

Die Klassen P und NP

Anfang der 70er Jahre:

- Erfolg in der Lösung wichtiger algorithmischer Probleme.
- Aber viele Probleme „widersetzen“ sich:
 - ▶ Überraschende Erkenntnis, viele dieser Probleme, die **NP-vollständigen Probleme**, sind alle gleich schwierig: Kann eines effizient gelöst werden, dann können alle effizient gelöst werden.
 - ▶ Alle diese Probleme besitzen (relativ) kurze, schnell verifizierbare Lösungen.
 - ▶ Weiterhin: Wenn irgendein NP-vollständiges Probleme effizient gelöst werden kann, dann können Rechner effizient **raten**.

Wir erhalten sehr starke Indizien, dass kein einziges NP-vollständiges Problem effizient lösbar ist.

Wie löse ich ein algorithmisches Problem?

- **Erster Schritt:** Ist das Problem zu komplex, dann
 - ▶ hoffe, dass deine Eingaben „einfach sind“, bzw. füge zusätzliches Wissen über die Eingabe der Problemstellung hinzu
 - ▶ oder schwäche die Problemstellung ab. Zum Beispiel, statt ein Optimierungsproblem exakt zu lösen, suche nach guten, approximativen Lösungen.
 - ▶ Oder wirf alle zur Verfügung stehenden Rechnerkapazitäten auf das Problem.
- **Zweiter Schritt:** Wenn die Komplexität des Problems geklärt ist, dann wende die fundamentalen Entwurfsmethoden an:
 - ▶ Greedy Algorithmen, Divide & Conquer, Dynamische Programmierung
 - ▶ für schwierige Probleme (Teil III der Vorlesung), Heuristiken, Approximationsalgorithmen, die lineare Programmierung, lokale Suche, Metropolis und Simulated Annealing, evolutionäre Algorithmen, Backtracking und Branch & Bound.

Schwierige Probleme

● Das Erfüllbarkeitsproblem:

- ▶ Für eine aussagenlogische Formel α ist zu entscheiden, ob α erfüllbar ist.
- ▶ Das „Entscheidungsproblem“ des Erfüllbarkeitsproblems ist

$$SAT = \{ \alpha \mid \text{die aussagenlogische Formel } \alpha \text{ ist erfüllbar} \}.$$

● Das Clique Problem:

- ▶ Für einen ungerichteten Graphen G und eine Zahl $k \in \mathbb{N}$ ist zu entscheiden, ob G eine **Clique** der Größe k besitzt.
 - ★ Eine Clique ist eine Menge von Knoten, so dass je zwei Knoten durch eine Kante verbunden sind.
- ▶ Das Entscheidungsproblem des Clique Problems ist

$$CLIQUE = \{ (G, k) \mid G \text{ hat eine Clique der Größe } k \}.$$

• Das Problem der längsten Wege:

- ▶ Für einen gerichteten Graphen G und eine Zahl k ist zu entscheiden, ob G einen Weg der Länge mindestens k besitzt.
- ▶ Das Entscheidungsproblem LW ist

$$LW = \{(G, k) \mid G \text{ besitzt einen Weg der Länge mindestens } k\}.$$

- ▶ Kürzeste Wege sind einfach, längste Wege sind schwer. **Warum?**

• Das Binpacking Problem:

- ▶ Entscheide, ob n Objekte mit den Gewichten $0 \leq w_1, \dots, w_n \leq 1$ in höchstens k Behälter gepackt werden können.
- ▶ Kein Behälter darf Objekte mit Gesamtgewicht > 1 aufnehmen.
- ▶ Das Entscheidungsproblem des Binpacking Problems ist

$$BINPACKING = \{(w_1, \dots, w_n, k) \mid k \text{ Behälter können alle Objekte aufnehmen}\}.$$

● Das Shortest-Common-Superstring Problem:

- ▶ Für Strings s_1, \dots, s_n und eine Zahl m entscheide, ob es einen „Superstring“ s der Länge höchstens m gibt, der alle Strings s_i als Teilstrings enthält.
- ▶ In der **Shotgun-Sequenzierung** wird ein DNA-String
 - ★ in kleine, sich überlappende Fragmente s_i zerlegt.
 - ★ Dann werden die Fragmente sequenziert und
 - ★ der ursprüngliche DNA-String s wird aus den sequenzierten Fragmenten als kürzester Superstring rekonstruiert.

Man nimmt also an, dass der DNA-String ein kürzester Superstring seiner Fragmente ist.

● Das Faktorisierungsproblem:

- ▶ Für eine natürliche Zahl N und ein Intervall $[a, b]$, jeweils in **Binärdarstellung** gegeben, entscheide, ob eine Zahl aus dem Intervall $[a, b]$ die Zahl N teilt.
- ▶ Die (vermutete) Schwierigkeit des Faktorisierungsproblems ist die Grundlage der Public-Key Kryptographie.
- ▶ Allerdings ist das Faktorisierungsproblem in aller Wahrscheinlichkeit **einfacher** als die obigen Probleme.

● Die Minimierung von Schaltungen:

- ▶ Gegeben ist ein Schaltkreis S und eine Zahl m . Entscheide, ob es einen mit S äquivalenten Schaltkreis mit höchstens m Gattern gibt.
- ▶ Die Minimierung von Schaltungen ist in aller Wahrscheinlichkeit sogar noch **schwieriger** als alle obigen Probleme. **Warum?**

Es genügt nicht eine Schaltung mit höchstens m Gattern zu raten, denn wie soll die Äquivalenz **effizient** überprüft werden?

- 1 Wann sollten wir eine Berechnung effizient nennen?
 - ▶ Wir müssen ein Rechnermodell zugrundelegen, das nicht nur auf die Rechner der heutigen Technologie zutrifft, sondern auch die Basis zukünftiger Rechnergenerationen ist.
 - ▶ Wir werden dann die Klasse P definieren.
- 2 Welches Rechnermodell kann Probleme mit kurzen, schnell verifizierbaren Lösungen lösen?
 - ▶ Ein Gedankenexperiment: Wir entwickeln ein nichtdeterministisches Rechnermodell, das neben den Fähigkeiten heutiger Rechner auch die Fähigkeit des Rätens besitzt.
 - ▶ Mit Hilfe dieses nichtdeterministischen Rechnermodells definieren wir die Klasse NP .
- 3 Wir müssen die Schwierigkeit von Problemen in NP vergleichen.
 - ▶ Wir definieren NP -vollständige Probleme als die schwierigsten Probleme in NP .

Die Klasse P

Welches formale Rechnermodell sollten wir wählen?

- **Forderung:** Das Rechnermodell sollte die wesentlichen Eigenschaften **heutiger** und **zukünftiger** Rechner beinhalten.
 - ▶ Ein Rechner muss jederzeit auf die Eingabe zugreifen können und Rechnungen auf einem unbeschränkt großen, jederzeit modifizierbaren Speicher ausführen können.
 - ▶ Die letzten technologischen Fortschritte sollten wir **ignorieren**, solange Berechnungen nicht um einen **Quantensprung** beschleunigt werden:

Jedes realistische Rechnermodell sollte mit einer höchstens **polynomiellen** Verzögerung durch unser Rechnermodell simulierbar sein.

- ▶ Unser Rechnermodell sollte so einfach wie möglich sein.

Wir arbeiten mit Turingmaschinen, also Nähmaschinen, die nebenbei noch rechnen.

- **Die Architektur:**

- ▶ Eine Turingmaschine besitzt ein nach links und nach rechts unendliches, **lineares Band**, das in Zellen unterteilt ist.
- ▶ Die Zellen speichern Buchstaben aus einem Arbeitsalphabet Γ und besitzen Zahlen aus \mathbb{Z} als Adressen.
- ▶ Die Turingmaschine manipuliert ihr Band mit Hilfe eines **Lese-/Schreibkopfes**. Der Kopf kann den von einer Zelle gespeicherten Buchstaben lesen, ihn überdrucken und zur linken oder rechten Nachbarzelle wandern, bzw. auf der Zelle verbleiben.

- **Die Startkonfiguration:**

- ▶ Die Eingabe $w = w_1 \cdots w_n$ ist in den Zellen $1 \dots, n$ gespeichert, wobei Zelle i (mit $1 \leq i \leq n$) den Buchstaben w_i speichert.
- ▶ Alle verbleibenden Zellen speichern das **Blankensymbol** \sqcup .
- ▶ Der Kopf liest Zelle 1.
- ▶ Die Maschine befindet sich im **Startzustand** q_0 .

Die Berechnung einer Nähmaschine

• Der Rechenschritt:

- ▶ Das Programm einer Turingmaschine wird durch eine partiell definierte **Zustandsüberföhrungsfunktion** δ mit

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{\text{links, bleib, rechts}\}$$

beschrieben. Q ist die Zustandsmenge, Γ das Arbeitsalphabet.

- ▶ Wenn die Maschine sich im Zustand q befindet, den Buchstaben $a \in \Gamma$ liest und wenn $\delta(q, a) = (q', a', \text{Richtung})$, dann
 - ★ überdruckt die Maschine den Buchstaben a mit a' ,
 - ★ wechselt in den Zustand q' und
 - ★ bewegt den Kopf zur durch $\text{Richtung} \in \{\text{links, bleib, rechts}\}$ vorgeschriebenen Nachbarzelle.
- Die Maschine **hält** im Zustand q , wenn sie im Zustand $q \in Q$ ist, den Buchstaben $\gamma \in \Gamma$ liest und wenn $\delta(q, \gamma) = \perp$ gilt, wenn also δ auf (q, γ) nicht definiert ist.
- Die Teilmenge $F \subseteq Q$ ist die Menge der akzeptierenden Zustände.
 - ▶ Die Maschine **akzeptiert** ihre Eingabe w , wenn sie in einem Zustand aus F hält.

- Eine Turingmaschine wird durch das 6-Tupel $M = (Q, \Sigma, \delta, q_0, \Gamma, F)$ beschrieben. Σ ist das Eingabealphabet. Für das Arbeitsalphabet Γ ist $\Sigma \cup \{B\} \subseteq \Gamma$.
- $L(M)$ ist das von M gelöste (Entscheidungs-)Problem, d.h. es ist

$$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}.$$

- $\text{schr\i{t}te}_M(w)$ ist die Anzahl der Schritte von M auf Eingabe w . Die worst-case Laufzeit von M auf Eingaben der Länge n ist dann

$$\text{zeit}_M(n) = \max\{\text{schr\i{t}te}_M(x) \mid x \in \Sigma^n\}.$$

Wie schnell kann eine Turingmaschine für Strings der Länge n erkennen, ob sie zur Sprache $a^x b^x$ gehören?

- (1) $\Theta(\log n)$
- (2) $\Theta(n)$
- (3) $\Theta(n^2)$
- (4) $\Theta(n^4)$
- (5) $\Theta(n^{n/2})$

Auflösung: (3) $\Theta(n^2)$

Effiziente Berechenbarkeit

Die Klasse \mathbb{P} besteht aus allen Entscheidungsproblemen L für die es eine Turingmaschine M mit $L = L(M)$ und

$$\text{zeit}_M(n) = O((n+1)^k) \text{ für eine Konstante } k$$

gibt. Wir sagen, dass L **effizient berechenbar** ist, wenn $L \in \mathbb{P}$ gilt.

Man kann zeigen:

All das, was in polynomieller Zeit auf einem (deterministischen) parallelen Supercomputer berechnet werden kann, gelingt auch in polynomieller Zeit auf einer Nähmaschine.

Die Definition der Klasse \mathbb{P} hängt nicht von der Architektur des Rechnermodell ab!

Wann gehört ein Entscheidungsproblem zur Klasse P ?

Es genügt zum Beispiel die Angabe eines deterministischen C++ Programms, das in polynomieller Zeit läuft.

Stimmen die Begriffe?

- **Das Positive:** Die Klasse P hängt nicht von der Architektur des Rechnermodells ab.
- **Das Negative:** Warum Einschränkung auf deterministische Berechnungen?
 - ▶ Randomisierte Berechnungen scheinen keinen Quantensprung zu ergeben,
 - ▶ aber Quantenrechner werden mächtiger sein, weil sie zum Beispiel effizient faktorisieren können.

+ Vom Standpunkt deterministischer Berechnungen:

- ▶ P ist zu groß, denn Algorithmen der Laufzeit $O(n^{1000})$ sind unrealistisch.
- ▶ Aber negative Aussagen der Form „Dieses Problem gehört nicht zur Klasse P “ sind deshalb umso stärker.

- Vom Standpunkt der Quantenberechnungen:

- ▶ P ist zu klein, denn P enthält das Faktorisierungsproblem nicht.

Alle Probleme, die durch Algorithmen mit polynomieller Laufzeit gelöst werden, gehören zur Klasse P .

- Alle regulären und kontextfreien Sprachen gehören zu P . (Siehe die Vorlesung „Theoretische Informatik 2“.)
- Sämtliche bisher behandelten Graphprobleme, wie etwa das Zusammenhangsproblem

$\{ G \mid G \text{ ist ein ungerichteter, zusammenhängender Graph} \}$

oder das Matchingproblem

$\{ (G, k) \mid G \text{ besitzt } k \text{ Kanten, die keinen gemeinsamen Endpunkt haben} \}$

gehören zu P .

- Das Entscheidungsproblem des kürzeste-Wege Problems

$\{(G, s, t, k) \mid G \text{ hat einen Weg von } s \text{ nach } t \text{ der Länge } \leq k\}$

gehört zu P , während das Problem des längsten Weges wahrscheinlich nicht in P liegt.

- Das Entscheidungsproblem der linearen Programmierung

$\{(A, b, c, t) \mid \text{Es gibt } x \text{ mit } A \cdot x \leq b, x \geq 0 \text{ und } c^t \cdot x \geq t\}$

gehört zu P , während die 0-1 Programmierung oder die ganzzahlige Programmierung wahrscheinlich nicht in P liegen.

- Erst vor wenigen Jahren wurde gezeigt, dass das Primzahlproblem

$$\{N \mid N \text{ ist eine Primzahl}\}$$

zu \mathcal{P} gehört. Beachte, dass $\lfloor \log_2 N \rfloor + 1$ die **Länge** der Eingabe N ist. Das Faktorisierungsproblem

$$\{(N, a, b) \mid \text{es gibt } x \in [a, b] \text{ und } x \text{ teilt } N\}$$

wird hingegen höchstwahrscheinlich nicht in \mathcal{P} liegen, zumindest hoffen dies die Kryptographen.

- 1 Die Definition der Klasse P und des Begriffs der **effizienten Berechnung**.
- 2 Ein Gedankenexperiment: **Welches Rechnermodell kann Probleme mit kurzen, schnell verifizierbaren Lösungen lösen?**
 - ▶ Wir entwickeln ein **nichtdeterministisches** Rechnermodell, das neben den Fähigkeiten heutiger Rechner auch die Fähigkeit des Rätens besitzt.
 - ▶ Mit Hilfe dieses nichtdeterministischen Rechnermodells definieren wir die Klasse NP .
- 3 Wir müssen die Schwierigkeit von Problemen in NP vergleichen.
 - ▶ Wir definieren **NP -vollständige** Probleme als die schwierigsten Probleme in NP .

Die Klasse NP

Nichtdeterministische Berechnungen

Wir möchten die Klasse aller Probleme mit kurzen, effizient verifizierbaren Lösungen definieren.

- Wir verwenden wieder Turingmaschinen, aber erlauben zu **raten**.
- Statt mit einer Zustandsüberföhrungsfunktion arbeiten wir mit einer **Zustandsüberföhrungsrelation**

$$\delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \{\text{links, bleib, rechts}\}.$$

- ▶ Wenn sich die Maschine im Zustand $q \in Q$ befindet und den Buchstaben $\gamma \in \Gamma$ liest, dann **kann** die nichtdeterministische Turingmaschine jeden Befehl

$$(q, \gamma) \rightarrow (q', \gamma', \text{Richtung})$$

ausföhren, solange

$$(q, \gamma, q', \gamma', \text{Richtung}) \in \delta \text{ gilt.}$$

Wann wird akzeptiert?

Eine nichtdeterministische Turingmaschine wird wieder durch das 6-Tupel $M = (Q, \Sigma, \delta, q_0, \Gamma, F)$ beschrieben, wobei aber diesmal δ eine Zustandsüberführungsrelation ist.

- M kann auf Eingabe w viele verschiedene Berechnungen durchlaufen:

Einige Berechnungen akzeptieren w , andere verwerfen w .

- M soll die Fähigkeit haben, eine Lösung zu raten:

Wir sollten sagen, dass M die Eingabe w genau dann akzeptiert, wenn mindestens eine Berechnung von M auf Eingabe w akzeptierend ist!

- Das von M gelöste Entscheidungsproblem $L(M)$ ist

$$L(M) = \{w \mid M \text{ akzeptiert } w\}.$$

Wie schnell wird gerechnet?

Wir fixieren eine Eingabe w . Dann hat eine nichtdeterministische Turingmaschine möglicherweise **langsame und schnelle** Berechnungen für Eingabe w . Wie setzen wir die Laufzeit fest?

- Wir möchten das „Raten **kurzer, schnell verifizierbarer** Lösungen“ formalisieren.
Wir wählen die Schrittzahl der **schnellsten** akzeptierenden Berechnung mit Eingabe w als die Laufzeit von M auf w .
- Die worst-case Laufzeit von M auf Eingaben der Länge n ist

$$\text{Zeit}_M(n) = \max \{ \text{Laufzeit von } M \text{ auf } w \mid w \in \Sigma^n \text{ und } M \text{ akzeptiert } w \}.$$

Die Definition von NP

Ein Problem L gehört genau dann zu NP, wenn es eine nichtdeterministische Turingmaschine M mit $L = L(M)$ und $\text{Zeit}_M(n) = O((n+1)^k)$ für eine Konstante k gibt.

- Auch nichtdeterministische Supercomputer können durch nichtdeterministische Nähmaschinen mit nur polynomieller Verzögerung simuliert werden.
- **Um nachzuweisen, dass ein Problem zu NP gehört, genügt zum Beispiel die Angabe eines nichtdeterministischen C++ Programms, das in polynomieller Zeit läuft.**

NP enthält alle Probleme mit kurzen, schnell verifizierbaren Lösungen: Übersetze „kurz und schnell“ mit „polynomiell“.

- Die Definition der Klasse P und des Begriffs der **effizienten Berechnung**.
- Die Definition **nichtdeterministischer Berechnungen** und die Definition der Klasse NP .
 - ▶ NP enthält alle Probleme mit kurzen, schnell verifizierbaren Lösungen.
- **Wir müssen die Schwierigkeit von Problemen in NP vergleichen.**
 - ▶ Wir definieren **NP -vollständige** Probleme als die schwierigsten Probleme in NP .

Die polynomielle Reduktion

Schwierigkeit?

Wann sollten wir sagen, dass ein Entscheidungsproblem L_2 mindestens so schwierig wie ein Entscheidungsproblem L_1 ist?

Wenn wir L_1 effizient lösen können, vorausgesetzt wir können L_2 effizient lösen!

- Und was genau heißt das?

- Eine Eingabe w ist gegeben. Wir müssen feststellen, ob $w \in L_1$.
- Wir haben eine Frage frei und berechnen eine Eingabe $M(w)$ mit Hilfe einer deterministischen Turingmaschine M , die in Zeit $\text{poly}(|w|)$ rechnet.
- Wir fragen, ob $M(w)$ in L_2 liegt und übernehmen den Urteilsspruch für w und L_1 .

Eine polynomielle Reduktion von L_1 auf L_2

Wir sagen, dass L_1 auf L_2 **polynomiell reduzierbar** ist und schreiben

$$L_1 \leq_p L_2,$$

wenn es eine deterministische Turingmaschine M gibt, so dass für **alle** w

$$w \in L_1 \Leftrightarrow M(w) \in L_2$$

gilt.

M berechnet die **transformierte** Eingabe $M(w)$ in polynomieller Zeit. Wir nennen M eine **transformierende** Turingmaschine.

Die NP -Vollständigkeit

- **Das Ziel:** NP-vollständige Probleme sollen die schwierigsten Probleme in NP sein.
- Sei L ein Entscheidungsproblem.
 - (a) L heißt NP-hart genau dann, wenn

$$K \leq_p L \text{ für alle Probleme } K \in \text{NP}.$$

- (b) L heißt NP-vollständig genau dann, wenn $L \in \text{NP}$ und L ein NP-hartes Problem ist. (Web)

- Wir haben die Klassen P und NP eingeführt und die NP -Vollständigkeit definiert.
- Haben wir die richtigen Begriffsbildungen gefunden? Wir müssen Antworten auf die folgenden Fragen finden:
 - ▶ Gibt es überhaupt NP -vollständige Probleme?
 - ▶ Angenommen, es gibt ein NP -vollständiges Problem. Wie kann dann die NP -Vollständigkeit weiterer Probleme gezeigt werden?
 - ▶ Angenommen, es ist $P \neq NP$.
Kann dann gezeigt werden, dass **jedes** NP -vollständige Problem schwierig ist, also **keine** effizienten Algorithmen besitzt?

Wir beginnen mit der letzten Frage.

NP-vollständige Probleme sind schwierig

Das Entscheidungsproblem L sei NP-vollständig. Wenn $P \neq NP$, dann ist $L \notin P$.

- Angenommen, die Aussage ist falsch, d.h. L gehört zu P .
- Wir müssen $P = NP$ zeigen.
 - ▶ Sei K ein beliebiges Problem in NP . Wir müssen zeigen, dass K bereits in P liegt.
 - ▶ Da L NP-vollständig ist, gilt $K \leq_p L$.
 - ▶ Es ist $K \in P$. Warum?
 - ★ Sei M die transformierende Turingmaschine für die Reduktion $K \leq_p L$.
 - ★ Wende M auf die Eingabe w von K an und berechne $M(w)$ in polynomieller Zeit.
 - ★ Da $L \in P$ überprüfe in polynomieller Zeit, ob $M(w) \in L$ gilt.
 - ★ Es ist $w \in K$ **genau dann, wenn** $M(w) \in L$:
Eingabe w wird genau dann akzeptiert, wenn $M(w) \in L$ gilt.
 - ▶ Da ein beliebiges Problem $K \in NP$ zu P gehört, folgt $NP \subseteq P$ und damit $NP = P$.

Die polynomielle Reduktion ist transitiv

Eine technische Vorbereitung:

Seien L_1 , L_2 und L_3 Entscheidungsprobleme.

Wenn $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, dann $L_1 \leq_p L_3$.

- M_1 und M_2 seien die transformierenden Turingmaschinen für die Reduktion $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$.
- Wende zuerst M_1 und dann M_2 auf die Eingabe w für L_1 an: Wir erhalten eine transformierende Turingmaschine für die Reduktion $L_1 \leq_p L_3$, denn aus

$$\forall w (w \in L_1 \Leftrightarrow M_1(w) \in L_2) \text{ und } \forall u (u \in L_2 \Leftrightarrow M_2(u) \in L_3)$$

folgt

$$\forall w (w \in L_1 \Leftrightarrow M_1(w) \in L_2 \Leftrightarrow M_2(M_1(w)) \in L_3).$$

Wie weist man NP-Vollständigkeit nach?

- Wenn L_1 NP-vollständig ist und wenn $L_1 \leq_p L_2$, dann ist L_2 ein NP-hartes Problem.
- Wenn zusätzlich $L_2 \in \text{NP}$ gilt, dann ist L_2 NP-vollständig.

Sei K ein beliebiges Problem in NP. Dann müssen wir die Reduktion $K \leq_p L_2$ nachweisen.

- ▶ Da L_1 NP-vollständig ist, wissen wir, dass $K \leq_p L_1$ gilt.
- ▶ Aus $K \leq_p L_1$ und $L_1 \leq_p L_2$ folgt $K \leq_p L_2$ und das war zu zeigen.

- Wenn ein NP-vollständiges Problem in P liegt, dann ist $P = NP$:
 - ▶ Heutige Rechner hätten die Kraft des Rätens ohne dass wir dies je beobachtet hätten.
 - ▶ Sehr starke Indizien dafür, dass NP-vollständige Probleme **nicht** effizient berechenbar sind.
- Was ist noch zu tun?
 - ▶ Wir kennen bisher kein einziges NP-vollständiges Problem.
 - ▶ Mehr noch, eine große Klasse **wichtiger Probleme** sollte NP-vollständig sein.
 - ▶ Heute kennt man **Tausende** NP-harter Probleme:
 - ★ In der technischen Informatik (Schaltungsminimierung, Verifikation),
 - ★ in Betriebssystemen (Lastverteilung, Scheduling),
 - ★ in der Bioinformatik (shortest common superstring, multiples Alignment, phylogenetische Bäume, Proteinfaltung),
 - ★ in Datenbanken (Bestimmung von Keys)
 - ★ oder in der Berechnung von **Gewinn-Strategien** für Spiele.