenstrukturen) zu optimieren.