



Technische
Universität
Braunschweig



Algorithmen und Datenstrukturen 2 – Übung #5

Komplexität, Reduktionen und Hashing

Ramin Kosfeld und Chek-Manh Loi

08.07.2026

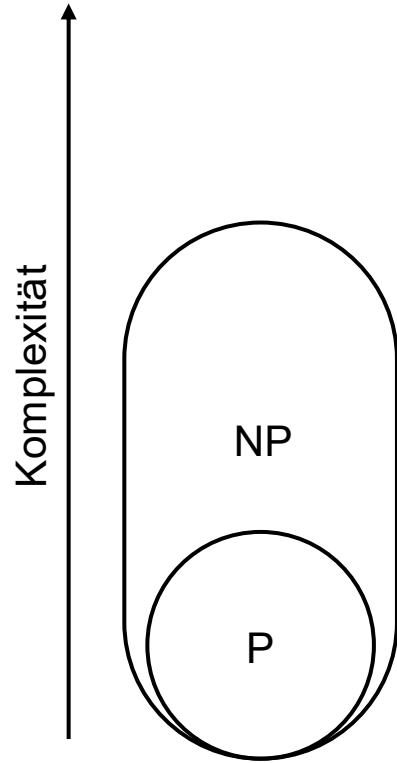
Heute

- Komplexität
 - Reduktionen
- Hashing
 - Modulo
 - Coding-Interview
 - ...

$$\varphi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



Komplexitätsklassen

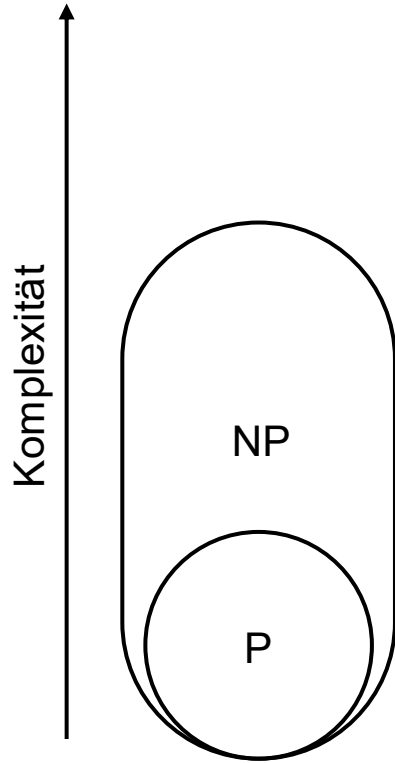


Ein Problem Π liegt in

- **P**, falls es einen Polynomialzeitalgorithmus gibt, der Π korrekt löst.
- **NP**, falls jede Lösung in polynomieller Zeit verifiziert werden kann.



Komplexitätsklassen



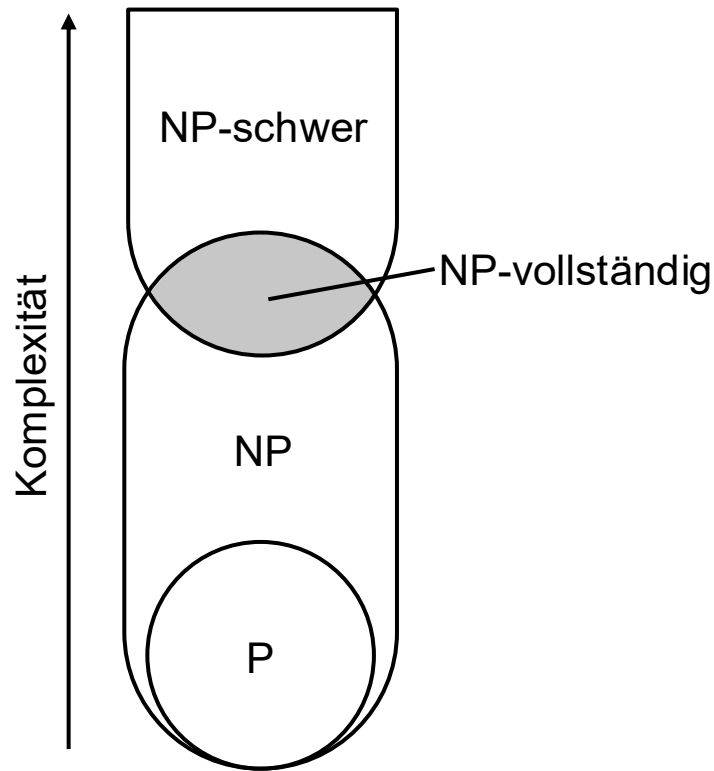
Ein Problem Π liegt in

- **P**, falls es einen Polynomialzeitalgorithmus gibt, der Π korrekt löst.
- **NP**, falls jede Lösung in polynomieller Zeit verifiziert werden kann.



1

Komplexitätsklassen



Ein Problem Π liegt in

- **P**, falls es einen Polynomialzeitalgorithmus gibt, der Π korrekt löst.
- **NP**, falls jede Lösung in polynomieller Zeit verifiziert werden kann.

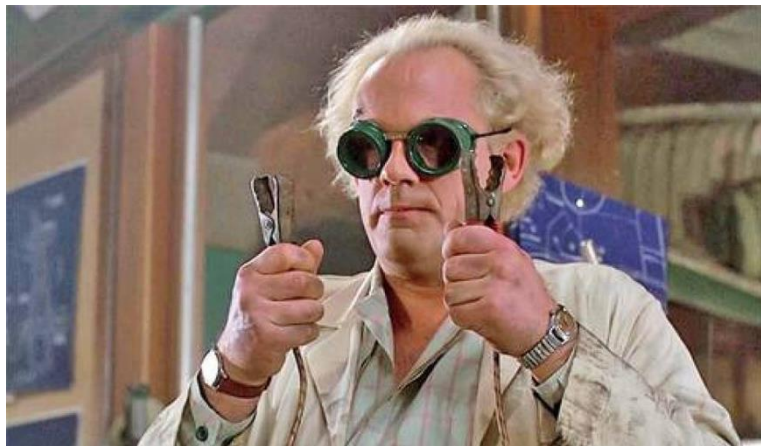
= mindestens so schwer wie jedes Problem aus NP

Ein Problem Π heißt

- **NP-schwer**, falls jedes Problem aus NP auf Π *reduziert* werden kann.
- **NP-vollständig**, falls Π NP-schwer ist und in NP liegt.

*Also die komplexesten Probleme in NP – denn mit ihnen kann man **auch alle anderen Probleme in NP** lösen!*

Reduktionen



Zeige: Ein Problem ist *mindestens* so schwer wie ein anderes.

Mindset: Ich weiß nicht, wie man Problem B löst. Aber *wenn* ich eine Box habe, die Problem B lösen kann ... *dann* kann man damit auch Problem A lösen.

(Mit polynomielltem Mehraufwand.)



Kann B lösen?



Kann A lösen!

(so ganz nebenbei. Wer weiß, was er noch alles kann...)



B ist mindestens so schwer wie A.

$$B \geq_p A$$

Reduktionen

Wir sagen:

“Wir reduzieren (von)
Problem A auf Problem B”



Reduktionen



Besitzen T , T^{-1} und $\mathcal{Alg}$ polynomielle Laufzeit,
kann Problem A in polynomieller Zeit gelöst werden.

Reduktionen



Ist Problem A NP-schwer und besitzen T und T^{-1} polynomielle Laufzeit, dann muss Problem B auch NP-schwer sein.

NP-Schwere

Ein Problem Π heißt *NP-schwer*, falls für jedes Problem $\Pi' \in \text{NP}$ eine Polynomialzeit-Reduktion von Π' auf Π existiert.

In der Literatur wird oft auch $\Pi' \leq_p \Pi$ geschrieben, wenn es eine Polynomialzeit-Reduktion von Π' auf Π gibt.

$\Pi' \leq_p \Pi$ heißt, man kann in polynomieller Zeit eine Instanz I' von Π' in eine Instanz I von Π transformieren, die eine Ja-Instanz ist gdw. I' eine Ja-Instanz ist.

$\Pi' \leq_p \Pi$: „ Π' ist höchstens so schwer wie Π “

Die Relation $\leq_p$ ist transitiv, da man polynomielle Reduktionen verketteten kann.

Um zu zeigen, dass ein Problem Π NP-schwer ist, reicht es also, eine Reduktion **von einem** als NP-schwer bekannten Problem Π' auf Π durchzuführen.

Ein paar Probleme

Ein paar Probleme

3-SAT

$$\varphi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$

Gegeben

Logische Formel φ in konjunktiver Normalform mit

- m Klauseln
- n Variablen
- Genau drei Literale pro Klausel

Frage

Gibt es eine φ erfüllende Belegung der Variablen?

3-SAT ist quasi die Mutter aller NP-vollständigen Probleme.
Alle NP-Schwere-Reduktionen reduzieren am Ende von 3-SAT.

Hamilton Cycle

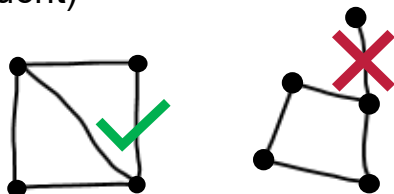
Hamiltonian Cycle

Gegeben

Ungerichteter Graph $G = (V, E)$

Frage

Gibt es einen Hamiltonkreis in G ?
(Einen Kreis, der jeden *Knoten* genau 1x besucht)



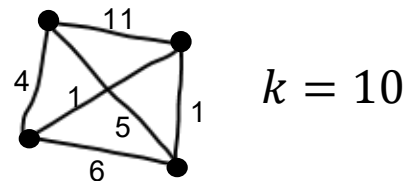
Traveling Salesman Problem

Gegeben

Vollständiger Graph $G = (V, E)$ mit
Kantenkosten $c: E \rightarrow \mathbb{R}^+$ und eine Zahl $k \in \mathbb{R}^+$

Frage

Gibt es eine Tour in G mit Kantenkosten
maximal k ?



Hamilton Cycle | Traveling Salesman Problem



Zeige:
Hamiltonian Cycle $\leq_p$ TSP

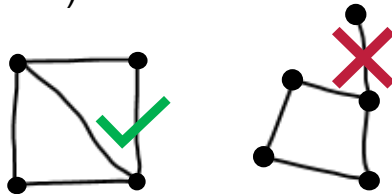
Nutze TSP-“Solver”,
um Hamilton Cycles zu finden!

Gegeben

Ungerichteter Graph $G = (V, E)$

Frage

Gibt es einen Hamiltonkreis in G ?
(Einen Kreis, der jeden *Knoten* genau
1x besucht)

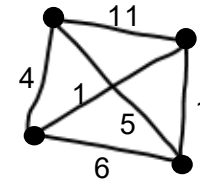


Gegeben

Vollständiger Graph $G = (V, E)$ mit
Kantenkosten $c: E \rightarrow \mathbb{R}^+$ und eine Zahl $k \in \mathbb{R}^+$

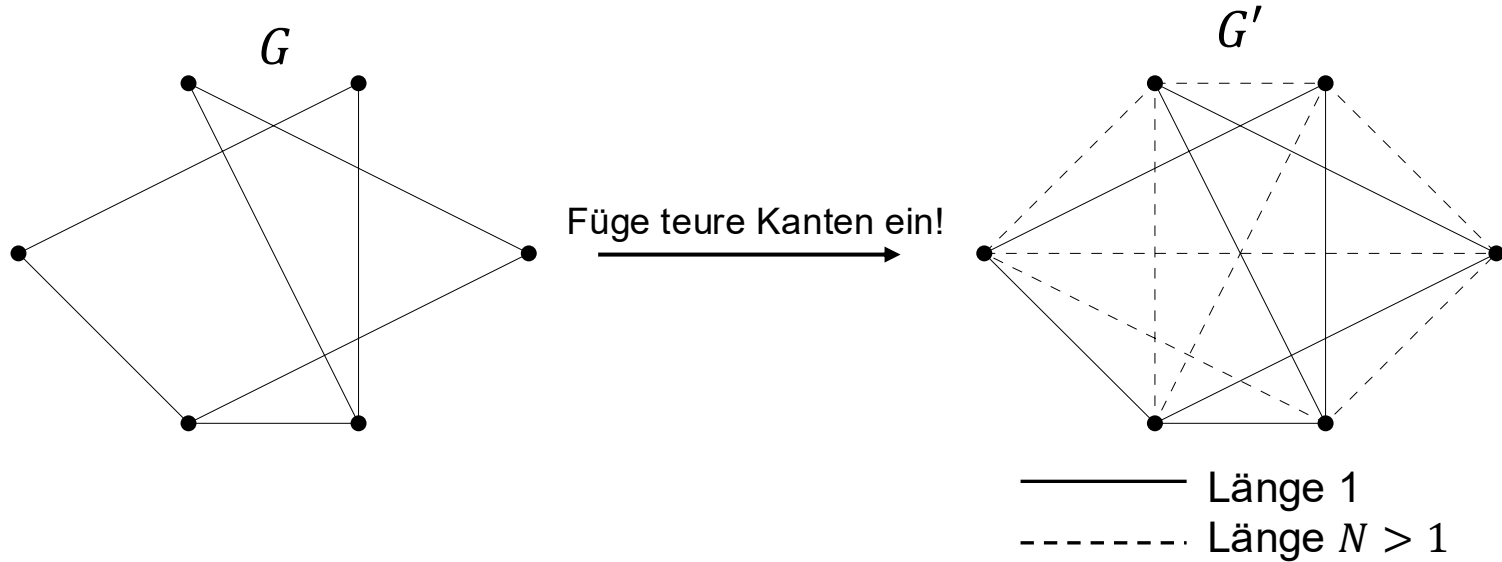
Frage

Gibt es eine Tour in G mit Kantenkosten
maximal k ?



$k = 10$

Reduktion von HC auf TSP



Zu zeigen

G besitzt genau dann einen Hamiltonkreis, wenn G' eine Tour der Länge $n := |V|$ besitzt.

Beweis Korrektheit

„ $\Rightarrow$ “

Wähle die gleichen Kanten von G in G' . Diese haben die Kosten n .

„ $\Leftarrow$ “

Besitzt G keinen Hamiltonkreis, so muss in G' mindestens eine Kante mit Gewicht N benutzt werden. Somit hat die Tour ein Gewicht von mindestens $n - 1 + N > n$, da $N > 1$.

Laufzeit

Es müssen $O(n^2)$ Kanten hinzugefügt werden.

Alle $O(n^2)$ Kanten müssen mit Kosten versehen werden.

Insgesamt also eine Laufzeit von $O(n^2)$.

Konsequenz

TSP kann nicht approximiert werden, es sei denn $P = NP$.

Hamiltonian Cycle $\leq_p$ TSP

Directed HC $\leq_p$ Hamiltonian Cycle $\leq_p$ TSP

Directed Hamiltonian Cycle

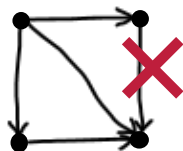
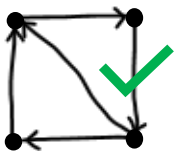
Directed Hamiltonian Cycle

Gegeben

Gerichteter Graph $D = (V, E)$

Frage

Gibt es einen *gerichteten* Hamiltonkreis in D ?



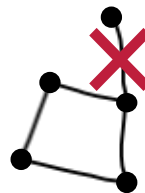
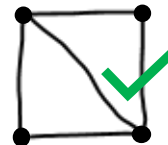
Hamiltonian Cycle

Gegeben

Ungerichteter Graph $G = (V, E)$

Frage

Gibt es einen Hamiltonkreis in G ?
(Einen Kreis, der jeden *Knoten* genau 1x besucht)



Directed Hamiltonian Cycle



Zeige:
Directed Hamiltonian Cycle $\leq_p$ Hamiltonian Cycle

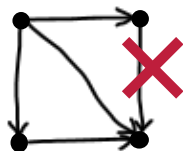
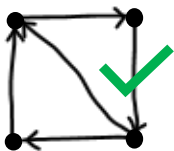
Nutze ungerichteten HC-
“Solver”, um gerichteten
Hamiltonkreis zu finden!

Gegeben

Gerichteter Graph $D = (V, E)$

Frage

Gibt es einen *gerichteten* Hamiltonkreis in D ?

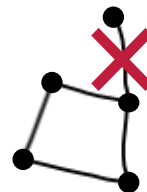
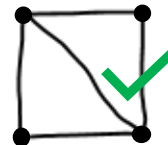


Gegeben

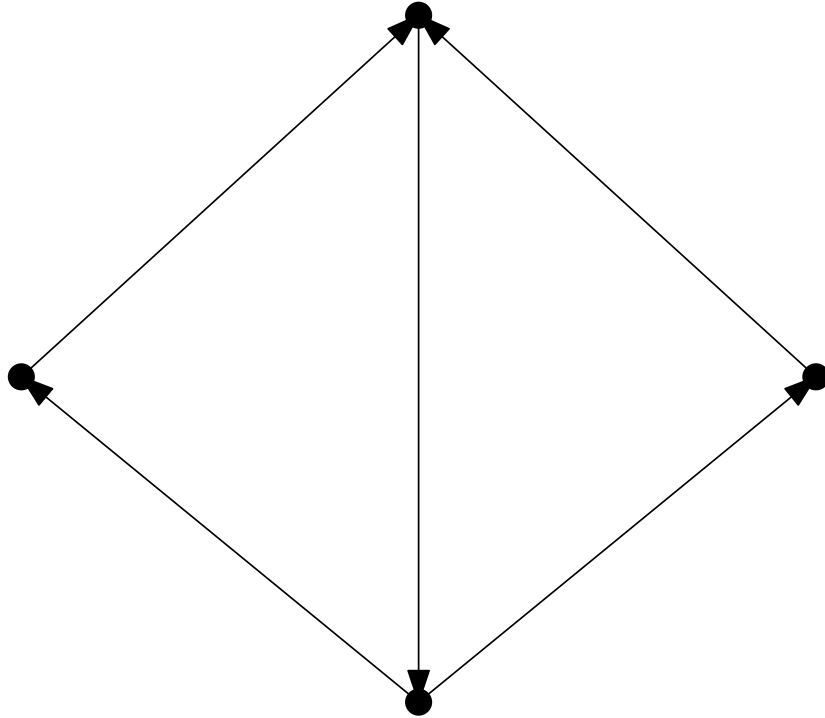
Ungerichteter Graph $G = (V, E)$

Frage

Gibt es einen Hamiltonkreis in G ?
(Einen Kreis, der jeden *Knoten* genau
1x besucht)



Reduktion von DHC auf HC

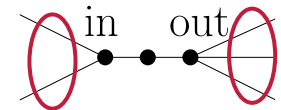
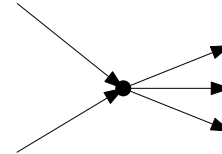


Wie kann man garantieren, dass im ungerichteten Graphen...

...nur eine der ursprünglich *eingehenden* Kanten verwendet wird?

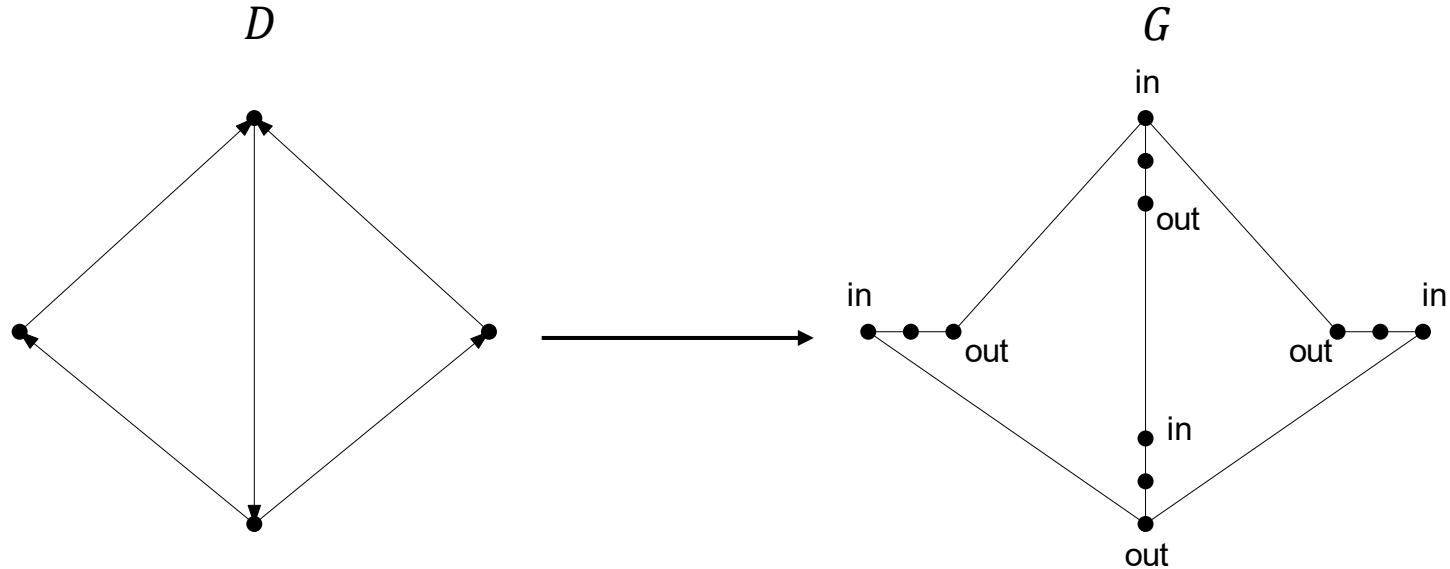
...nur eine der ursprünglich *ausgehenden* Kanten verwendet wird?

Idee: Teile Knoten auf!



Jeweils nur eine möglich!

Reduktion von DHC auf HC



Zu zeigen

D besitzt genau dann einen gerichteten Hamiltonkreis, wenn G einen Hamiltonkreis besitzt.

Directed HC $\leq_p$ Hamiltonian Cycle $\leq_p$ TSP

$3\text{-SAT} \leq_p \text{Directed HC} \leq_p \text{Hamiltonian Cycle} \leq_p \text{TSP}$

3-SAT

3-SAT

Gegeben

3-SAT ist NP-schwer.

Logische Formel φ in konjunktiver Normalform mit

- Genau drei Literalen pro Klausel

Frage

Gibt es eine φ erfüllende Belegung der Variablen?

$$\varphi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

Directed Hamiltonian Cycle

Gegeben

Gerichteter Graph $D = (V, E)$

Frage

Gibt es einen *gerichteten* Hamiltonkreis in D ?



3-SAT



Zeige:
3-SAT $\leq_p$ **Directed Hamiltonian Cycle**

Zeige: Ein Algorithmus, der gerichtete Hamiltonpfade findet, kann auch 3-SAT Formeln entscheiden!

Gegeben

Logische Formel φ in konjunktiver Normalform mit

- Genau drei Literalen pro Klausel

Frage

Gibt es eine φ erfüllende Belegung der Variablen?

$$\varphi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

Gegeben

Gerichteter Graph $D = (V, E)$

Frage

Gibt es einen *gerichteten* Hamiltonkreis in D ?

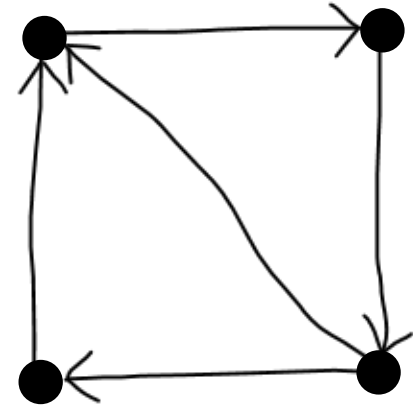
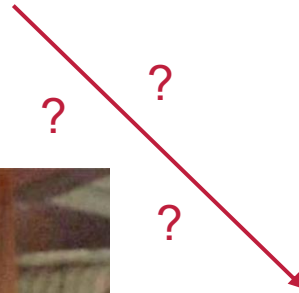


Reduktion 3SAT auf DHC

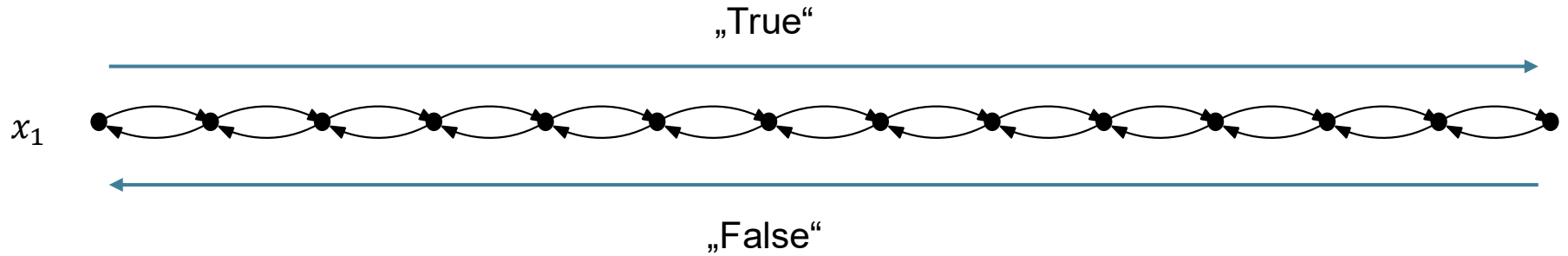
$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4)$$

Wie baue ich
damit Variablen-
Gadgets?

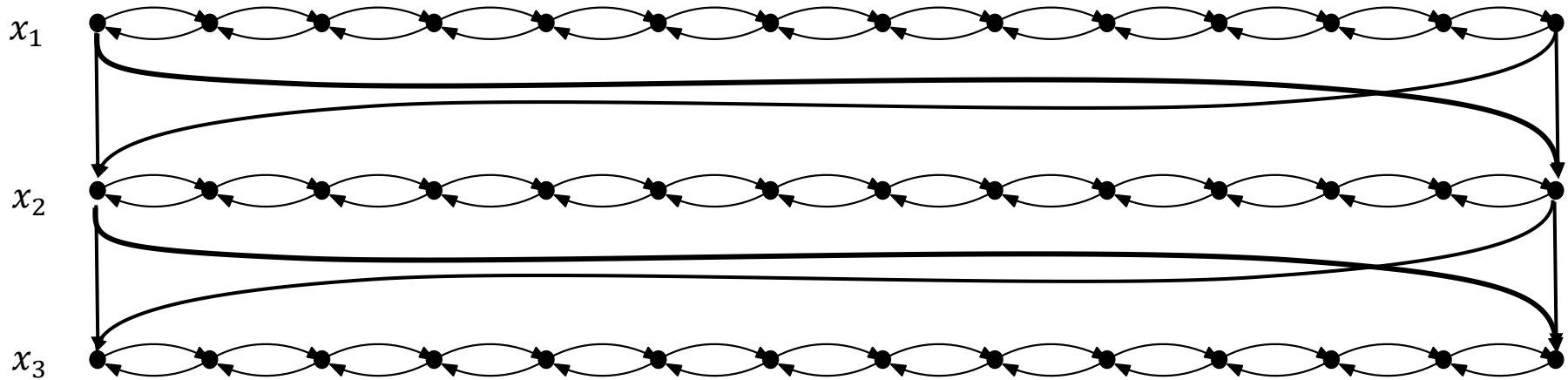
Was ist mit
Klausel-
Gadgets?



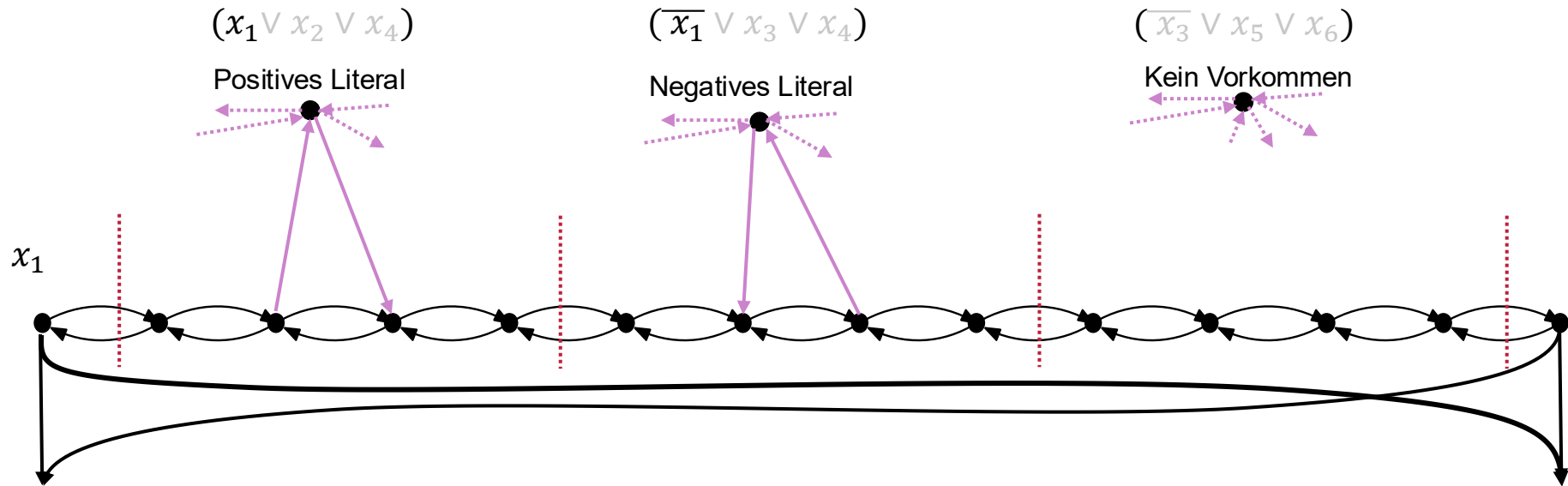
Variablen-Gadgets



Variablen-Gadgets

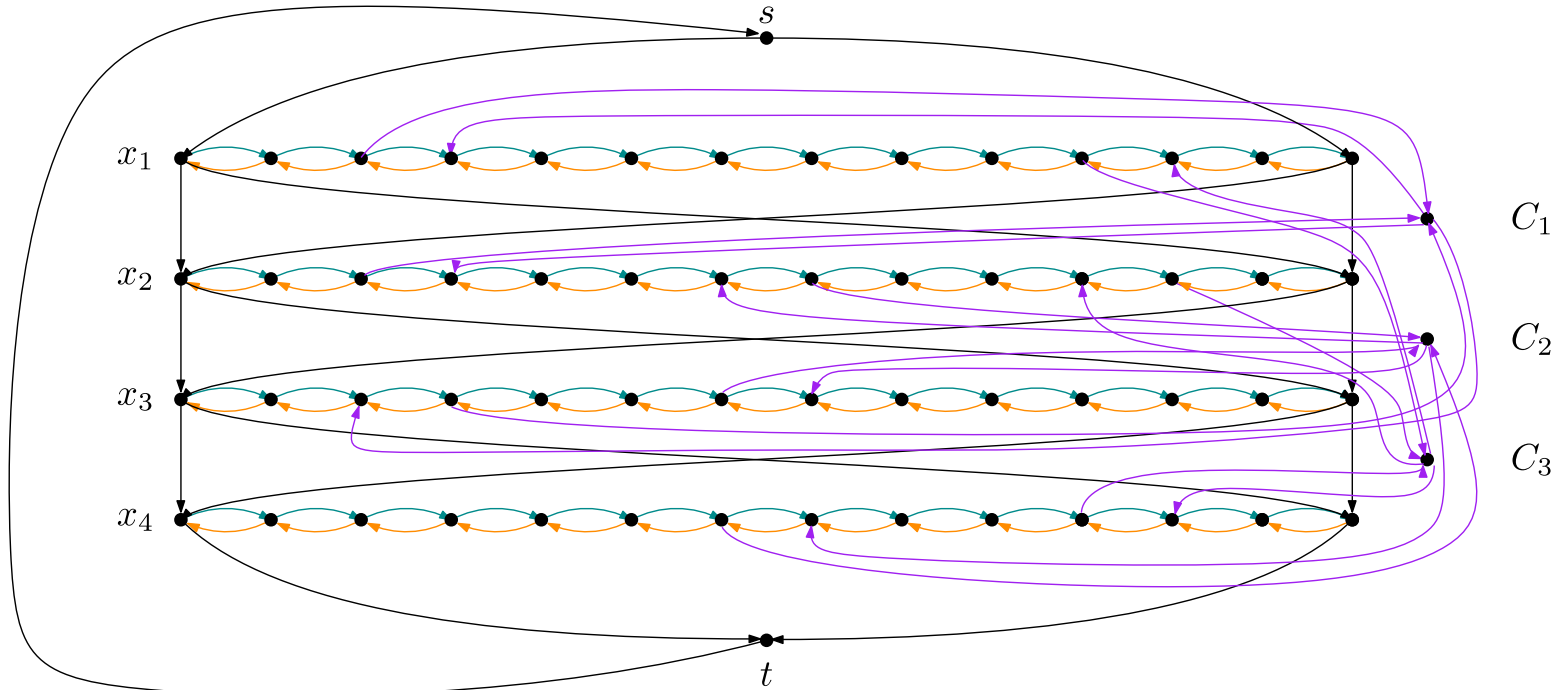


Klausel-Gadgets



Beispiel der Reduktion

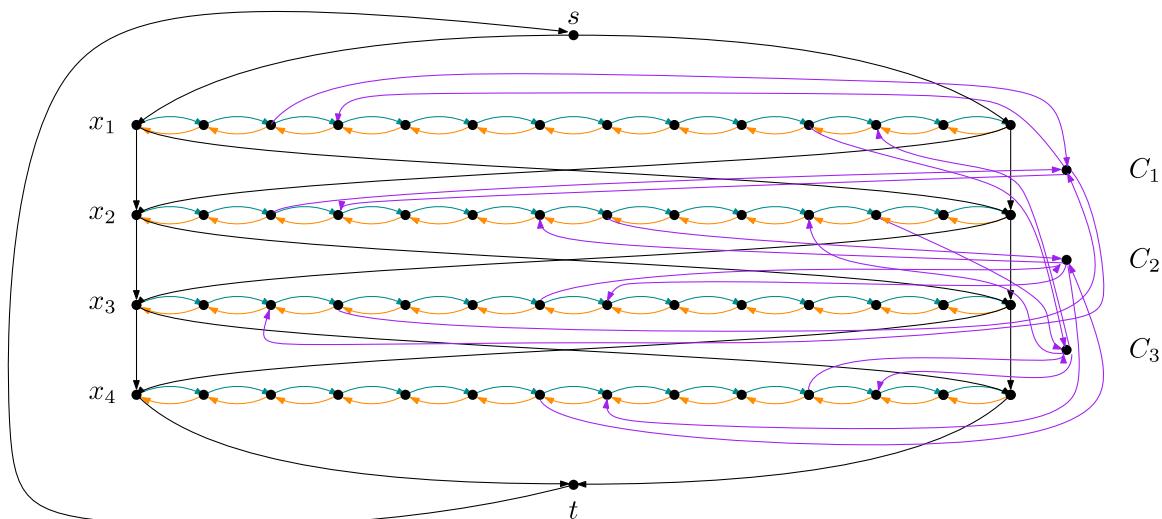
$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_2} \vee x_3 \vee x_4) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



Korrektheit

„Wenn die Formel erfüllbar ist, dann gibt es einen Hamiltonkreis.“

Laufe die Variablengadgets in der richtigen Richtung ab und laufe zwischendurch die Klauseln ab. (Nach Konstruktion möglich)



Korrektheit

„Wenn die Formel erfüllbar ist, dann gibt es einen Hamiltonkreis.“

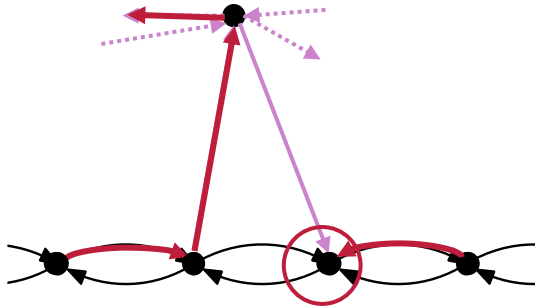
Laufe die Variablengadgets in der richtigen Richtung ab und laufe zwischendurch die Klauseln ab. (Nach Konstruktion möglich)

„Wenn es einen Hamiltonkreis gibt, dann ist die Formel erfüllbar.“

Dazu müssen wir zeigen:

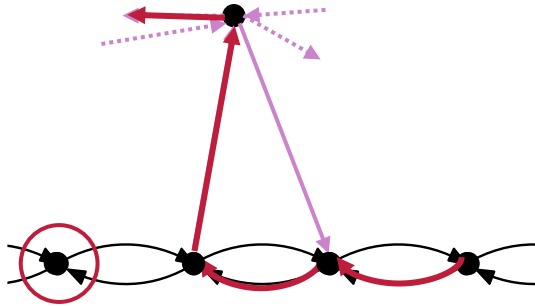
1. Geht man zu einer Klausel, muss man zur selben Variable zurück.
2. Man darf nur Klauseln ablaufen, wenn die richtige Richtung gewählt wurde.

Man muss wieder zurück...



Annahme: Man geht nicht zurück.

Man muss richtig rum laufen...



Annahme: Man läuft Klauseln falsch ab.

Korrektheit

„Wenn es einen Hamiltonkreis gibt, dann ist die Formel erfüllbar.“

Dazu müssen wir zeigen:

1. Geht man zu einer Klausel, muss man zur selben Variable zurück. ✓
2. Man darf nur Klauseln ablaufen, wenn die richtige Richtung gewählt wurde. ✓

Also:

1. Variablen-Gadgets werden in einem Zug durchlaufen
 - Die Richtung gibt uns *true* oder *false*
2. Gibt es keine Belegung der Variablen, sodass φ erfüllt wird, so gibt es immer mindestens ein Klausel-Gadget, das nicht abgelaufen werden kann.
 - Es gibt also keinen Hamiltonkreis.

Laufzeit der Transformation

Wir erstellen einen Graphen mit

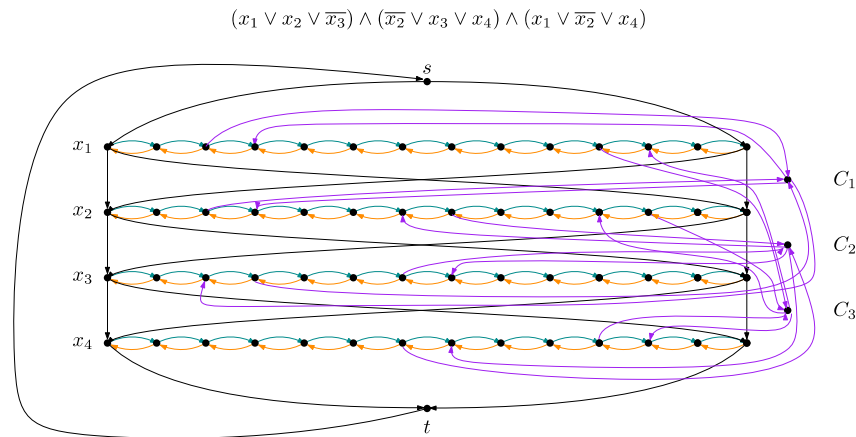
- $4nm + m + 2$ Knoten
- $\mathcal{O}(nm)$ vielen Kanten

Wir brauchen also $\mathcal{O}(nm)$ Zeit, den Graphen zu erstellen.

Lösung für 3-SAT berechnen

- $\mathcal{O}(1)$

Uns interessiert nur, **ob** die Formel erfüllbar ist!



Schritte für einen NP-Schwere Beweis

Zeige: Problem Π ist NP-schwer

- Suche ein geeignetes NP-schweres Problem Π' .
- Reduziere **von** Π' **auf** Π : $\Pi' \leq_p \Pi$
- Beweise Korrektheit:
„Für jede Instanz $I_{\Pi'}$ von Π' gibt es genau dann eine Lösung, wenn es für die Instanz $T(I_{\Pi'})$ von Π eine Lösung gibt.“
- Beweise polynomielle Laufzeit von T und T^{-1} .

Beliebte Fehler

- Nur für ein Beispiel gezeigt.
- Reduktion falsch herum.
- Im Korrektheitsbeweis nur eine Richtung gezeigt.
- Laufzeit der Transformation nicht berücksichtigt.
- Codierungsgröße von $T(I_{\Pi'})$ ist nicht polynomiell durch die Größe von $I_{\Pi'}$ beschränkt.

Fragen?

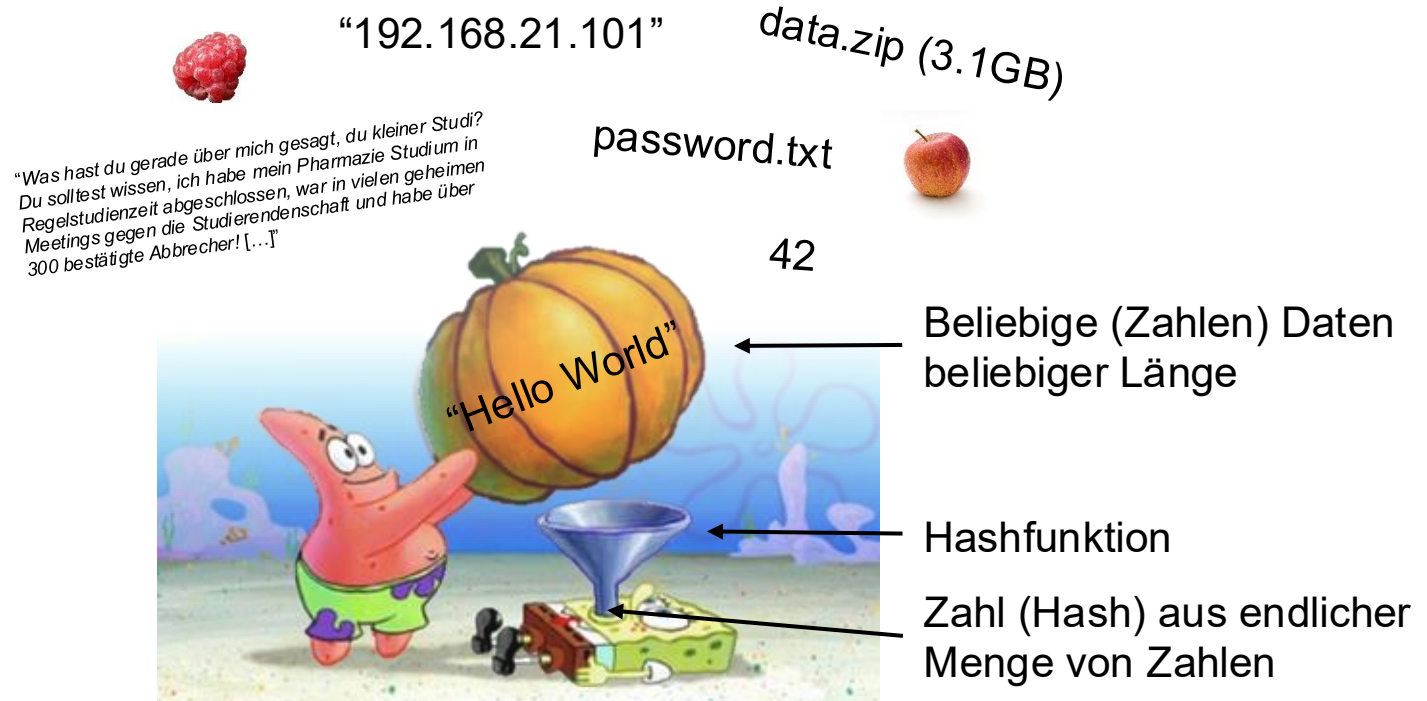
Hashing

Hashing



(Symbolbild)

Hashing



890b 23a8	8410 b221	590a 23b8	2401 ba98
b0a1 83f1	09c8 89dd	78bc 08ff	3bcd f783

Hashing: Kontexte

Diese Veranstaltung

- Hash-Tabellen

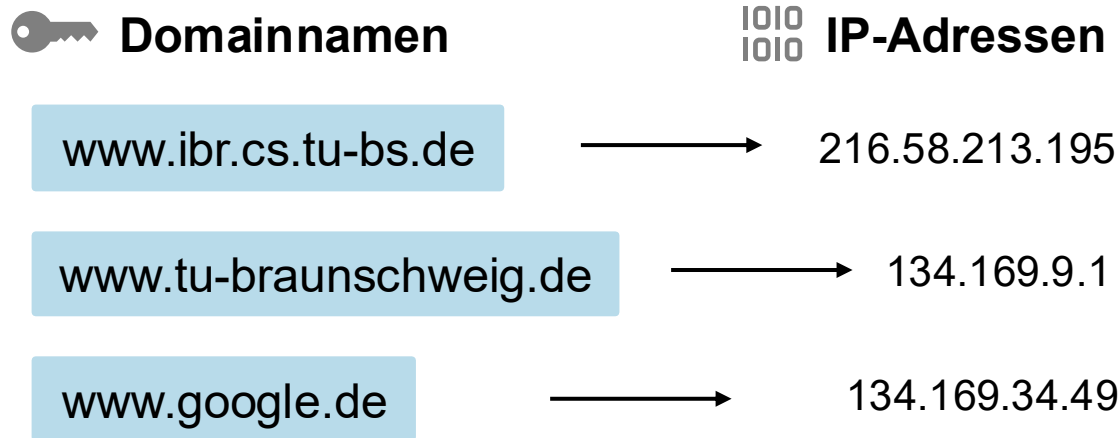
Weitere Kontexte / Anwendungen

- Checksums
- Kryptografie
- Security
- Datenbanken
- Caches
- ...

Dictionaries

Assoziative Datenstruktur, speichere Elemente (Value) unter einem Schlüssel (Key).

Beispiel



Keine Reihenfolge garantiert. Hier kann man nicht sortieren, Vorgänger finden etc.

Ein Blick zurück zu AuD 1

Wir könnten Dictionaries mit Datenstrukturen aus AuD1 bauen ...

Laufzeiten

Suchen

Einfügen

Löschen

Ein Blick zurück zu AuD 1

Wir könnten Dictionaries mit Datenstrukturen aus AuD1 bauen ...

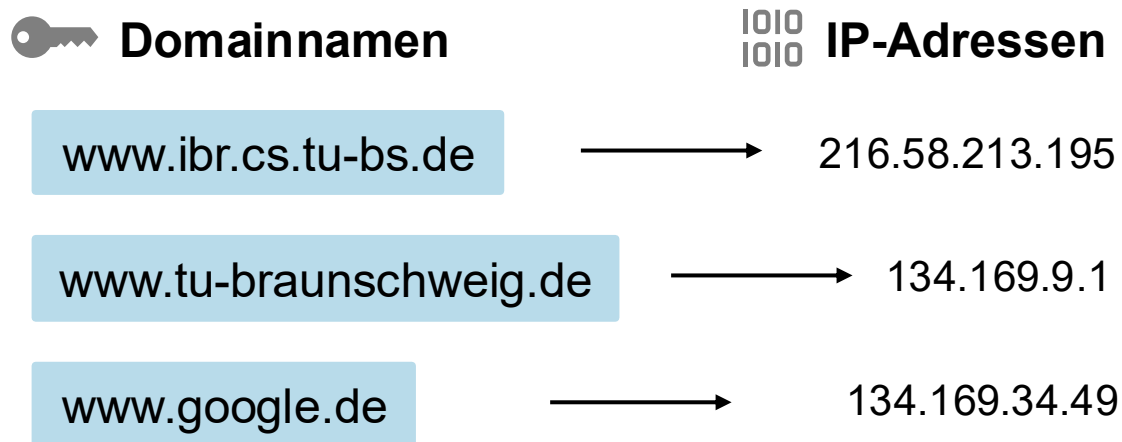
<i>Laufzeiten</i>	Verkettete Liste	Binäre Bäume	AVL- Bäume	(Max) Heaps
Suchen	$\mathcal{O}(n)$	$\mathcal{O}(h)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Einfügen	$\mathcal{O}(1)$	$\mathcal{O}(h)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Löschen	$\mathcal{O}(n) / \mathcal{O}(1)$	$\mathcal{O}(h)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$

Mit Hashing geht das tatsächlich noch schneller!

Hash-Tabellen

Assoziative Datenstruktur mit extrem schnellen Operationen für Suchen, Einfügen und Löschen.

Beispiel



Hash-Tabellen

Schlüssel

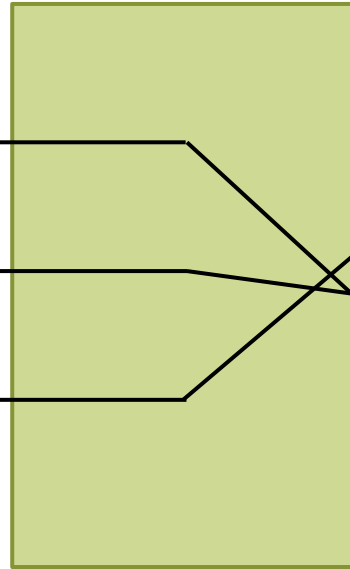
Hashfunktion

Hashtabelle

www.ibr.cs.tu-bs.de

www.tu-braunschweig.de

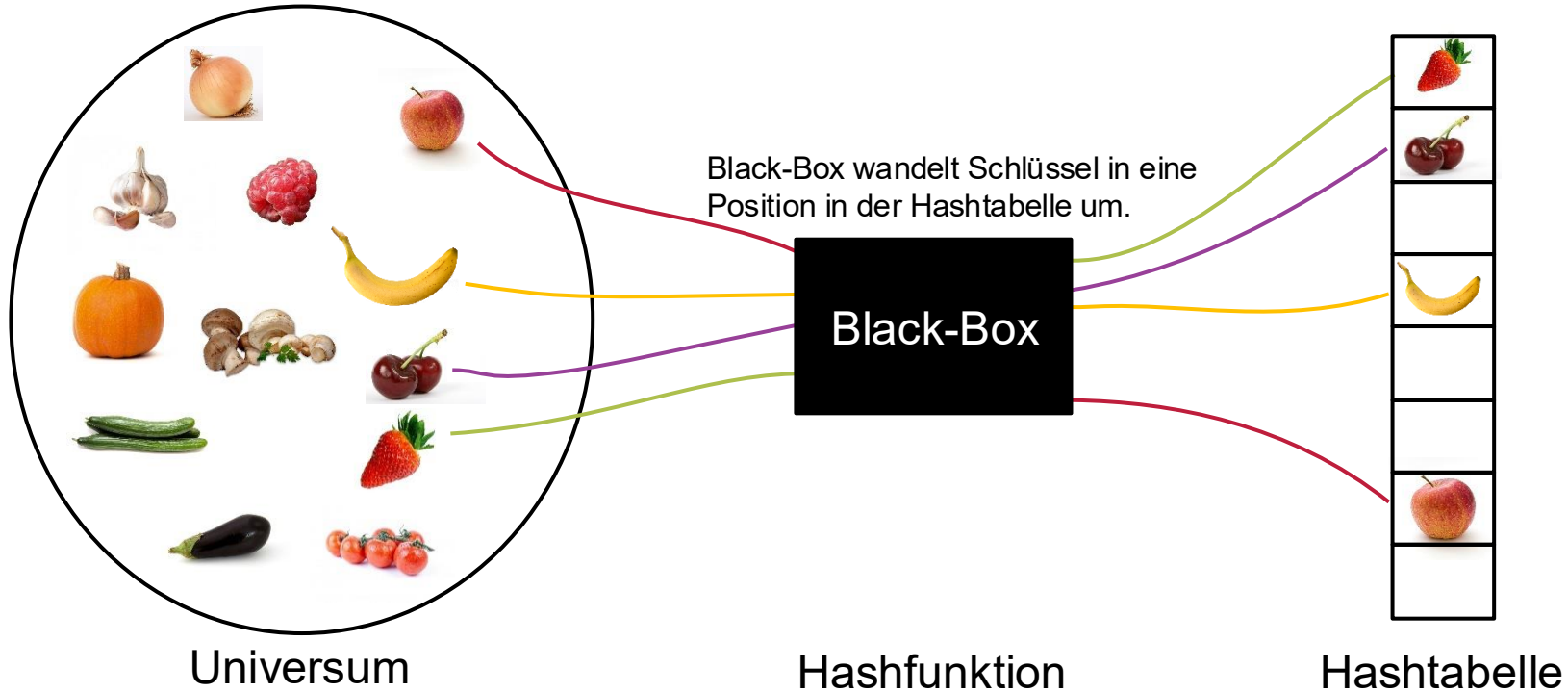
www.google.de



Hash	Wert
00	216.58.213.195
01	
02	
03	134.169.9.1
04	
05	134.169.34.49
06	

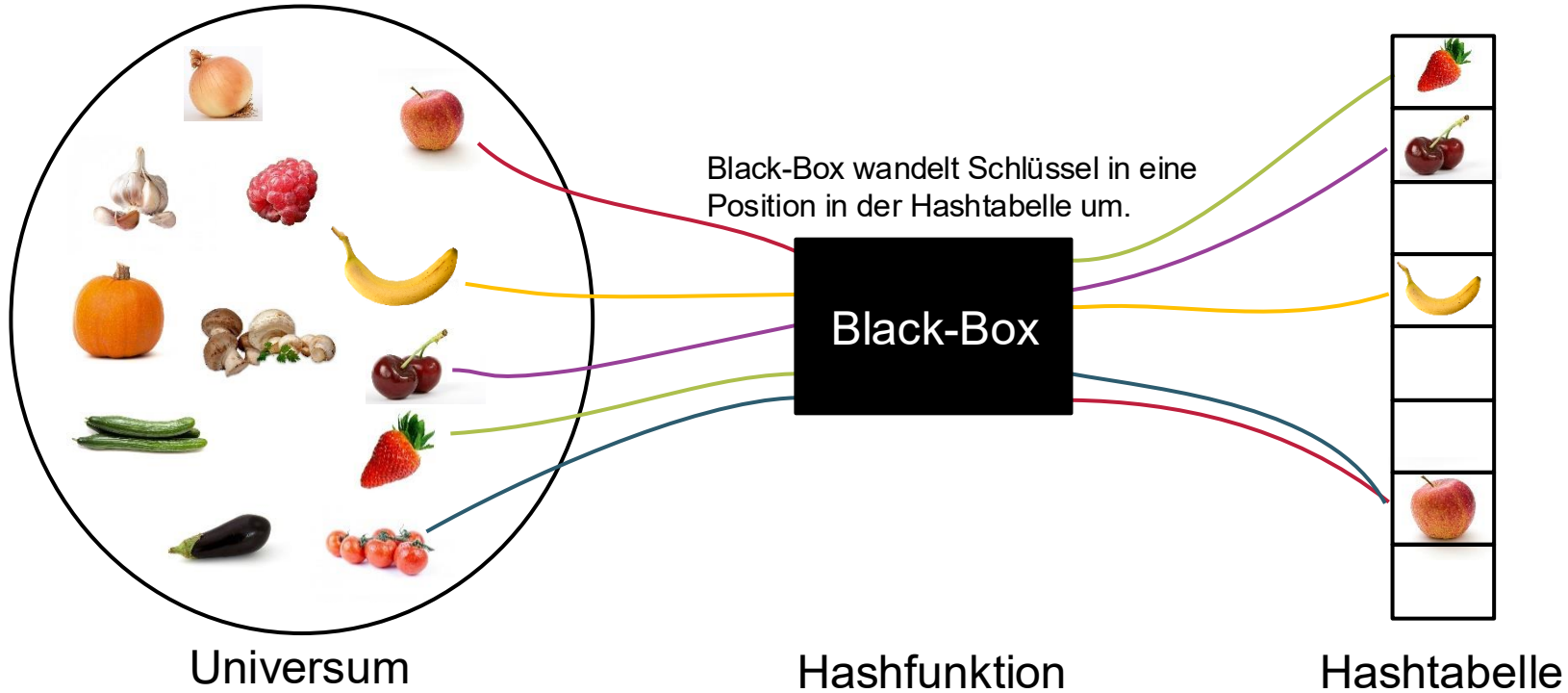
Hashing

Jedes Objekt besitzt einen Schlüssel



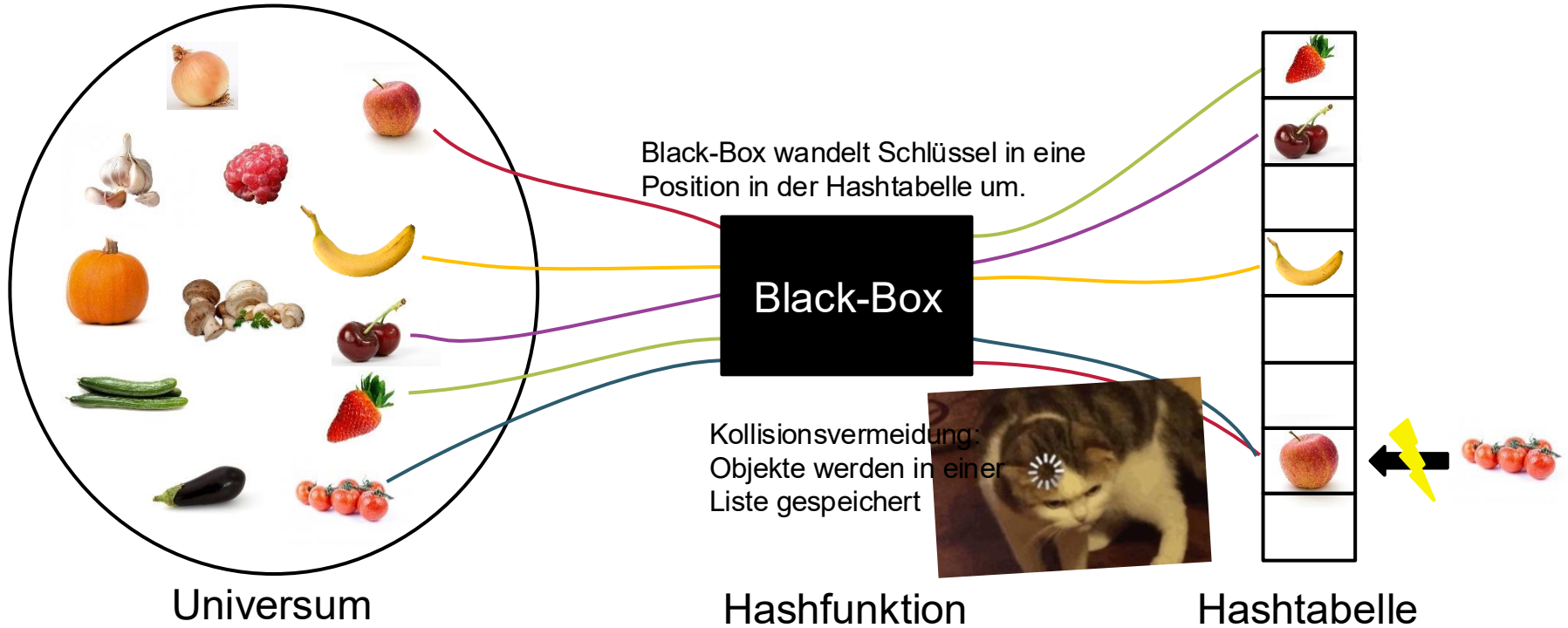
Hashing

Jedes Objekt besitzt einen Schlüssel



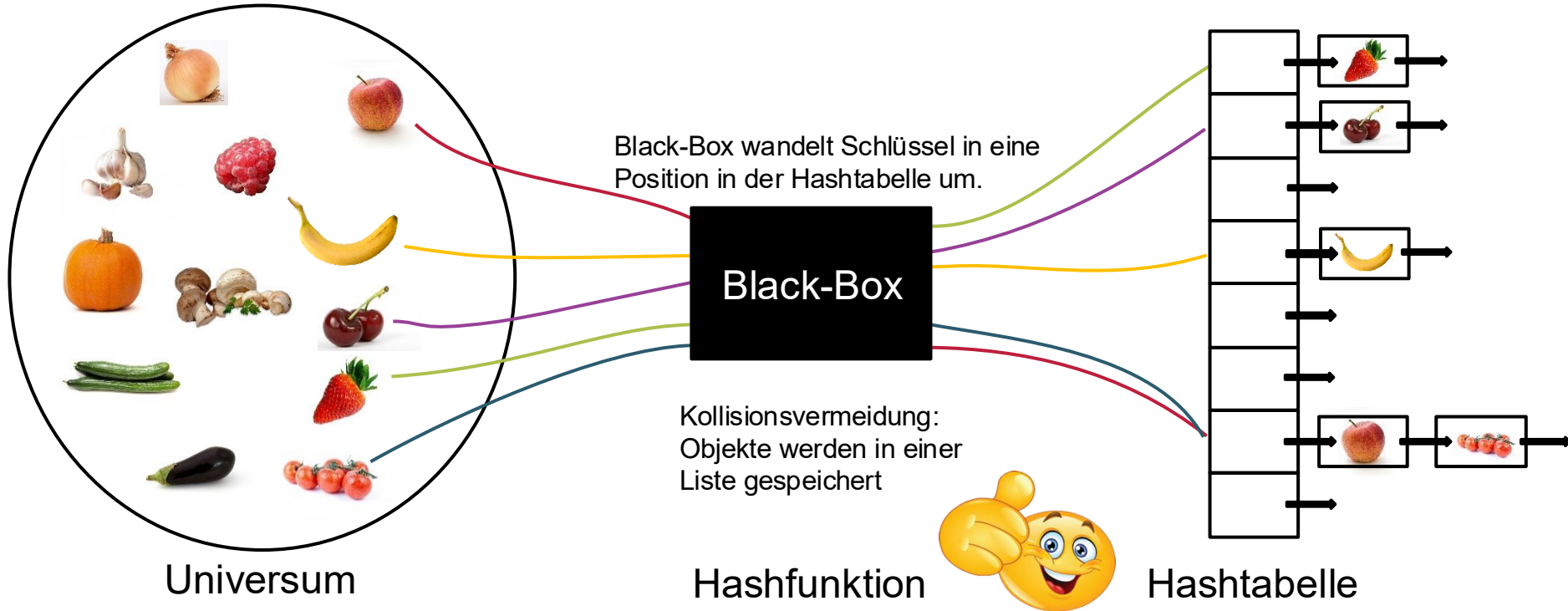
Hashing – Kollision

Jedes Objekt besitzt einen Schlüssel



Hashing – Listen

Jedes Objekt besitzt einen Schlüssel



Hashing – Listen in der Praxis

Vorteile

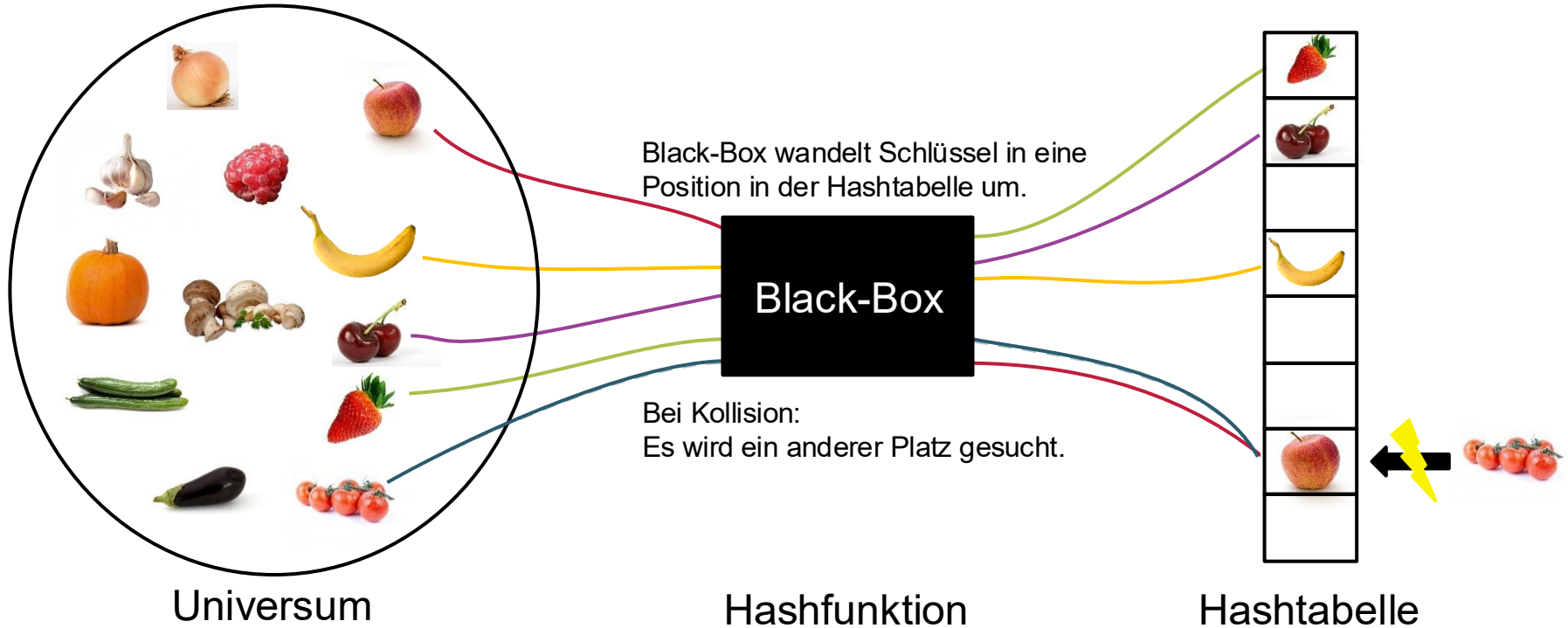
- Einfach zu implementieren
- Robust
- Stabilität der Adressen der Einträge selbst bei Rehashing (z.B. Vergrößerung der Tabelle)

Nachteile

- Zusätzliche Indirektion (siehe Pointer nach, siehe dann Element nach, ...): Laufzeitkosten
- Zusätzlicher Platzbedarf
- Zusätzliche dynamische Allokationen (kann man in den Griff kriegen)
- Iteration der Tabelle ohne Extraaufwand langsam:
 - Betrachte jeden Slot ('Bucket')
 - Dann jeweils Iteration durch verkettete Liste

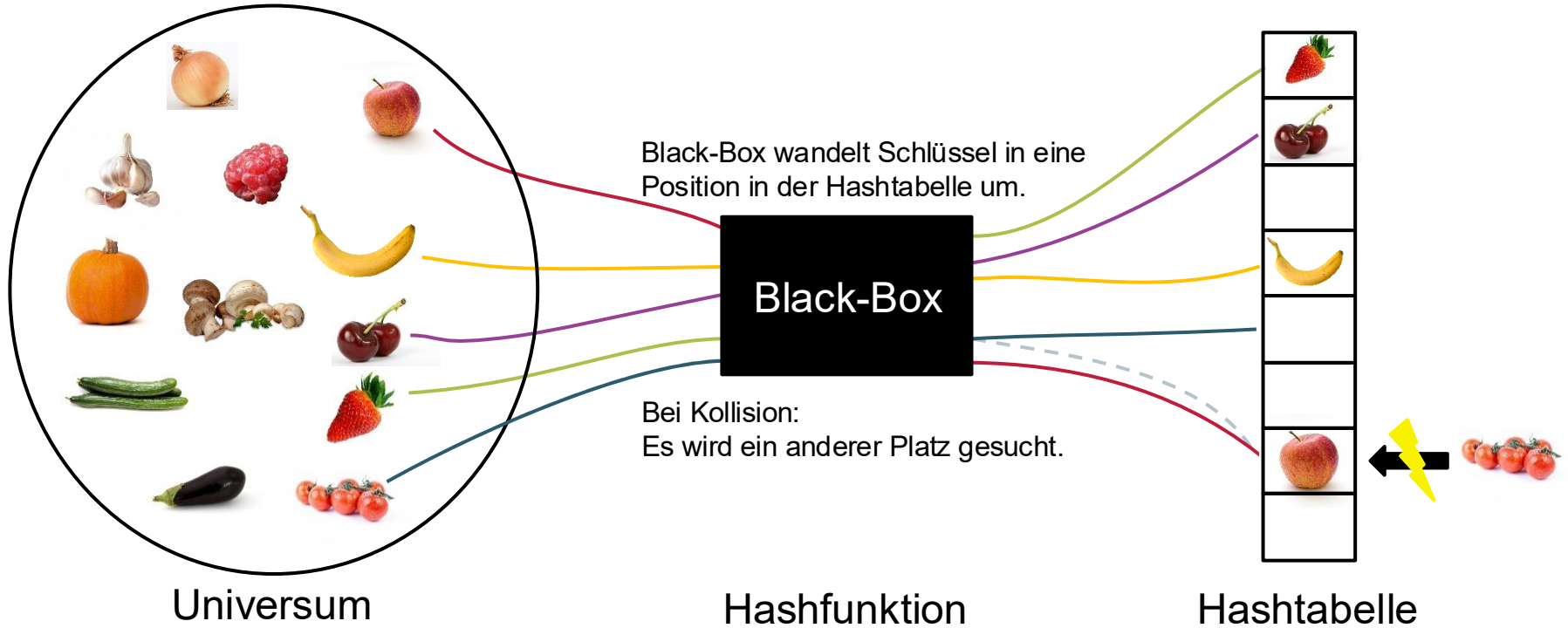
Hashing – offene Adressierung (offenes Hashing)

Jedes Objekt besitzt einen Schlüssel



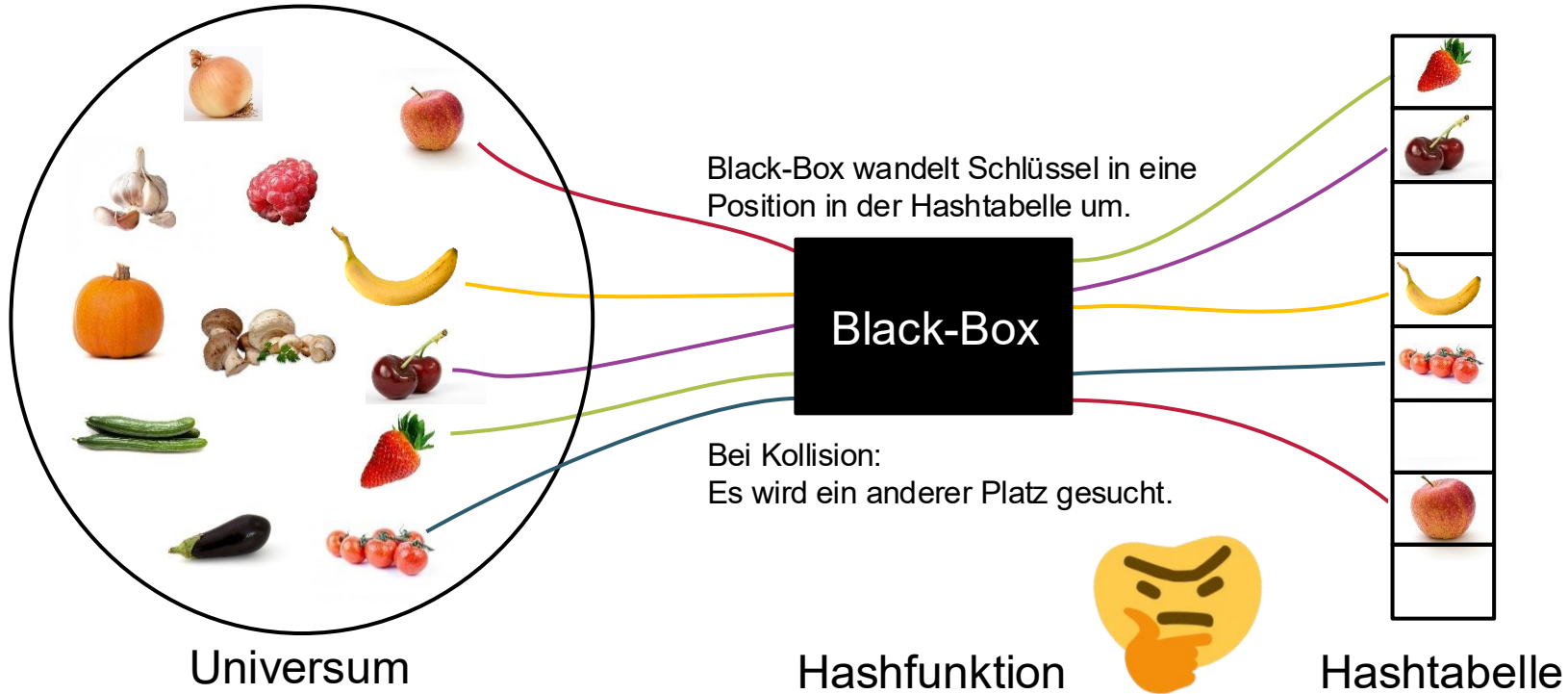
Hashing – offene Adressierung (offenes Hashing)

Jedes Objekt besitzt einen Schlüssel



Hashing – offene Adressierung (offenes Hashing)

Jedes Objekt besitzt einen Schlüssel



... aber hat das nicht Konsequenzen?

- Bei offenem Hashing ist nicht mehr eindeutig klar, wo in der Tabelle ein Schlüssel landet
- ... das heißt, alle drei Operationen
 - INSERT
 - SEARCH
 - DELETE
- ... müssen angepasst werden!

Hashfunktionen für offenes Hashing

Hashfunktion nicht mehr nur Mapping $\text{Key} \rightarrow \text{Hash}$, $x \mapsto h(x)$

Sondern: Wir müssen auch bestimmen, wie wir sondieren

Also: Wo schauen wir nach i fehlgeschlagenen Lookups?

Hashfunktion hier also jetzt Mapping $(\text{Key}, i) \rightarrow \text{Index}$, $(x, i) \mapsto t(x, i)$

Sondierungsmethoden:

- Lineare Sondierung
- Quadratische Sondierung
- Multiplikative Sondierung
- Doppeltes Hashing

In Aufgaben: Üblicherweise explizit angegeben

Für $m \in \mathbb{N}$, $x \in \mathbb{Z}$ und $r \in \{0, \dots, m-1\}$ ist

$$x \bmod m = r,$$

falls eine Zahl $q \in \mathbb{Z}$ existiert, sodass gilt

$$x = q \cdot m + r.$$

Damit ist auch für beliebiges $i \in \mathbb{Z}$

$$(x + i \cdot m) \bmod m = x \bmod m.$$

Beispiel:

$$27 \bmod 7 = 6$$

$$31 \bmod 9 = 4$$

Man kann zeigen:

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$

$$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$$

(das hilft euch in HA / Klausur)

Klassische Aufgabenstellung

Betrachte ein anfangs leeres Array A der Größe 11, es gibt also die Speicherzellen $A[0], \dots, A[10]$. In diesem Array führen wir offenes Hashing mit der folgenden Hashfunktion durch:

$$t(i, x) = 7x + (3x + 1) \cdot i \bmod 11$$

Dabei ist x ein einzusetzender Schlüssel und i die Nummer des Versuches, x in eine unbesetzte Speicherzelle des Arrays zu schreiben, beginnend bei $i = 0$. Berechne zu jedem der folgenden Schlüssel die Position, die er in A bekommt:

26, 152, 4

Klassische Aufgabenstellung

Betrachte ein anfangs leeres Array A der Größe 11, es gibt also die Speicherzellen $A[0], \dots, A[10]$. In diesem Array führen wir offenes Hashing mit der folgenden Hashfunktion durch:

$$t(i, x) = 7x + (3x + 1) \cdot i \bmod 11$$

Dabei ist x ein einzusetzender Schlüssel und i die Nummer des Versuches, x in eine unbesetzte Speicherzelle des Arrays zu schreiben, beginnend bei $i = 0$. Berechne zu jedem der folgenden Schlüssel die Position, die er in A bekommt:

26, 152, 4

- Was für ein Sondieren ist das hier?
 - Lineares Sondieren
 - Quadratisches Sondieren
 - Multiplikatives Sondieren
 - ⇒ • Doppeltes Hashing
- Welches Problem hat die zweite Hashfunktion $(3x + 1) \cdot i \bmod 11$?
 - Kann 0 sein, dann finden wir keine alternativen Positionen!
 - Hier zu einfach gehalten aus Demonstrationszwecken
 - Passt auf, falls ihr so etwas mal implementiert :)

Klassische Aufgabenstellung

$$t(i, x) = 7x + (3x + 1) \cdot i \bmod 11$$

26, 152, 4

x : Schlüssel

i : Versuch (Start bei 0)

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	A[10]
						26		152		4

INSERT 26:

- $t(0, 26) = 6$
- $7 \cdot 26 = 182$
- $182 \bmod 11 = 6$
- keine Kollision

INSERT 152:

- $t(0, 152) = 8$
- $7 \cdot 152 = 1064$
- $1064 \bmod 11 = 8$
- keine Kollision

INSERT 4:

- $t(0, 4) = 6$
- $4 \cdot 7 = 28$
- $4 \cdot 7 \bmod 11 = 6$
- Kollision!

- $t(1, 4) = 8$
- Kollision!
- $t(2, 4) = 10$
- Keine Kollision

Hashing in der Praxis

Übliche Vorgehensweise

In der Praxis: Objekte implementieren Hashfunktion

- Kennen normalerweise nicht die Größe der Tabelle
- Liefern normalerweise Hashwerte im Bereich eines Hardware-Integers (z.B. $0 \leq h(x) < 2^{64}$)
- Normalerweise findet die Reduktion Modulo m separat statt
- Dieser Teil wird dann normalerweise von der Hashtabelle implementiert
- Gleiches gilt üblicherweise für das Sondieren

Hashing in der Praxis

In Programmiersprachen:

- Java (HashMap)
 - C++ (std::unordered_map): üblicherweise mit Listen
 - C++ (absl::flat_hash_map): offen, quadratische Sondierung
 - Python (dict): offen, spezielle Sondierungsfunktion
 - JS: Objects, auch kompletter Standard {"key": "value", ...}
 - Lua: Es *gibt* nicht mal Arrays, quasi alles ist ein Dictionary
 - ...
-
- Was, wenn wir überhaupt nicht wissen, wie viele Elemente in unserer Hash-Tabelle landen werden?
 - → Rehashing (mittendrin neue Tabelle erstellen, alle Elemente neu eintragen)

Coding-Interview

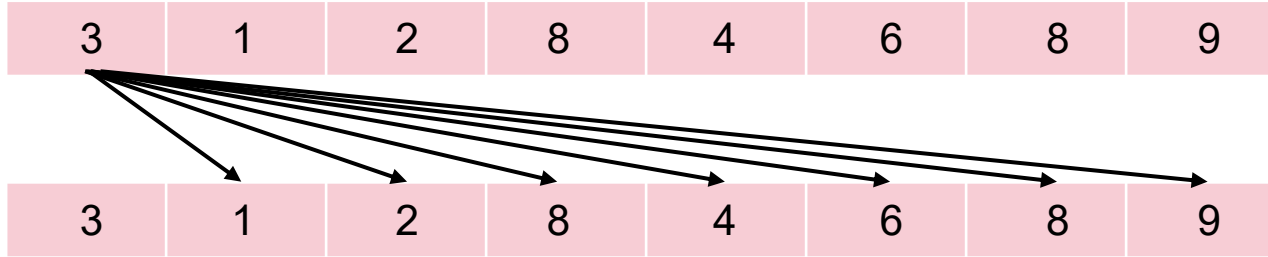
Coding-Interview

Element Uniqueness Problem:

3	1	2	8	4	6	8	9
---	---	---	---	---	---	---	---

Frage: Sind alle Zahlen unterschiedlich?

Coding-Interview



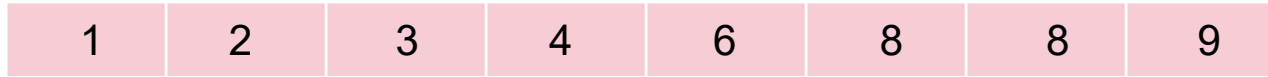
Naiver Ansatz:

```
for i in 1, ..., n:  
    for j in i + 1, ..., n:  
        if l[i] == l[j]:  
            return True  
return False
```

$O(n^2)$

Coding-Interview

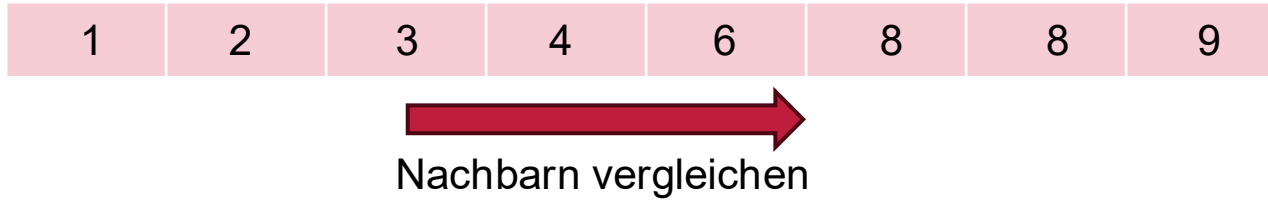
3	1	2	8	4	6	8	9
---	---	---	---	---	---	---	---



Nachbarn vergleichen

Sortieren !

Coding-Interview

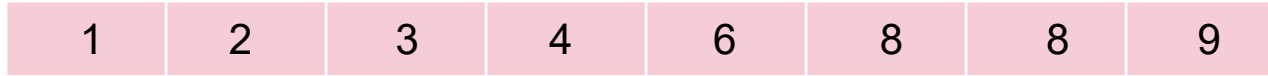


Besserer Ansatz:

```
sort(l)  
for i in 1, ..., n - 1:  
    if l[i] == l[i + 1]:  
        return True  
return False
```

$O(n \log n)$

Coding-Interview



Nachbarn vergleichen

B

Geht das noch besser?

for i in 1, ..., n - 1:

$O(n \log n)$

Ja! Hashing :)

return false

Coding-Interview

3	1	2	8	4	6	8	9
---	---	---	---	---	---	---	---



Noch besserer Ansatz:

```
h = HashMap()  
for elem in l:  
    if h.contains(elem):  
        return True  
    h.insert(elem)  
return False
```

Average Laufzeit:

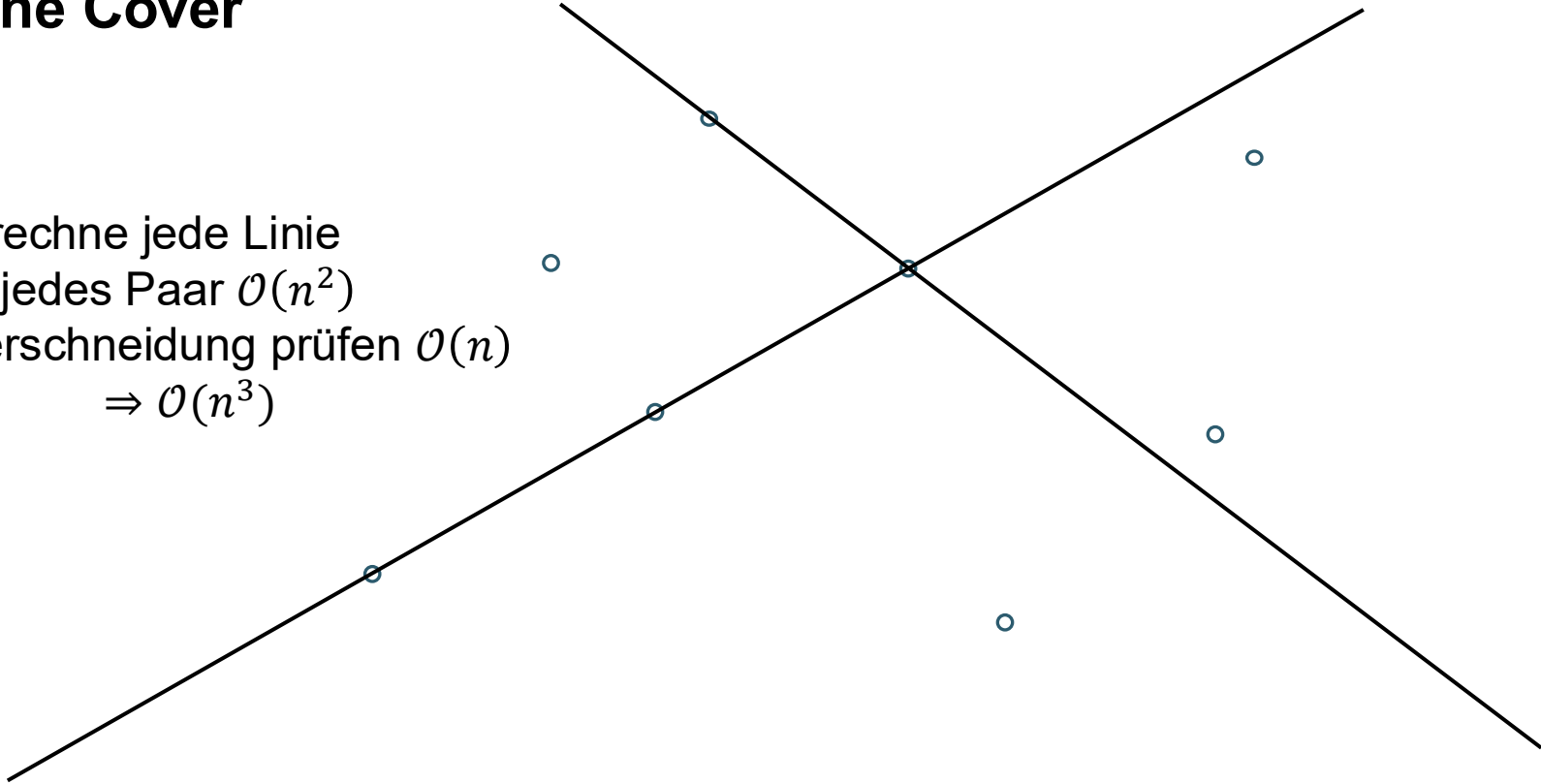
$$O(n)$$

Worst-case Laufzeit:

$$O(n^2)$$

Point Line Cover

1. Berechne jede Linie
 - Für jedes Paar $\mathcal{O}(n^2)$
 - Überschneidung prüfen $\mathcal{O}(n)$
 $\Rightarrow \mathcal{O}(n^3)$

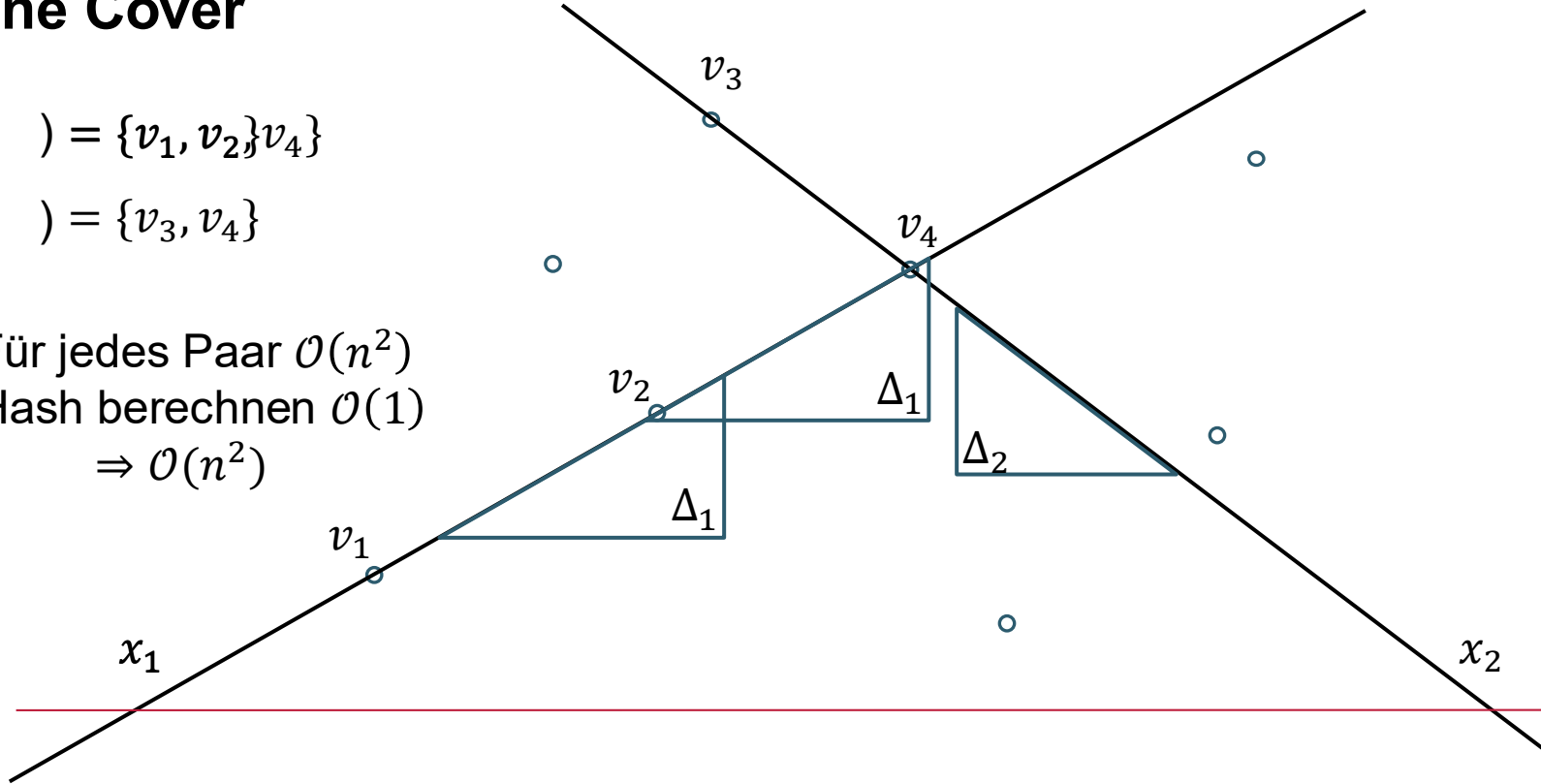


Point Line Cover

$$(\quad, \quad) = \{v_1, v_2\}v_4\}$$

$$(\quad, \quad) = \{v_3, v_4\}$$

- Für jedes Paar $\mathcal{O}(n^2)$
- Hash berechnen $\mathcal{O}(1)$
 $\Rightarrow \mathcal{O}(n^2)$



... und nächstes Mal

Semesterrückblick

Zusammenfassung Algorithmen und Datenstrukturen 2	Wiederholung I Branch & Bound
Wiederholung II Reduktion	Klausur

3SAT-3

$$(x_i \vee x_j \vee x_k) \quad (\bar{x}_i \vee \bar{x}_o \vee x_q) \quad (x_i \vee \bar{x}_p \vee x_j) \quad (x_i \vee x_p \vee \bar{x}_o)$$

Für jedes der n_i Literale von Variable x_i :

- Erzeuge Variablen $x_{1,i}, \dots, x_{n,i}$.
- Ersetze das c -te Literal von x_i mit einem Literal der Variablen $x_{c,i}$.
- Füge Klauseln der folgenden Form hinzu.

$$(x_{1,i} \vee \bar{x}_{2,i}) \wedge (x_{2,i} \vee \bar{x}_{3,i}) \wedge \dots \wedge (x_{n,i} \vee \bar{x}_{1,i})$$

Äquivalent: $(x_{2,1} \rightarrow x_{1,i}) \wedge (x_{3,i} \rightarrow x_{2,i}) \wedge \dots \wedge (x_{n,i} \rightarrow x_{1,i})$

Beobachtungen:

- Die neuen Variablen tauchen genau drei Mal auf.
- Ist eine Variable auf *true* gesetzt, sind alle *true*. (Repräsentieren die gleiche Variable!)
- Die alte Variable taucht nicht mehr auf.

Knapsackvarianten

Problem	0-1-KNAPSACK		MAXIMUM KNAPSACK		FRACTIONAL KNAPSACK	
Typ	Entscheidungsproblem		Optimierungsproblem		Optimierungsproblem	
Komplexität	NP-vollständig		NP-vollständig		P	
Algorithmen	DP	B&B	Greedy ₀	Greedy _k ($k > 0$)	DP	B&B
Bemerkung	-	-	Wert ist bel. schlecht	$1 - \frac{1}{k+1}$ -Approximation	Optimal	Optimal
Laufzeit	$O(nZ)$	$O(2^n)$	$O(n \log n)$	$O(n^{k+1})$	$O(nZ)$	$O(2^n)$

Weitere Probleme:

- SUBSET SUM: NP-vollständig, Dynamisches Programm
- PARTITION: NP-vollständig, Dynamisches Programm

Wiederholung und Klausurvorbereitung :)

Nächste Woche, am 15.07.2026!