Zusammenfassung - Kryptographie

Marc Meier

23. Oktober 2015

Korrektheit und Vollständigkeit der Informationen sind nicht gewährleistet. Macht euch eigene Notizen oder ergänzt/korrigiert meine Ausführungen!

Inhaltsverzeichnis

Planung der Veranstaltung

1. Wiederholung (ca 3-4 Termine)
   - Probleme, Kodierung von Problemen, Laufzeit von Algorithmen
   - P, PN, NPC...
   - Grundlegende Probleme & Algorithmen
2. Algorithmen für schwierige Probleme (ca. 15 Termine)
   - parallele / randomisierte Algorithmen
   - Approximationsalgorithmen
   - parametrisierte Algorithmen
3. Kryptographie (ca 10 Termine)
   - Public-Key-Kryptographie

1 Grundlagen und Wiederholungen

1.1 Probleme, deren Kodierung und Laufzeit von Algorithmen

Anhand einiger Beispiele soll die Wichtigkeit der Kodierung der Probleme bzw. der Eingabe verdeutlicht werden.

1.1.1 Sortieren

Eingabemenge: Natürliche Zahlen $a_1, a_2, ..., a_n$

Ausgabe: Eingabe aufsteigend sortiert

Beim Sortieren handelt es sich um ein *Suchproblem*. Beispiele für bekannte Sortieralgorithmen und deren Laufzeitkomplexität sind Bubblesort $\mathcal{O}(n^2)$ und Mergesort $\mathcal{O}(n \log n)$[1].

1.1.2 Eingabelänge N

Die Länge der Eingabe entspricht *nicht* der Anzahl der zu sortierenden Elemente n. Dies ist aufgrund der Kodierung in eine maschinenlesbare Form der Fall. Für gewöhnlich verwenden heutige Computer eine Darstellung im Binärsystem. Daher gilt:

$$N := \sum_{i=1}^{n} \log a_i \tag{1}$$

[1] Sofern nicht anders angegeben entspricht $\log n$ immer dem dualen Logarithmus von n (Basis 2)

Im Folgenden soll die Eingabe [11, 13, 113] kodiert werden. Für eine Turingmaschine mit dem Eingabealphabet $\Sigma := \{0, 1, \$\}$ würde diese folgendermaßen aussehen: 1011**$**1101**$**1110001. Um eine Lesbarkeit für Binärrechner zu ermöglichen, werden weiterhin folgende Ersetzungen durchgeführt: $1 \rightarrow 11$ $0 \rightarrow 00$ $\$ \rightarrow 01$ Die resultierende Kodierung wäre nun: 11001111**01**11110011**01**11111100000011. Sei nun L die maximale Wortlänge, d.h. die größte Binärdarstellung der Eingabezahlen a_i.

$$L := max_{i=1}^{n} \log a_i \tag{2}$$

Daraus folgt für Bubblesort:

$$\begin{aligned}
\mathcal{O}(n^2) &= \mathcal{O}(L \cdot n^2) && \mid \text{Algo. } n\text{-mal durchlaufen mit max. Eingabelänge } L \text{ als obere Abschätzung} \\
&= \mathcal{O}(N \cdot N^2) && \mid L \leq N (\text{offensichtlich wegen } N := \sum_{i=1}^{n} \log a_i) \\
&= \mathcal{O}(N^3) && \mid n \leq N^2
\end{aligned}$$

1.1.3 Primzahl

Eingabemenge: Eine natürliche Zahl n

Ausgabe: Ist n eine Primzahl?

Es handelt sich um ein *Entscheidungsproblem*. Die Eingabelänge ist $N := \log n$. Ein naiver Algorithmus wird im Folgenden beschrieben.

```
1: if n = 2 then
2:     return Ja
3: else
4:     for d := 2 to n − 1 do
5:         if n mod d = 0 then
6:             return Nein
7:         else
8:         end if
9:     end for
10:    return Ja
11: end if
```

Die Komplexität lässt sich wie folgt herleiten: Für die Verzweigungen (Zeilen 1 und 5), sowie für die return-Statements in den Zeilen 2 und 6 ist eine beschränkte Komplexität $\mathcal{O}(1)$ anzunehmen. Alle Statements in der for-Schleife werden $n - 2$ mal wiederholt, die Komplexität ist $\mathcal{O}(n)$.

Die daraus resultierende Komplexität $\mathcal{O}(n)$ ist äquivalent zu $\mathcal{O}(2^{\log n})$. Weil $\log n$ der Eingabelänge N entspricht, ist die Komplexität entsprechend $\mathcal{O}(2^N)$, also exponentiell. Demnach ist der Algorithmus nicht effizient.

1.1.4 Cliquensuche im Graphen

CLIQUE (gehört zu den Entscheidungsproblemen)

Eingabemenge: Graph $G = (V, E)$ und $k \in N$

Ausgabe: Hat G mindestens paarweise verbundene Knoten (eine Clique C mit $\geq k$ Knoten)?

CLIQUE (Suchproblem) (gehört zu den Optimierungsproblemen)

Eingabemenge: Graph $G = (V, E)$

Ausgabe: Bestimme eine Clique C in G mit maximaler Knotenzahl.

Behauptung: Wenn CLIQUE (Entscheidungsproblem) in polynomieller Zeit lösbar ist, so ist auch die Suchversion von CLIQUE in polynomieller Zeit lösbar.[3]

Beweis

[3] In polynomieller Zeit, abhängig von der Knotenzahl $|V|$ des Graphen

$\Leftarrow$ Zur Eingabe (G,k) von CLIQUE bestimmen wir zunächst mit dem Algorithmus für CLIQUE-Suchversion eine größte Clique C von G. Es wird verglichen, ob $|C| \geq k$ gilt. Ist dies der Fall, gibt der Algorithmus *Ja* zurück, andernfalls *Nein*. Die Laufzeit entspricht offensichtlich der Clique-Suchversion.

$\Rightarrow$ Betrachte folgende Zwischenversion für CLIQUE: Betrachte folgende Zwischenversion für CLIQUE[4]:

> **CLIQUE-Zwischenversion:**
>
> **Eingabe:** Graph G=(V,E)
>
> **Aufgabe:** Bestimme die Maximalzahl $\omega(G)$ von paarweise verbundenen Knoten in G

Behauptung: CLIQUE in polynomieller Zeit lösbar $\Rightarrow^{(1)}$ CLIQUE-Zwischenversion in polynomieller Zeit $\Rightarrow^{(2)}$ CLIQUE-Suchversion in polynomieller Zeit lösbar.

zu (1): Sei A ein Algorithmus für CLIQUE in polynomieller Zeit. Zur Eingabe G von CLIQUE-Zwischenversion wende A auf Eingaben $(G, n), (G, n-1), ..., (G, 1)$ an! Ausgabe ist $\omega(G) =$ erstes k mit $A(G, k) = Ja$

for k:= n to 1 **do**
    if A **then**(G,k) = 'ja' then return $\omega(G) = k$
    end if
end for

Laufzeit: $n \cdot \mathcal{O}(n^c) = \mathcal{O}(n^{c+1})$

zu (2): : Sei A ein polyzeit Algorithmus für Clique-Zwischenversion. d.h. $A(G) = \omega(G)$ in $\mathcal{O}(n^d)$ für eine Konstante d.

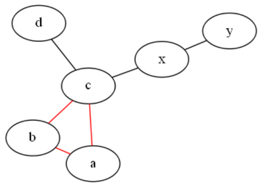
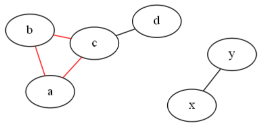
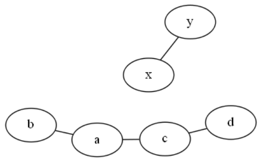
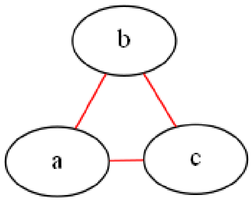
Zur Eingabe G von Clique-Suchversion wende folgendes an:

for jede Kante $e \in E$ **do**
    if $\omega(G - e) = \omega(G)$ **then** $G := G - e$
    end if
end for
return C := V(G) ohne isolierte Knoten

Hinweis: Ich bin mir nicht sicher, ob es sich um das korrekte Beispiel aus der Vorlesung handelt.

Laufzeit: $m \cdot \mathcal{O}(n^d) = \mathcal{O}(n^{d+2})$

[4] $\omega(G) = max\{|C| : C$ ist eine Clique in $G\}$ ist die Cliquenzahl

G	$\omega(G) = 3$		Vollständiger Graph
$G_2 := G - cx$	$\omega(G) = 3$		Nach Entfernen der Kante cx hat sich die Cliquenzahl nicht geändert. Die Kante ex ist also nicht für die Clique notwendig.
$G_3 := G_2 - bc$	$\omega(G) = 2$		Nach Entfernen der Kante bc hat sich die Cliquenzahl verringert. Offensichtlich wird bc also für die Clique benötigt.
$C := \{a, b, c\}$	$\omega(G) = 3$		Im weiteren Verlauf wird (offensichtlich) noch die Kante cd entfernt. Anschließend werden die isolierten Knoten d, x und y entfernt und die Lösung $C := \{a, b, c\}$ zurückgegeben.

1.1.5 Gegenüberstellung „einfache Formulierung“ vs „formale Formulierung“

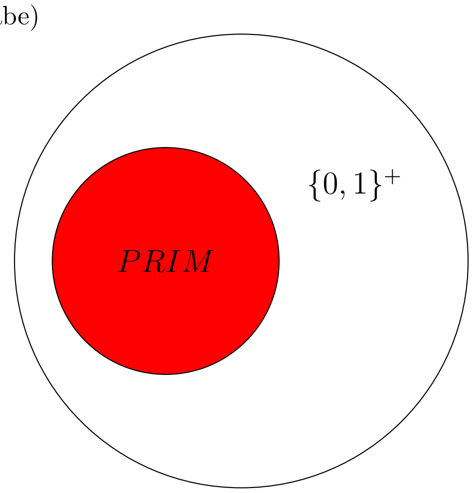
Die genaue Formulierung des Problems ist wichtig:
„Ist n prim?“
genauer:

Eingabemenge: Eine natürliche Zahl n

Ausgabe: Ist n eine Primzahl?

Formal: $PRIM$[5] $= \{bin(n) | n \in N \text{ ist prim}\} \subset \{0,1\}^+$

Venn-Diagramm $PRIM \subset \{0,1\}^+$ TODO (Und offensichtlich falsch, da ich noch keinen Plan von TIKZ habe)



[5] bin(n) - Binärdarstellung von n

1.2 $\mathbb{P}$, $\mathbb{NP}$, $\mathbb{NP}$-vollständig

$\mathbb{P}$ = Menge aller *Entscheidungsprobleme*, für die ein (deterministischer) polynomialzeit-Algorithmus existiert (formal via Deterministischer Turing Maschine, „effizient lösbare Probleme"). Ein *Optimierungsproblem* ist lösbar, wenn die entsprechende Entscheidungsversion in P ist.

1.2.1 Grundlegende Probleme in $\mathbb{P}$

- **Sortieren**
- **PRIM** - Allerdings nicht der vorgestellte Algorithmus. Effizienter Algorithmus wurde 2002 vorgestellt. Komplexität $\mathcal{O}(\log n)^{12}$), zur Zeit sogar $\mathcal{O}(\log n)^{6}$)`https://de.wikipedia.org/wiki/AKS-Primzahltest`
- **2-SAT** (Modellierung mit Graphenproblem für Beweis)

> **Übrige Notizen**
> In der Praxis ermittelt man die Komplexität der Einfachheit halber mit uniformen Kostenmaß. Sortierung: Eingabe: $a_1, a_2, ..., a_n$ beziehungsweise $bin(a_1)\#bin(a_2)\#...\#bin(a_n)$ mit Ersetzung Aufgabe: Sortiere Zahlen aufsteigend Mergesort: $\mathcal{O}(n \cdot \log n)$ Problem -¿ Eingabelänge (siehe oben, hatten wir schon)

1.3 $\mathbb{NP}$-Vollständigkeit Reduktionsbeispiele

3-SAT $\leq_p$ INDSET
"$\Rightarrow$":
Eingabe: Sei $F = K_1 \wedge K_2 \wedge \ldots \wedge K_m$ und $K_j = (l_{j,1} \vee l_{j,2} \vee l_{j,3})$, wobei $l_{j,1}, l_{j,2}, l_{j,3} \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_3}\}$.

Erklärung der Notation am Beispiel:

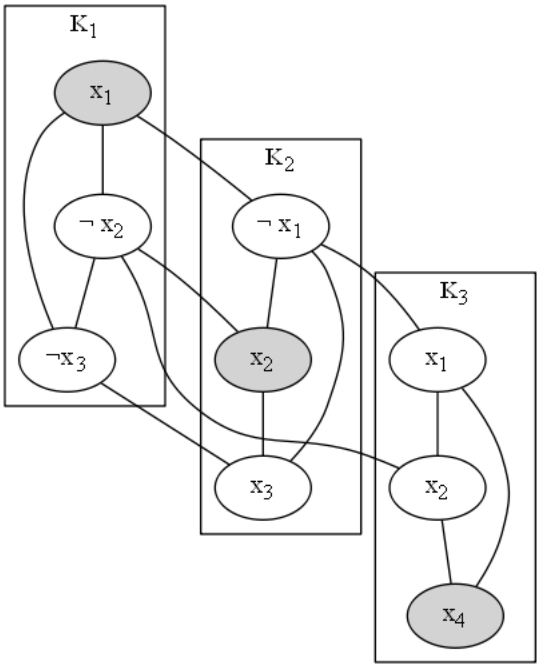
$$F = \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{K_1} \wedge \underbrace{(\overline{x_1} \vee x_2 \vee x_3)}_{K_2} \wedge \underbrace{(x_1 \vee x_2 \vee x_4)}_{K_3}$$
$$K_1 : l_{1,1} = x_1; l_{1,2} = \overline{x_2}, \text{usw.}$$

Der Beweis erfolgt in zwei Schritten:

1. Konstruiere einen Graphen G und eine natürliche Zahl k aus der Formel F.
2. Zeige das der Graph eine unabhängige Knotenmenge U mit $|U| = k$ hat.

Konstruiere den Graphen G wie folgt:

- Literale entsprechen Knoten
- Verbinde die drei Literale innerhalb einer Klausel K_j miteinander
- Verbinde die Knoten unterschiedlicher Klauseln, wenn die entsprechenden Literale zueinander komplementär sind (Bsp: x_1 mit $\overline{x_1}$)



Beispiel Formel F: Verbinden Knoten der Knoten

- Setze k wie folgt: $k = m$ (= die Anzahl der Klauseln in F)

Annahme: Sei b eine erfüllende Belegung für F, d.h. in jeder Klausel K_j existiert ein Literal $l_{j,k}$ mit $b(l_{j,k}) = 1$. Wähle in Graph G aus jeder Klauselmenge genau einen Knoten mit entsprechendem wahren Literal. Aus der Konstruktion von G folgt, das diese k Knoten unverbunden sind und eine unabhängige Menge bilden.
Am Beispiel: $b(x_1) = b(x_2) = 1; b(x_3) = b(x_4)$. Daraus folgen die ausgewählten Knoten $x_1, x_2 \text{und} \overline{x_3}$. Diese ergeben eine unabhängige Menge.

"$\Leftarrow$":
Annahme: Sei U eine unabhängige Menge von G mit $|U| \geq k = m$ Knoten. Dann gilt $|U| = k$, weil jede der m Klauselmengen ein Dreieck bildet, d.h. $|UK_j| = 1$, für jede Klauselmenge K_j. U definiert nun eine Belegung für F wie folgt:

$$\begin{aligned} b(l_{j,k}) &= 1 \quad \text{,falls } l_{j,k} \in U \\ &= 0 \quad \text{,sonst} \end{aligned}$$

$b(l_{j,k}) = 1 \Rightarrow b(\overline{l_{j,k}}) \notin U \Rightarrow b(\overline{l_{j,k}}) = 0$
$\Rightarrow$ F ist erfüllbar unter b!

Beispiel Belegung: $b(x_1) = b(x_2) = b(x_3) = 0; b(x_4) = 1$

3-SAT $\leq_p$ 3-Färbung
Eingabe: Sei $F = K_1 \wedge K_2 \wedge \ldots \wedge K_m$ und $K_j = (l_{j,1} \vee l_{j,2} \vee l_{j,3})$, wobei $l_{j,1}, l_{j,2}, l_{j,3} \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_3}\}$.

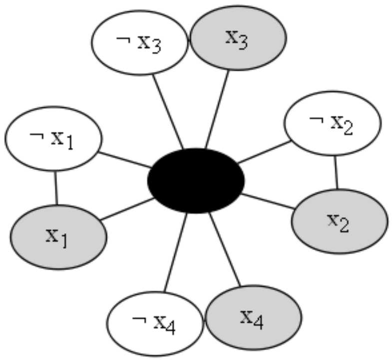
Beispiel:

$$F = \underbrace{(x_1 \vee \overline{x_2} \vee \overline{x_3})}_{K_1} \wedge \underbrace{(\overline{x_1} \vee x_2 \vee x_3)}_{K_2} \wedge \underbrace{(x_1 \vee x_2 \vee x_4)}_{K_3}$$

"$\Rightarrow$": Graph G ist 3-färbbar $\Rightarrow$ F ist erfüllbar.

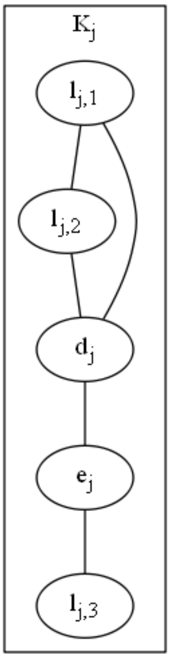
Konstruiere den Graphen G wie folgt:

- Literale entsprechen Knoten und verbinde komplementäre Literale miteinander. Idee: Füge einen Hilfsknoten R ein (im Bild schwarz) und färbe diesen, sodass alle Literale x_i und $\overline{x_i}$ mit genau zwei anderen Farben gefärbt werden müssen.



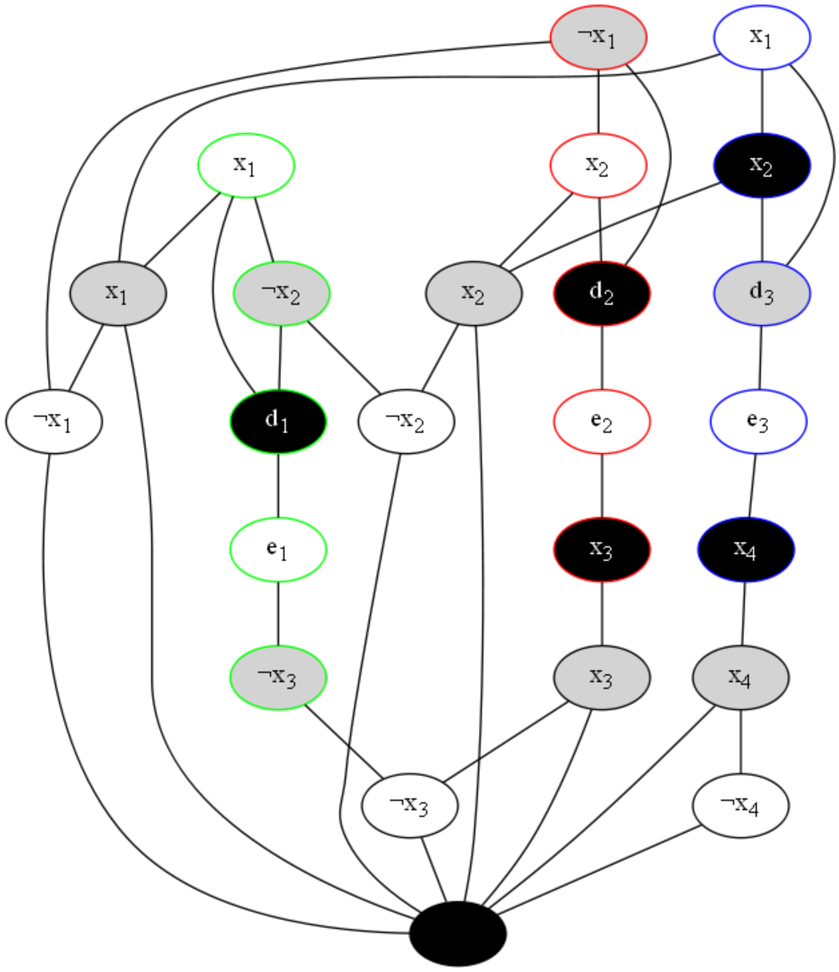
Einfügen des Hilfsknoten R

- Für jede Klausel K_j konstruiere den folgenden Teilgraphen:



Teilgraph für Klausel K_j In meinen Aufzeichnungen werden die l Knoten durch a, b und c (=l_3) ersetzt?!

- Für $K_j = \{l_{j,1}, l_{j,2}, l_{j,3}\}$ füge die folgenden Kanten ein: $al_{j,1}, bl_{j,2}, cl_{j,3}$.
- Füge nun zwei zusätzliche Knoten W und F ein, sowie die Kanten RW, RF, FW. Zusätzlich die Kanten Fd_j, Fe_j für alle j.



Beispielgraph, konstruiert aus Formel F

"$\Leftarrow$": Sei F eine erfüllbare Formel (unter 3-SAT) $\Rightarrow$ Graph G ist 3-färbbar.

Sei b eine erfüllende Belegung für F. Färbe nun alle Knoten x_i mit $b(x_i)$ und alle Knoten $\overline{x_i}$ mit $b(\overline{x_i})$. Nun erweitern wir den Graphen auf 3 Farben indem wir zunächst die Knoten aller Klauseln K_j färben. Abschließend werden F, W und R gefärbt. Dies ergibt eine 3-Färbung des Graphen.