Algorithmik für Schwere Probleme

thgoebel@ethz.ch

ETH Zürich, FS 2021

This documents is a **short** summary for the course *Algorithmik für Schwere Probleme* at ETH Zurich. It is intended as a document for quick lookup, e.g. during revision, and as such does not replace attending the lecture, reading the slides or reading a proper book.

We do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. Feel free to point out any erratas.

Contents

1 Einführung **3**
1.1 Definitionen 4

2 Pseudopolynomielle Algorithmen **6**

3 Parametrisierte Algorithmen **7**
3.1 Kernbildung 7
3.2 Suchbäume 7
3.3 Iterative Kompression 7

1 Einführung

Polynomzeit-Reduzierbarkeit Ein Entscheidungsproblem Π_1 ist "polynomzeit-reduzierbar" auf ein anderes Entscheidungsproblem Π_2:

$$\begin{aligned}
&\iff \exists \text{ Algo } \mathcal{A} \text{ s.t. } \text{time}_{\mathcal{A}} \in \text{poly} \land \Pi_1(x) = \Pi_2(\mathcal{A}(x)) \\
&\iff \Pi_2 \text{ mindestens so schwer wie } \Pi_1 \\
&\iff \Pi_1 \text{ höchstens so schwer wie } \Pi_2 \\
&\iff \Pi_1 \preceq_P \Pi_2
\end{aligned}$$

NP-schwer (NP-hard) Ein Problem Π das "mindestens so schwer" ist wie alle Probleme in NP. D.h. alle Probleme in NP lassen sich auf Π reduzieren:

$$\forall \Pi' \in NP : \Pi' \preceq_P \Pi$$

Π muss nicht notwendigerweise in NP liegen (d.h. kann schwerer sein)! Beispiel: das Halteproblem (nicht entscheidbar, daher $\notin NP$).

NP-vollständig (NP-complete) Ein Problem Π, das in NP liegt und NP-schwer ist. "Repräsentativ" für die Menge NP, da sich alle Probleme aus NP darauf reduzieren lassen.
Beispiel: Satisfiability-Problem SAT (Satz von Cook).

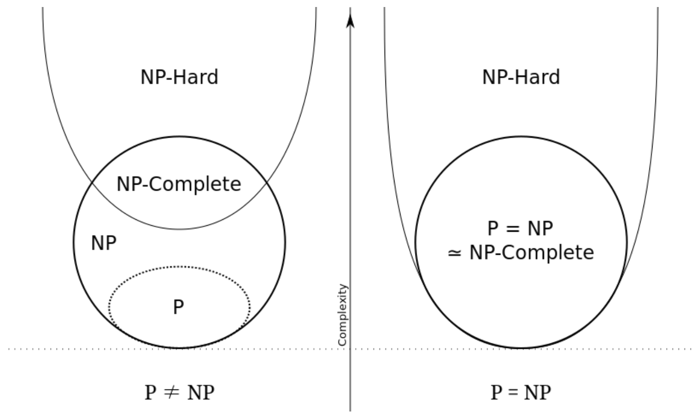

Figure 1: Mengendiagramm der Beziehungen (Quelle: Wikipedia)

"Schwere Probleme" In dieser Vorlesung: NP-schwere Probleme, aber generell alle Probleme die sich nicht in Polynomzeit lösen lassen. Sinnvollerweise gehen wir davon aus dass $P \neq NP$.

Alle Instanzen unseres Problems sind deterministisch in Polynomzeit nicht lösbar. Mögliche Ansätze:

a) nicht exakt sondern approximativ lösen (Approximationsalgorithmen)

b) nicht deterministisch sondern nichtdeterministisch lösen (Randomisierte Algorithmen)

c) nicht polynomiell sondern moderat exponentiell lösen[1]

d) nicht alle sondern alle Instanzen mit einer bestimmten Struktur lösen (Parametrisierte Algorithmen)

[1] D.h. die Basis der Exponentation ist klein, z.B. 1.4^n statt 2^n.

e) anderweitig zusätzliche Informationen über die Eingabe nutzen (Reoptimierung, Win-Win-Strategy)

f) Heuristiken[2]

1.1 Definitionen

Entscheidungsproblem $P = (L, U, \Sigma)$ wobei

- Σ ein Alphabet
- $U \subseteq \Sigma^*$ die Menge der zulässigen Eingaben (als Wörter über dem Alphabet, als *Sprache*)
- $L \subseteq U$ die Menge der akzeptierten Eingaben (*JA-Instanzen*)

Ein Algorithmus $\mathcal{A}$ *löst* P falls gilt:

$$\forall u \in U : A(x) = \begin{cases} 1 \text{ oder JA}, & \text{if } x \in L \\ 0 \text{ oder NEIN}, & \text{if } x \in U - L \end{cases}$$

Vertex Cover Problem VC "Der Hefepilz der parametrisierten Algorithmiker – ein Modellproblem."
Eingabe U: ungerichteter Graph $G = (V, E)$ und $k \leq |V|, k \in \mathbb{N}$.
Ausgabe L: JA falls $\exists C \subseteq V$ s.t. $|C| \leq k$ mit $\forall\{u, v\} \in E : u \in C \vee v \in V$.

Satisfiability-Problem SAT
Eingabe: CNF-Formel $\Phi = C_1 \wedge \cdots \wedge C_m$ mit Klauseln C_i über Variablen $x_1, \ldots, x_n$.
Ausgabe: eine Variablen-Belegung die Φ erfüllt.

Bei l-SAT enthält jede Klausel maximal l Literale.

Optimierungsproblem $U = (L, M, cost, goal)$ wobei

- L die Sprache der zulässigen Eingaben[3]
- $\mathcal{M} : L \mapsto \Sigma^*$ so dass $M(x)$ die Sprache der akzeptierten Lösungen für Eingabe x
- *cost*: $\forall x \in L \; \forall y \in M(x) : cost(y, x) =$ Kosten der Lösung y für Eingabe x
- $goal \in \{\min, \max\}$ das Optimierungsziel
- $Opt_U(x) = goal\{cost(y, x) | y \in M(x)\}$ die Kosten einer optimalen Lösung für Eingabe x

Minimum Vertex Cover Problem MIN-VC Wie VC, mit $cost(C, G) = |C| =$ Grösse des Vertex Covers und $goal = \min$.

MAX-SAT Wie SAT, mit $cost =$ Anzahl belegte Variablen und $goal = \max$.

Laufzeit eines Algorithmus' $\mathcal{A}$ auf Eingabe x ist $\text{time}_{\mathcal{A}}(x)$ wobei $\text{time}_{\mathcal{A}} : \mathbb{N} \mapsto \mathbb{N}$. Die Laufzeit von $\mathcal{A}$ in Abhängigkeit von der Grösse n der Eingabe ist: $\text{time}_{\mathcal{A}}(n) = \max\{\text{time}_{\mathcal{A}}(x) \mid |x| = n, x \in L\}$. Die Laufzeit wird in $\mathcal{O}$-Notation angegeben.

[2] Nachteil: Im Gegensatz zu den anderen Ansätzen ist hier die Qualität (Laufzeit, ...) nicht beweisbar.
[3] Oben noch U!

Schwellwertsprache (threshold language) definiert für ein Optimierungsproblem U:

$$Lang_U = \{(x,a) \in L \times \{0,1\}^* \mid Opt_U(x) \leq Number(a)\}$$

wo $Number(a)$ die Zahl mit der Binärdarstellung a (der *Schwellwert*) ist und $goal = \min$.
Beispiel: $Lang_{MIN-VC} = VC$. Aber $Lang_{MAX-SAT} \neq SAT$ (da SAT leichter sein kann)!

U heisst "NP-schwer" falls $Lang_U$ NP-schwer ist (warum?).[4]

[4]Recall that NP-Schwere für Entscheidungsprobleme definiert ist.

2 Pseudopolynomielle Algorithmen

Zahlproblem (integer value problem IVP)
Eingabe: darstellbar als Zahl $x = x_1\#\ldots\#x_n$; $x_i \in \{0,1\}^*$ und interpretiert als Vektor $Int(x) = (Number(x_i), \ldots, Number(x_n))$.
Beispiel: Travelling Salesman Problem TSP (via Adjazenzmatrix des Graphen).

Sei $Max-Int(x) = \max\{Number(x_i)\}$ die grösste vorkommende Zahl (im Wert, nicht in der Darstellung). $Max - Int(x)$ kann exponentiell in $|x|$ sein.

Pseudopolynomiell Sei U ein Zahlproblem und $\mathcal{A}$ ein Algorithmus der U löst. $\mathcal{A}$ heisst *pseudopolynomiell* falls für alle Eingaben x ein Polynom p existiert, so dass

$$\text{time}_{\mathcal{A}}(x) \in \mathcal{O}\Big(p\big(|x|, Max - Int(x)\big)\Big)$$

D.h. auf Eingaben mit "kleinen Zahlen" ist $\mathcal{A}$ polynomiell.

Rucksackproblem (Knapsack problem KP)
Eingabe I: Gewichte $w_i \in \mathbb{N}^+$, Kosten/Nutzen $c_i \in \mathbb{N}^+$, Limit/Kapazität $b \in \mathbb{N}^+$, wo $i \in \{1, \ldots, n\}$.
Ausgabe: Indexmenge $T \subseteq \{1, \ldots, n\}$ s.t. $\sum_{i \in T} \leq b$
Kosten: $cost(T, I) = \sum_{i \in T} c_i$
Ziel: max

Lösung mit DP: Iteration über alle Teilprobleme I_i und Speichern von Tripeln $(k, W_{i,k}, T_{i,k}) =$ (Nutzen, Gewicht, Indexmenge), also Mengen $T_{i,k}$ die exakt Nutzen k mit minimalen Gewicht $W_{i,k}$ erreichen. In jeder Iteration behalte für jeden vorhandenen Nutzen ein Tripel mit minimalen Gewicht. Lese am Ende den maximal erreichten Nutzen k^* (und sein T_{n,k^*}) aus.
Laufzeit: $\mathcal{O}\big(|I|^2 \cdot Max - Int(I)\big)$ da $|I|$ Iterationen und jeder Schritt in $\leq \sum_j^n c_j = |I| \cdot Max - Int(I)$.

3 Parametrisierte Algorithmen

3.1 Kernbildung

3.2 Suchbäume

3.3 Iterative Kompression