Informatik-Propädeutikum

Dozentin: Dr. Claudia Ermel

Betreuer: Sepp Hartung, André Nichterlein, Clemens Hoffmann
Sekretariat: Christlinde Thielcke (TEL 509b)

TU Berlin
Institut für Softwaretechnik und Theoretische Informatik
Prof. Niedermeier
Fachgruppe Algorithmik und Komplexitätstheorie
http://www.akt.tu-berlin.de

Wintersemester 2013/2014

Gliederung

5 Algorithmische Komplexität
- O-Notation zur Laufzeitanalyse
- Traveling Salesperson Problem
- 3-SAT: Erfüllen aussagenlogischer Formeln
- 3-Coloring: Färben von Graphen mit maximal 3 Farben
- Hamilton-Kreis: Finden von Wegen durch alle Knoten im Graphen
- Graphprobleme: Independent Set, Vertex Cover, Dominating Set
- Die Komplexitätsklassen **P** und **NP**
- Das Reduktionskonzept und **NP**-Vollständigkeit

Algorithmische Komplexität

Bisher: es gibt nicht-berechenbare Probleme, z.B. das Halteproblem und das Postsche Korrespondenzproblem.

... und es gibt viele Algorithmen zur Lösung konkreter Probleme wie z.B. den Propose&Reject-Algorithmus für Stable Matching oder den rekursiven Algorithmus zum Potenzieren einer Zahl.

„Offensichtlich“ sind diese Algorithmen korrekt, doch wie bewerten bzw. analysieren wir ihre **Effizienz**, sprich die von ihnen benötigte Laufzeit,[6] also die Anzahl der von ihnen ausgeführten Elementaroperationen?

Naheliegend: Laufzeit ist von der Eingabegröße abhängig – große Eingaben benötigen in der Regel mehr Laufzeit...

Zentrale Frage: Wann nennen wir ein Berechnungsproblem effizient lösbar? Wie lässt sich das formalisieren?

---

[6] Eine zweite wichtige Ressource ist der Speicherplatzbedarf; dieser ist in den meisten (aber nicht allen) Anwendungen gegenüber der Laufzeit sekundär.

O-Notation zur Laufzeitanalyse

Eine genaue Laufzeitangabe für einen Algorithmus in Abhängigkeit von der Eingabegröße n ist oft schwierig und vom jeweiligen Maschinenmodell abhängig. Dem begegnet man (teilweise) mit der Benutzung der O-Notation:

Definition: Seien f und g Funktionen von den natürlichen in die reellen Zahlen. Dann bezeichnet man mit $O(f(n))$ die Menge aller Funktionen g, so dass gilt

$$\exists c > 0 \quad \exists n_0 > 0 \quad \forall n \geq n_0 : g(n) \leq c \cdot f(n).$$

(Intuition: Ignoriere konstante Faktoren.)

Beispiele:

- $3n^4 + 5n^3 + 7\log_2 n \in O(n^4)$.
- $n \log_2 n \in O(n^2)$.
- $n\sqrt{n} \in O(n^2)$.
- $n^{10} \in O(2^n)$.

Propose&Reject für kdk Matching – revisited

Zur Erinnerung:

Propose&Reject [Gale, Shapley 1962]

Initialisiere alle $m \in M$ und alle $w \in W$ als „frei“
while $\exists m \in M$: m ist frei und $\exists w \in W$ der m noch keinen Antrag gemacht hat
  $w \leftarrow$ erste noch „unbeantragte“ Frau in m’s Präferenzfolge
  if w ist frei **then:**
    (m, w) wird Paar, $m \leftarrow$„verlobt“, $w \leftarrow$„verlobt“
  else if w zieht m ihrem aktuellen „Verlobten“ m' vor **then:**
    (m, w) wird Paar, $m \leftarrow$„verlobt“, $w \leftarrow$„verlobt“, $m' \leftarrow$„frei“
  else: w lehnt m ab

Unter der Annahme $|M| = |W| = n$ hat der Propose&Reject-Algorithmus eine Laufzeit von $O(n^2)$.

Hier und nachfolgend: Laufzeitanalyse immer auf den schlimmsten möglichen Fall bezogen (**Worst-Case-Szenario**)!

O-Notation und Laufzeitbeispiele

Notation	Bedeutung	Anschauliche Erklärung	Beispiele für Laufzeiten
$f \in \mathcal{O}(1)$	f ist beschränkt	f überschreitet einen konstanten Wert nicht (unabhängig vom Wert des Arguments).	Nachschlagen des x-ten Elementes in einem Feld.
$f \in \mathcal{O}(\log x)$	f wächst logarithmisch	f wächst ungefähr um einen konstanten Betrag, wenn sich das Argument verdoppelt. Die Basis des Logarithmus ist dabei egal.	Binäre Suche im sortierten Feld mit x Einträgen
$f \in \mathcal{O}(\sqrt{x})$	f wächst wie die Wurzelfunktion	f wächst ungefähr auf das Doppelte, wenn sich das Argument vervierfacht	naiver Primzahltest mittels Teilen durch jede ganze Zahl $\leq \sqrt{x}$
$f \in \mathcal{O}(x)$	f wächst linear	f wächst ungefähr auf das Doppelte, wenn sich das Argument verdoppelt.	Suche im unsortierten Feld mit x Einträgen (Bsp. Lineare Suche)
$f \in \mathcal{O}(x \log x)$	f hat super-lineares Wachstum		Fortgeschrittenere Algorithmen zum Sortieren von x Zahlen Mergesort, Heapsort
$f \in \mathcal{O}(x^2)$	f wächst quadratisch	f wächst ungefähr auf das Vierfache, wenn sich das Argument verdoppelt	Einfache Algorithmen zum Sortieren von x Zahlen Selectionsort
$f \in \mathcal{O}(2^x)$	f wächst exponentiell	f wächst ungefähr auf das Doppelte, wenn sich das Argument um eins erhöht	Erfüllbarkeitsproblem der Aussagenlogik (SAT) mittels exhaustivem Verfahren
$f \in \mathcal{O}(x!)$	f wächst faktoriell	f wächst ungefähr auf das $(x+1)$-fache, wenn sich das Argument um eins erhöht.	Problem des Handlungsreisenden (Mit Enumerationsansatz)

Einige Laufzeiten von bereits bekannten Algorithmen

Hinweis: Nachfolgend nehmen wir an, dass z.B. eine Zuweisung oder die Addition zweier Zahlen eine Elementaroperation ist.

1. Der rekursive Algorithmus zum Finden einer 6-Färbung einer Landkarte hat eine Laufzeit von $O(n^2)$, wobei n die Anzahl der Länder ist.
2. Der rekursive Algorithmus für die Türme von Hanoi hat eine Laufzeit von $O(2^n)$.
3. Der rekursive Algorithmus zur Berechnung der n-ten Fibonacci-Zahl hat eine Laufzeit von mindestens $O(2^{n/2})$.
4. Der iterative Algorithmus zur Berechnung der n-ten Fibonacci-Zahl hat eine Laufzeit von $O(n)$.

Zentrale Frage: Welche der obigen Algorithmen gelten als **effizient**?

Die vereinfachte Antwort der (theoretischen) Informatik ist, dass alle Algorithmen mit **polynomieller Laufzeit** als effizient gelten, und solche mit **„super-polynomieller“ Laufzeit** als ineffizient. Oben sind also der erste und der letzte Algorithmus effizient.

Funktionenwachstum

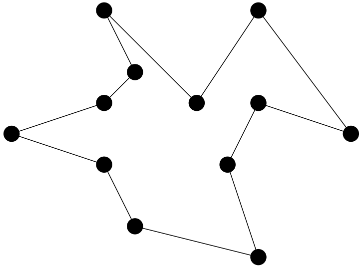
Der Unterschied zwischen „**polynomiellem Wachstum**“ einer Funktion und „**exponentiellem Wachstum**“ ist enorm:

n	n^2	2^n	$n!$
5	25	32	120
10	100	1024	$\approx 3 \cdot 10^6$
20	400	$\approx 10^6$	$\approx 2 \cdot 10^{18}$
50	2500	$\approx 10^{15}$	$\approx 3 \cdot 10^{64}$
100	10000	$\approx 10^{30}$	$\approx 9 \cdot 10^{157}$

Beispiel eines Berechnungsproblems mit bester bekannter Laufzeit $O(2^n)$: Traveling Sales Person (TSP)...

Traveling Salesperson Problem (TSP) I

Eingabe: n Punkte mit paarweisen Abständen und eine Zahl ℓ.
Aufgabe: Gibt es eine Rundtour der Länge $\leq \ell$, die alle Punkte genau einmal besucht?



Triviale Lösungsfindung: Probiere alle $n!$ Möglichkeiten (Permutationen der Punkte), wähle die Beste aus.

Verbesserung: Mithilfe der Methode des dynamischen Programmierens lässt sich die Laufzeit i.w. zu $O(2^n)$ verbessern; eine weitere beweisbare Verbesserung ist seit fünfzig Jahren offen!

Funktionenwachstum und hypothetische Laufzeiten

Algorithmenlaufzeit (d.h. Anzahl der Elementaroperationen) bei Eingabegröße n und hypothetische Rechenzeit bei Annahme von 10^9 Elementaroperationen pro Sekunde.

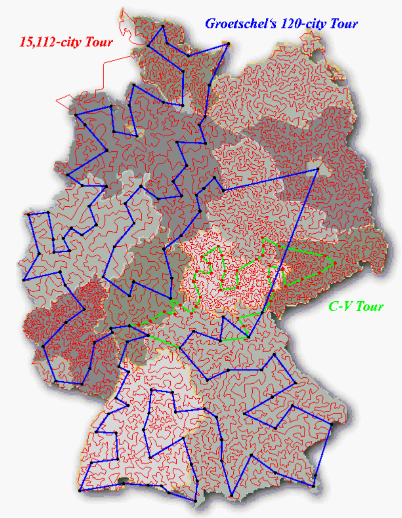
n	$n!$	2^n	Laufzeit $n!$	Laufzeit 2^n
5	120	32	10^{-7} Sekunden	$< 10^{-7}$ Sekunden
10	$\approx 3 \cdot 10^6$	1024	10^{-3} Sekunden	10^{-6} Sekunden
20	$\approx 2 \cdot 10^{18}$	$\approx 10^6$	77 Jahre	10^{-3} Sekunden
50	$\approx 3 \cdot 10^{64}$	$\approx 10^{15}$	$9 \cdot 10^{47}$ Jahre	12 Tage
100	$\approx 9 \cdot 10^{157}$	$\approx 10^{30}$	$3 \cdot 10^{141}$ Jahre	10^{13} Jahre

Bemerkung: In der Praxis gelten für Eingabegröße n selbst Laufzeiten wie $O(n^2)$ oft als ineffizient; für die meisten Probleme ist keine Laufzeit besser als $O(n)$ (sogenannte **Linearzeit**) zu erreichen.

Aber was tun bei exponentiellen Laufzeiten?

Traveling Salesperson Problem (TSP) II

Mit ausgefeilten Heuristiken lassen sich dennoch vergleichsweise große TSP-Instanzen exakt lösen (Fortschritt durch Algorithmik!):



Schnellere Rechner bauen? Hilft praktisch nichts!

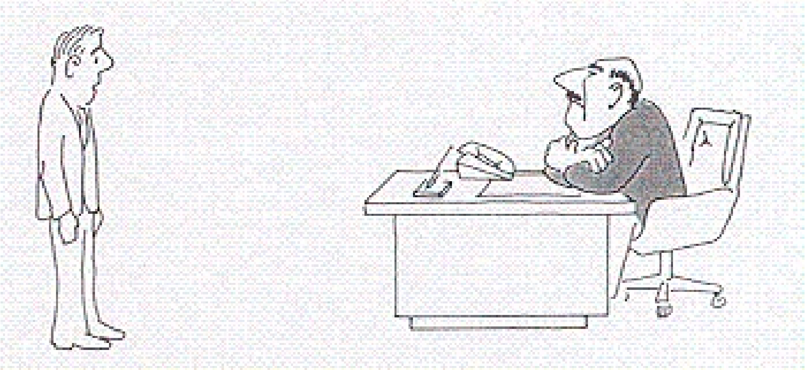
Im Kampf gegen exponentielle Laufzeiten helfen schnellere Rechner kaum:
Nehmen wir an, wir brauchen 2^n Rechenschritte zum Finden z.B. einer kürzesten Rundtour bei einer TSP-Instanz mit n Punkten. Dann könnte man

- mit tausendfach schnelleren Rechnern in etwa der gleichen Zeit Rundtouren in Eingaben mit 10 mehr Punkten finden;
- mit millionenfach schnelleren Rechnern in etwa der gleichen Zeit Rundtouren in Eingaben mit 20 mehr Punkten finden.

$\rightsquigarrow$ Fluch des exponentiellen Wachstums!

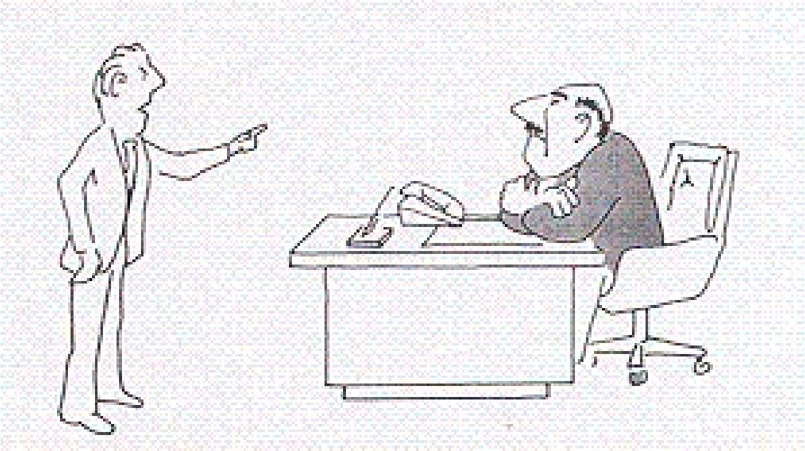
Wie wissen wir, ob wir nicht nur zu dumm waren, für wichtige Probleme Polynomzeit- statt Expontialzeitalgorithmen zu finden?
Was, wenn der Chef unbedingt einen schnellen Algorithmus fordert?
Die möglichen Sachlagen sind folgende:

Zu dumm für einen effizienten Algorithmus?



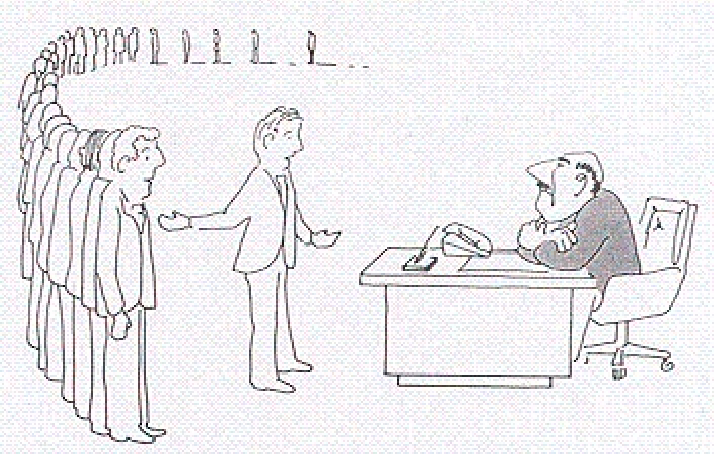
Ich kann keinen effizienten Algorithmus finden. Ich glaube, ich bin einfach zu dumm.

Es gibt keinen effizienten Algorithmus!



Ich kann keinen effizienten Algorithmus finden, weil es keinen gibt.

Nicht nur ich finde keinen Algorithmus!



Ich kann keinen effizienten Algorithmus finden, aber all diese berühmten, klugen Leute können es auch nicht.

Berechnungsschwere Entscheidungsprobleme: 3-SAT

Eingabe: Aussagenlogische Formel F in „konjunktiver Normalform“ („ein großes UND von kleinen ODERn“) mit höchstens drei Literalen pro Klausel.

Frage: Ist F *erfüllbar*, d.h. gibt es eine $\{0, 1\}$-wertige Belegung der in F verwendeten Booleschen Variablen derart, dass F zu *wahr* (1) ausgewertet wird?

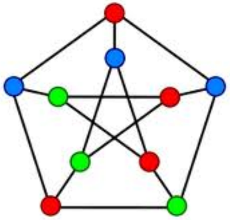
Beispiel: $(x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_2 \vee x_3 \vee x_4)$ ist erfüllbar z.B. mit $x_1 = 0$, $x_2 = 0$, $x_3 = 1$ (und x_4 beliebig).

Achtung: Im Gegensatz zu 3-SAT ist 2-SAT (nur zwei statt drei Literale pro Klausel) effizient lösbar (nichttrivialer Algorithmus)!

Berechnungsschwere Entscheidungsprobleme: 3-Coloring

Eingabe: Ein ungerichteter Graph.

Frage: Lassen sich die Knoten des Graphen mit drei Farben so färben, dass keine zwei mit einer Kante verbundenen Knoten die gleiche Farbe haben?



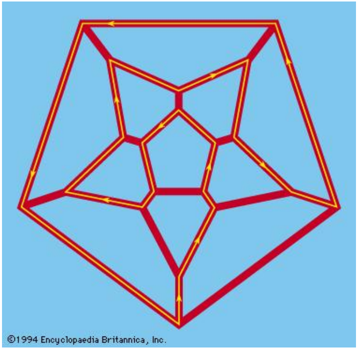
Achtung: Im Gegensatz zu 3-Coloring ist 2-Coloring (nur zwei statt drei Farben) effizient lösbar (einfacher Algorithmus)!

Berechnungsschwere Entscheidungsprobleme: Hamilton-Kreis

Das Hamilton-Kreis-Problem ist i.w. ein Spezialfall des TSP (warum?):

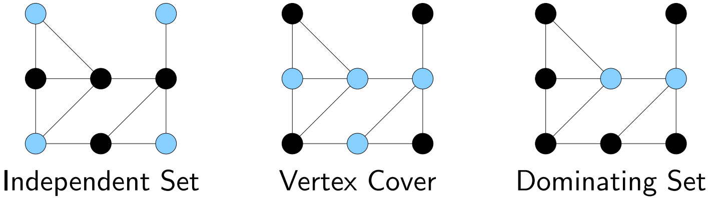
Eingabe: Ein ungerichteter Graph.

Frage: Gibt es eine Route durch den Graphen, die jeden Knoten genau einmal durchläuft und dann zum Ausgangsknoten zurückkehrt?



Achtung: Im Gegensatz zum Finden eines Hamilton-Kreises ist das Finden eines Euler-Kreises leicht (warum?)!

Weitere berechnungsschwere Entscheidungsprobleme IV



Eingabe: Ein ungerichteter Graph G und eine Zahl $k > 0$.
Independent-Set-Frage: Gibt es k Knoten in G, die paarweise untereinander nicht mit einer Kante verbunden (adjazent) sind?

Vertex-Cover-Frage: Gibt es k Knoten in G, sodass jede Kante in G mindestens einen dieser Knoten als Endpunkt hat?

Dominating Set-Frage: Gibt es k Knoten in G, sodass jeder andere Knoten mindestens einen dieser k Knoten als Nachbarn hat?

Bemerkung: Independent Set und Vertex Cover sind sehr eng verwandt und „gleichschwer" zu lösen (warum?)!

Die Komplexitätsklasse NP

Alle bisher betrachteten berechnungsschweren Probleme liegen in der Klasse **NP**, also insbesondere TSP, 3-SAT, 3-Coloring, etc.

Sie haben folgendes gemeinsam:
Hat man eine Lösung gefunden (dies ist der berechnungsintensive Anteil), so ist leicht zu überprüfen, ob die gefundene bzw. vorgeschlagene Lösung tatsächlich die geforderten Eigenschaften erfüllt!

Insbesondere gilt, dass jede Lösung in **polynomieller** Laufzeit überprüfbar ist (daher N**P**!).

Zusammenfassend heißt das, dass alle Probleme in NP mit einem „**Rate&Verifiziere-Algorithmus**“ gelöst werden können:

1. Rate eine Lösung.
2. Verifiziere, ob die geratene Lösung korrekt ist.

Das Raten ist der kritische Teil, da er „**Nichtdeterminismus**“ (daher **N**P!) voraussetzt und bislang nichts Besseres bekannt ist als alle Ratemöglichkeiten (leider sind das exponentiell viele) durchzuprobieren um eine gültige Lösung zu finden (falls sie existiert).

Rate&Verifiziere am Beispiel 3-SAT

Zu ratende Lösung: Belegung der Booleschen Variablen mit Werten aus $\{0, 1\}$;

bei n Variablen gibt es 2^n mögliche verschiedene Belegungen, die ggf. alle zu überprüfen sind.
Durch nichtdeterministisches Raten wird dieser Aufwand „umgangen“.

(Nichtdeterminismus ist kein reales Rechnerkonzept, sondern ein gedankliches Hilfsmittel!)

D.h.: Ein „Beweis“ besteht hier aus genau n Bits, nämlich den Wahrheitswerten 0 oder 1 für die n Variablen der Formel.

Verfikation der Lösung: Belege die Formelvariablen gemäß der jeweiligen Belegung mit den Werten 0 und 1 und überprüfe, ob sich die Formel zu 1 auswertet.

Die Komplexitätsklassen P und NP

Phänomen: Z.B. 2-Coloring ist leicht in polynomieller Zeit zu entscheiden, hingegen 3-Coloring scheinbar nicht. Beide haben polynomiell große „**Beweise**“, sprich die korrekten Färbungen (je Knoten Angabe einer Farbe).
Dies führt zur Unterscheidung der zwei wichtigsten „**Komplexitätsklassen**“ der Informatik:

P: Menge von Entscheidungsproblemen, die sich mit **polynomiell** langen Beweisen lösen lassen, welche auch in **polynomiell** vielen Schritten gefunden werden können.

NP: Menge von Entscheidungsproblemen, die sich mit **polynomiell** langen Beweisen lösen lassen.

2-Coloring liegt in **P**, 3-Coloring aber lediglich in **NP**. 3-Coloring gehört zu den schwierigsten Problemen in **NP**, d.h. zu den so genannten „**NP**-vollständigen Problemen“.

Das P versus NP-Problem

Klar ist, dass P $\subseteq$ NP.

Die große offene Frage: Ist

$$\mathbf{P} = \mathbf{NP}?$$

Anders gesagt: Besitzen alle Entscheidungsprobleme mit kurzen Beweisen auch schnelle Konstruktionsverfahren (also Algorithmen), um diese Beweise zu finden?

Bemerkung: Es gibt viele Tausende, in verschiedensten praktischen Anwendungen auftretende **NP**-Probleme, für die bisher kein effizienter Algorithmus angegeben werden konnte. Und das, obwohl sich schon Abertausende von Forschern daran versucht haben!

Zur Soziologie des P versus NP-Problems

Zunächst: Könnte man die Frage $\mathbf{P} = \mathbf{NP}$ beantworten, so wäre man ein Weltstar der Wissenschaft, wie es ihn bislang nicht gab ... und auch um eine Million Dollar reicher (Clay Mathematics Institute).

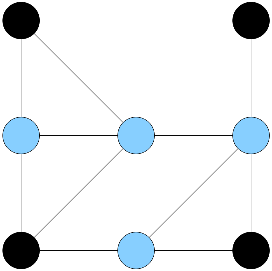
Wohl kein wissenschaftliches Problem wurde von derart vielen Menschen erfolglos bearbeitet.

Eine **Umfrage** im Jahre 2002 in den USA unter 100 namhaften Theoretischen Informatikern ergab folgende Gefühlslage:

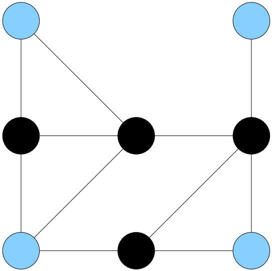
1. 5 glaubten an die Beantwortung bis 2009;
2. 35 an die Beantwortung bis 2049;
3. 7 an die Beantwortung zwischen 2100 und 2110;
4. 5 an die Beantwortung zwischen 2200 und 2300;
5. 5 daran, dass es nie gelöst wird.

Weiterhin glaubten 61 der Befragten, dass $\mathbf{P} \neq \mathbf{NP}$ gilt, nur 9 glaubten $\mathbf{P} = \mathbf{NP}$, der Rest traute sich nicht...
2012 wurde die Umfrage wiederholt mit i.w. gleichem Ergebnis.

Independent Set ist so schwer wie Vertex Cover



Vertex Cover



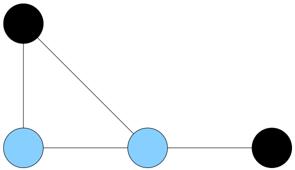
Independent Set

Einfache Beobachtung: Die Lösungsmengen sind „komplementär“ zueinander („aus schwarz wird blau“ und umgekehrt).

Dominating Set ist mindestens so schwer wie Vertex Cover

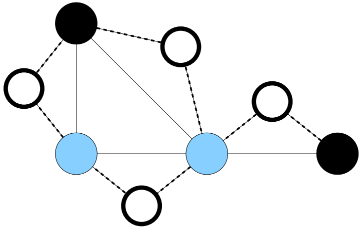
Kann man Dominating Set effizient lösen, so auch Vertex Cover!

Vertex Cover



Knotenmenge V
Kantenmenge E
k

Dominating Set



Knotenmenge $V \cup \{v_e \mid e \in E\}$
Kantenmenge $E \cup \{\{v_e, x\} \mid e \in E, x \in e\}$
k

Die Lösung der Dominating Set-Instanz rechts liefert eine Lösung der Vertex Cover-Instanz links!

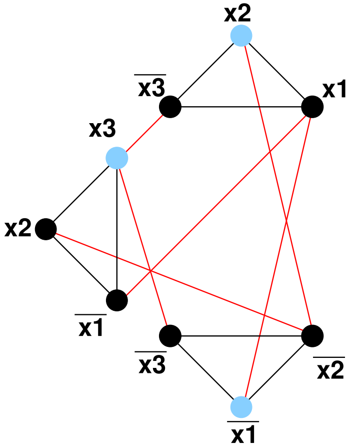
Independent Set ist mindestens so schwer wie 3-SAT

Konstruktion

- Klausel $F_i := l_{i1} \vee l_{i2} \vee l_{i3} \rightsquigarrow$ Dreieck mit Knoten l_{i1}, l_{i2}, l_{i3}.
- Knoten aus verschiedenen Dreiecken werden durch eine Kante verbunden, wenn sich die entsprechenden Literale „widersprechen", also eines Negation des anderen ist.
- $k := m$, d.h. Anzahl der Klauseln.

Beispiel:

$(x1 \vee x2 \vee \overline{x3}) \wedge (\overline{x1} \vee x2 \vee x3) \wedge (\overline{x1} \vee \overline{x2} \vee \overline{x3})$



Das Reduktionskonzept und **NP**-Vollständigkeit

In vorangehenden Beispielen haben wir die Lösung eines Entscheidungsproblems auf die Lösung eines anderen Entscheidungsproblems „**reduziert**“ (besser: zurückgeführt). Folgendes war dabei wichtig:

1. Die „Zurückführung“ („Übersetzung“ des einen Problems in das andere) ist in polynomieller Laufzeit (also effizient) berechenbar.
2. Die Eingabe des einen Problems ergibt dann und nur dann die Antwort „ja“ wenn es auch die neu berechnete Instanz des neuen Problems tut.

Ein Entscheidungsproblem A heißt **NP-vollständig**, wenn es

1. in **NP** liegt und
2. jedes andere Probleme in **NP** sich mit einer wie oben beschriebenen Übersetzung auf A zurückführen lässt.

NP-Vollständigkeit

Intuition:

Die **NP**-vollständigen Probleme sind die schwersten Probleme in **NP**!

Dazu gehören: TSP, 3-SAT, 3-Coloring, Vertex Cover etc.

Wichtige Tatsache: Kann man für ein **NP**-vollständiges Problem zeigen, dass es in **P** liegt, so würde daraus folgen:

$$\mathbf{P} = \mathbf{NP}$$

Die Bibel der NP-Vollständigkeit

Das bisher wertbeständigste aller wissenschaftlichen Informatikbücher (seit 1979):