

# Problemreduktion

Idee: Gegeben Problemklasse A und Problemklasse B.

Wir wissen, wie wir ein Problem der Klasse A auf ein Problem der Klasse B abbilden können.

Wenn wir wissen, wie Probleme der Klasse B, gelöst werden können, können wir auch Probleme der Klasse A lösen.

Wenn wir wissen, welchen Aufwand Probleme der Klasse A mindestens haben, wissen wir, daß dies auch Minimum für Probleme der Klasse B ist.

# Problemreduktion

Problemreduktion kann das Problem auch verschlimmern:

- Paketversand über eine Hierarchische Struktur (erst mal alles zur Zentrale, von dort weiter verteilen)
- zum Spiegeleibraten erst mal in den Keller gehen

Oft ist erhöhter Aufwand durch Reduktion akzeptabel, weil

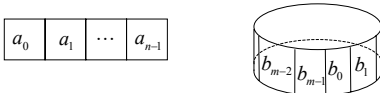
- Kapselung von Programmmodulen macht Programmieren schneller und weniger fehleranfällig (Black Box)
- zusätzlicher Aufwand ist vernachlässigbar klein

Aber

- zu viel Kapselung macht Programmcode oft träge (unnötige Parameterübergaben und -tests)

# Problemreduktion, String Matching

Aufgabe: Kommt Zeichenfolge  $A = a_0, a_1, \dots, a_{n-1}$  in der zyklischen Folge  $B = b_0, b_1, \dots, b_{m-1}$  vor?



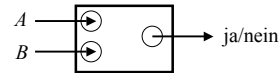
Standard: prüfe für alle  $0 \leq k \leq n-1$ , ob  $\forall_{i=0}^{n-1} a_i = b_{(k+i) \bmod m}$

Was, wenn Algorithmus für nicht-zyklische Folgen

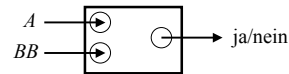
bereits vorhanden:  $a_0, a_1, \dots, a_{n-1}, b_0, b_1, \dots, b_{m-1}$

# Problemreduktion, String Matching

Algorithmus für  $A = a_0, a_1, \dots, a_{n-1}$  in  $B = b_0, b_1, \dots, b_{m-1}$  bekannt:



Dann ist Lösung für  $A = a_0, a_1, \dots, a_{n-1}$  in  $B = \dots, b_{m-2}, b_{m-1}, b_0, b_1, \dots, b_{m-1}, b_0, \dots$



# Komplexitätsklassen

Problemreduktion macht Aussage über Zugehörigkeit zu Komplexitätsklassen

Komplexität hängt von zugrundegelegter Rechnerarchitektur ab:  
Feststellen ob Palindrom mit 1-Kopf-Touringmaschine in  $O(n^2)$  mit 2-Kopf-Touringmaschine in  $O(n)$

Man definiert grobe Klassen wie  $O(\text{Polynom})$ ,  $O(\text{Exponent})$   
dann ist Palindromprüfung stets in  $O(\text{Polynom})$  machbar (für jede sinnvolle Rechnerarchitektur)

# Bestimmung unterer Grenzen

Bestimmung unterer Grenzen für den Aufwand:

Es ist offensichtlich:

Wenn Problem A auf Problem B reduziert werden kann, und wenn Problem A einen Mindestaufwand  $O(T(n))$  hat, dann hat B auch diesen Mindestaufwand.

Wenn A zur Komplexitätsklasse  $K$  gehört und auf B reduziert werden kann, und die Reduktion nicht aufwendiger als  $K$  ist, dann gehört auch B in die Klasse  $K$

# Problemreduktion, Polygonerzeugung

Erzeugen eines echten Polygons aus Punkten:

Frage: Läßt sich Sortieren auf Polygonerzeugung reduzieren?

Wenn ja, dann hätten wir untere Schranke für Polygonaufwand, da untere Schranke für Sortieren bekannt ist.

(Man kann zeigen, daß unter bestimmten vernünftigen Bedingungen Sortieren mindestens  $O(n \log n)$  benötigt).

# Sortieren reduziert auf Polygonerzeugung

Behauptung: einzige Möglichkeit, mehrere Punkte auf Einheitskreis zu einem echten Polygon zu verbinden, ist, jeden Punkt mit seinem Nachbarn zu verbinden



Beweis: Werden zwei nicht benachbarte Punkte verbunden, trennt die Verbindung die anderen Punkte in zwei Teile, die nicht ohne Überkreuzung verbunden werden können. Mindestens eine Verbindung der Gruppen ist aber nötig.

# Sortieren reduziert auf Polygonerzeugung

Gegeben: zu sortierende Zahlen  $x_1, \dots, x_n$

Reduktion: bestimme Minimum und Maximum (in  $O(n)$ ),

berechne  $y_i = x_i \frac{2\pi - \varepsilon}{\max - \min}$



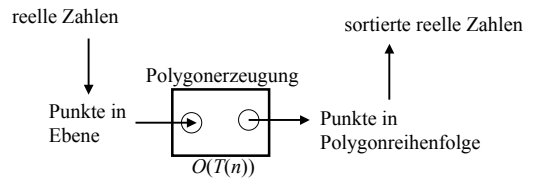
bestimme Punkte  $p_i$  auf Einheitskreis mit Winkel  $y_i$

berechne Polygon für Punkte  $p_i$

Sortierte Ausgabe der  $p_i$  entspricht Sortierung der  $x_i$

# Sortieren reduziert auf Polygonerzeugung

Fazit:



Bereits bewiesen: für Standard-1-CPU Rechner geht Sortieren nicht schneller als in  $O(n \log n)$

Also ist Komplexität für Polygonerzeugung  $O(n \log n)$

# Reduktion von Matrixoperationen Symmetrische Matrizen

Frage: Ist der Aufwand für Multiplikation symmetrischer Matrizen evtl. geringer?

$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix}$  benötigt 8 Produkte:

$\begin{bmatrix} a & b \\ b & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ f & h \end{bmatrix}$  benötigt 7 Produkte:

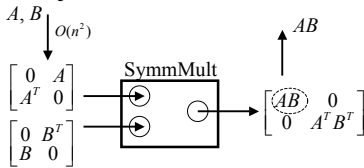
# Reduktion von Matrixoperationen Symmetrische Matrizen

Antwort: Nein, Aufwand für symmetrische Multiplikation ist so hoch wie für beliebige Multiplikation.

Beweis: Multiplikation der beliebigen Matrizen  $A, B$  läßt sich reduzieren auf symmetrische Multiplikation:

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}$$

# Symmetrische Matrizen



Also: Wenn ein Algorithmus **SymmMult** existiert, der in  $O(T(n))$  zwei symmetrische  $n \times n$  Matrizen multipliziert, und wenn  $T(2n) = O(T(n))$ , dann kann man beliebige Matrizen in  $O(T(n) + n^2)$  multiplizieren.

Bemerkung:  $T(2n) = O(T(n))$  gilt für jedes Polynom, und daß Matrixmultiplikation in polynomialer Zeit geht ist klar.

# Reduktion von Matrixoperationen Quadrieren

Noch eine interessante Frage:

Ist der Aufwand für das Quadrieren einer beliebigen Matrix signifikant kleiner als eine beliebige Multiplikation?

Antwort: Nein!  
Argumentation wie zuvor:  $\begin{bmatrix} 0 & A \\ B & 0 \end{bmatrix}^2 = \begin{bmatrix} AB & 0 \\ 0 & BA \end{bmatrix}$

**ACHTUNG:** Wir haben nicht gezeigt, daß symmetrische Multiplikation bzw. Quadrieren nicht schneller als in  $O(n^{2.81})$  (Strassen) geht!

Nur gezeigt: Wenn **SymmMult** oder **Quadrieren** in  $O(T(n))$  geht, dann auch beliebige Multiplikation!

# Rechnermodelle

Offensichtlich: Aufwand für Lösung eines Problems hängt von der Rechnerarchitektur ab.

Bemerkung: Zeitkomplexität (time)  
= wie viele Rechenschritte für Lösung

Platzkomplexität (space)  
= wie viele Speicherzellen werden benutzt

Zeit- und Platzkomplexität sind nicht unabhängig:  
Sortieren von  $a_1, a_2, \dots, a_n$  geht in  $O(n)$ ,  
wenn  $O(\max a_i)$  Speicher zur Verfügung steht

# Palindrome und Rechnermodelle

Beispiel für Abhängigkeit der Zeitkomplexität vom Rechnermodell

Aufgabe: gegeben Zeichenfolge  $A = a_1, a_2, \dots, a_n$   
gesucht  $x=1$  gdw.  $A$  ist Palindrom  
(d.h.  $\forall i \in \{1..n\} : a_i = a_{n+1-i}$ )

Algorithmus für von-Neumann-Rechner:

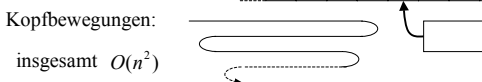
```
for i=1 to n
  if a[i] != a[n+1-i] return 0
return 1
```

Zeitaufwand ist  $O(n)$

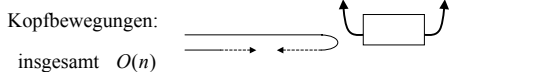
# Palindrome und Turing-Maschinen

Palindromprüfung mit Turing-Maschinen

1-Kopf Turing-Maschine:  $\dots \# a_1 a_2 \dots a_n \# \dots$



2-Kopf Turing-Maschine:  $\dots \# a_1 a_2 \dots a_n \# \dots$



insgesamt  $O(n)$

# Simulation von Rechnermodellen

Simulation von Rechnermodellen durch andere Modelle

Es gibt Algorithmus für 1-Kopf-Turing-Maschine, der Algorithmen für 2-Kopf-Turing-Maschine simulieren kann (Emulator).

Allgemein kann man für die meisten sinnvollen sequentiellen deterministischen Rechnermodelle zeigen:

Berechnet Modell A ein Problem in  $O(T(n))$ , dann kann Modell B das Problem in  $O(\text{Polynom}(T(n)))$  berechnen.

# Komplexität von Problemen

Begriff **Komplexität** läßt sich nicht nur auf *Algorithmen* sondern auch auf *Probleme* anwenden.

Komplexität eines Problems = Komplexität des „billigsten“ Algorithmus, der es löst

Oft wird dabei Komplexität als  $n$ -Tupel angesehen:

(Zeitkomplexität, Platzkomplexität, Hardwarekomplexität, ...)

Uns interessiert erst mal nur die Zeitkomplexität.

# verschiedene Komplexitätsklassen

Superfeine Einteilung:  $T(n)$  Operationen  
Feine Einteilung:  $O(T(n))$  Operationen

Grobe Einteilung: Polynom( $n$ ) Operationen  
oder Exponent( $n$ ) Operationen

Grobe Einteilung ist sinnvoll, wenn von Rechnermodellen, die sich gegenseitig mit polynomialem Aufwand simulieren können, abstrahiert werden soll.

Dann gehört Problem „Palindromtest“ zur Klasse Polynom( $n$ ) (oder kurz **P**).

# Die Komplexitätsklasse P

Die Klasse **P**:

Probleme, für die es einen Algorithmus gibt mit Aufwand  $O(\text{Polynom}(n))$

Algorithmen, die in  $O(\text{Polynom}(n))$  laufen, heißen **effizient** (*efficient*).

Probleme, für die es einen effizienten Algorithmus gibt, heißen **effizient lösbar** oder **praktikabel** (*tractable*)

# Entscheidungsprobleme

Wenn Länge der Ausgabe eines Algorithmus im Aufwand berücksichtigt werden muß, kann das kompliziert werden,

daher: Beschränkung auf Probleme, die nur 1 (ja) oder 0 (nein) als Ausgabe erwarten (Entscheidungsprobleme)

Die meisten Probleme, lassen sich auf ein oder mehrere Entscheidungsprobleme reduzieren

Beispiel: ursprüngliches Problem: gesucht  $\max(a,b)$   
1/0-Darstellung: ist  $a = \max(a,b)$   
oder ist  $b = \max(a,b)$

# Algorithmus = Sprachenerkener

Wenn ein Algorithmus nur 0/1 ausgibt, dann ist er ein „Sprachenerkener“.

Er erkennt die Sprache  $L = \{W_1, W_2, \dots\}$ , für die gilt:

wenn  $W \in L$  eingegeben wird, wird als Ausgabe 1 produziert.

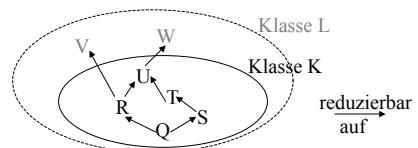
D.h. Probleme lassen sich als Sprachen darstellen.

z.B.  $L_{\text{Palindrom}} = \{0, 1, 00, 11, 000, 010, 101, 111, 0000, 0110, 1001, \dots\}$

# Definition von „härter“

Wenn Problem X mit „geringem“ Aufwand auf Problem Y reduzierbar ist, nennen wir das Problem Y **härter**.

(Y kann mindestens X, und noch mehr).



## Definition von „K-hart“

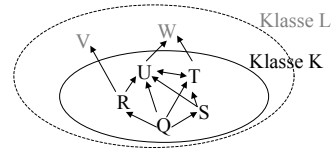
Wir wissen, wenn Problem A auf Problem B reduzierbar ist (mit vernachlässigbarem Aufwand), dann gehört B mindestens zur Komplexitätsklasse von A.

Für P heißt das: Wenn Problem A in P ist, und B läßt sich mit polynomialem Aufwand auf A reduzieren, dann ist B auch in P.

Man sagt: Ein Problem, auf das sich alle Probleme einer Klasse K reduzieren lassen, heißt **K-hart** (*K-hard*)

Wenn es selbst auch in K ist, dann ist es **K-vollständig** (*K-complete*)

## Begriffe K-hart, K-vollständig



Problem	Q	R	S	T	U	V	W
k-hart	-	-	-	✓	✓	-	✓
k-vollständig	-	-	-	✓	✓	-	-

## Problemreduktion = Sprachenübersetzung

Problem X entspricht Sprache  $L(X)$   
=> Problemreduktion auf Y entspricht Abbildung  $L(X) \rightarrow L(Y)$

Beispiel: Problem X: ist  $a_1$  Maximum von  $A = a_1, a_2, \dots, a_n$ ?  
Problem Y: ist  $b_1$  Minimum von  $B = b_1, b_2, \dots, b_n$ ?

$L(X) = \{0, 1, 00, 10, 11, 000, 100, 101, 110, 111, 0000, \dots\}$   
 $L(Y) = \{0, 1, 00, 01, 11, 000, 001, 010, 011, 111, 0000, \dots\}$

Abbildung  $L(X) \rightarrow L(Y)$ :  $f(a_1, \dots, a_n) = (1 - a_1, \dots, 1 - a_n)$   
offensichtlich:  $A \in L(X) \Leftrightarrow B = f(A) \in L(Y)$

## Sprache auf sich selbst reduziert

Man kann auch eine Sprache auf sich selbst reduzieren:

Beispiel: Problem X: ist  $A = a_1, a_2, \dots, a_n$  teilbar durch 3?

$L(X) = \{11, 110, 1001, 1100, 1111, 10010, 10101, \dots\}$

Abbildung  $L(X) \rightarrow L(X)$ :  
 $f(a_1, \dots, a_n) = \text{Quersumme}(a_1, \dots, a_n)$

offensichtlich:  $A \in L(X) \Leftrightarrow f(A) \in L(X)$

## polynomial reduzierbar/äquivalent

Wir nennen auch die Abbildung der Sprachen **Reduktion**.

Man sagt: Wenn es eine in polynomialer Zeit berechenbare Abbildung von  $L_1$  auf  $L_2$  gibt, dann heißt  $L_1$  **polynomial reduzierbar** auf  $L_2$ .

Wenn sowohl  $L_1$  auf  $L_2$  als auch  $L_2$  auf  $L_1$  polynomial reduzierbar ist, dann heißen  $L_1$  und  $L_2$  **polynomial äquivalent**.

Leicht zu sehen: „polynomial reduzierbar“ ist transitiv und reflexiv.

Folge aus „polynomial reduzierbar“:  
Wenn ein Problem X in der Komplexitätsklasse P liegt, und wenn Y auf ein Problem X polynomial reduzierbar ist, dann liegt Y auch in P.

## Nichtdeterminismus

```
FOR I=1 TO 100
  PRINT "
```

Nichtdeterministische Algorithmen im Sinne der Theorie sind **nicht** Algorithmen, deren Ausgang unvorhersagbar ist (wegen Zufallszahlengenerator oder so),

sondern Algorithmen, die eine Eingabe als zu einer Sprache gehörend akzeptieren, aufgrund der Auswertung mehrerer Programmabläufe.“

# Nichtdeterministische Algorithmen

Definition für nichtdeterministischen Algorithmus:

Verwendet „normale“ Anweisungen.

Zusätzlich gibt es eine „Verzweigungsanweisung“ (d.h. eine ja/nein Entscheidung)

Jede Verzweigung teilt einen Programmpfad in zwei auf.

Jeder Programmpfad liefert als Ergebnis 1 oder 0

Der Algorithmus gibt 0 aus, wenn alle Pfade 0 ausgeben und 1, wenn mindestens ein Pfad eine 1 ausgibt

# Nichtdeterminismus: Verzweigungsanweisungen

Interpretation von Verzweigungsanweisungen:

- es genügt, nur ja/nein (bzw. 0/1) Entscheidungen zu verwenden
- wir befragen ein Orakel, wo es lang geht
- wir haben unendlich viele Prozessoren, und lassen bei jeder Abzweigung einen neuen Prozessor mitrechnen
- eine Turing-Maschine hat für einen Zustand mehrere Nachfolgezustände (geht auch für andere Rechnermodelle)

# Beispiel für Nichtdeterminismus

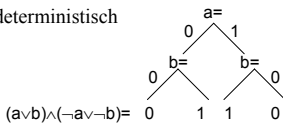
Beispiel: Problem: gibt es für die Formel  $(a \vee b) \wedge (\neg a \vee \neg b)$  eine erfüllende Belegung der Variablen?

Deterministisch:

```
for a=0 to 1
  for b=0 to 1
    if  $(a \vee b) \wedge (\neg a \vee \neg b)$  return 1
  return 0
```

Aufwand für  $n$  Variablen =  $O(2^n)$

Nichtdeterministisch



Aufwand nur für eine Auswertung einer Formel mit  $n$  Variablen =  $O(n)$

# Eine Interpretation des Nichtdeterminismus

Interpretation Nichtdeterministischer Algorithmen:

Gesucht ein Beweis für eine Behauptung

deterministisch: finde einen Beweis  
nichtdeterministisch: prüfe ob gegebener Beweis korrekt ist

Gesucht Belegung für Variablen

deterministisch: finde passende Belegung  
nichtdeterministisch: prüfe ob gegebene Belegung paßt.

# Laufzeit nichtdeterministischer Algorithmen

Da es zum Akzeptieren (d.h. 1 Ausgeben) genügt, wenn ein Berechnungspfad eine 1 liefert, kann die Berechnung abgebrochen werden, sobald dies geschieht.

D.h. wenn eine Eingabe zur akzeptierten Sprache gehört ist die Gesamtlaufzeit die Laufzeit des kürzesten Pfades, der eine 1 liefert.

Diese Aussage kann man nicht für Eingaben machen, die nicht zur akzeptierten Sprache gehören.

# Die Komplexitätsklasse NP

Bereits bekannt: Klasse P der Probleme, für die es einen polynomialen Algorithmus gibt.

Definition der Klasse **NP**: Menge aller Probleme, für die es einen nichtdeterministischen Algorithmus mit polynomialen Aufwand gibt.

Offensichtlich gilt:  $P \subseteq NP$



Ruhm und Ehre für den, der beweist, daß  $P \subset NP$

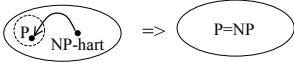
# NP-hart / NP-vollständig

Die Begriffe K-hart und K-vollständig gibt es natürlich auch für NP:

Ein Problem X ist **NP-hart**, wenn man jedes Problem aus NP auf X reduzieren kann.

Ein Problem X ist **NP-vollständig**, wenn es NP-hart ist und wenn es zu NP gehört.

Folge: Wenn jemals ein polynomialer Algorithmus für ein NP-hartes Problem gefunden wird, dann wäre das der Beweis für  $P=NP$ .



# NP-Vollständigkeit

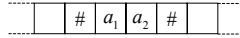
Wie identifiziert man ein Problem X als NP-vollständig?

- entweder „zu Fuß“ beweisen, daß  $X \in NP$  und alle  $Y \in NP$  polynomial reduzierbar auf X
- oder beweisen, daß  $X \in NP$  und daß ein bekanntes NP-vollständiges Problem Y auf X polynomial reduzierbar ist.

Also Henne-Ei-Problem: Wer gibt uns einen ersten „zu Fuß“-Beweis, damit weitere NP-vollständige Probleme leichter gefunden werden?

# Beweis von Cook

Betrachten wir Turing-Maschine, die prüfen soll, ob in einer Eingabe  $A = (a_1, a_2)$  gilt  $a_1 \leq a_2$



Diese Bedingung läßt sich schreiben als

$$(a_1 = 0 \wedge a_2 = 0) \vee (a_1 = 0 \wedge a_2 = 1) \vee (a_1 = 1 \wedge a_2 = 1)$$

oder  $(\neg a_1 \wedge \neg a_2) \vee (\neg a_1 \wedge a_2) \vee (a_1 \wedge a_2)$

Es ist möglich (Beweis von Cook, 1971), für jeden Algorithmus eine aussagenlogische Formel anzugeben, die genau dann erfüllbar ist, wenn die zugehörige Turing-Maschine in einem akzeptierenden Zustand terminiert.

# NP-vollständiges Problem gesucht

In anderen Worten:

Alle Operationen und Zustände einer Turing-Maschine lassen sich in Aussagenlogik darstellen.

D.h. jedes Problem, das eine Turing-Maschine lösen kann, läßt sich auf das Finden einer erfüllenden Belegung für eine aussagenlogische Formel abbilden.

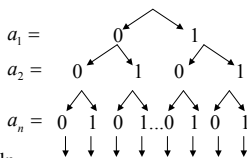
Letzteres Problem nennen wir **SAT** (*satisfiability problem*).

# Ist SAT ein NP-vollständiges Problem

SAT ist NP-vollständig weil:

SAT ist **NP-hart** (jeder Turing-Maschinenalgorithmus läßt sich darauf reduzieren)

SAT ist  $\in NP$  nichtdeterministischer Algorithmus:



Auswertung der  $2^n$  Formeln in jeweils  $O(n)$

# SAT und Normalformen

Normalformen

sei die Formel  $a$  xor  $b$  gegeben

Disjunktive Normalform:  $(a \wedge \neg b) \vee (\neg a \wedge b)$

Konjunktive Normalform:  $(a \vee b) \wedge (\neg a \vee \neg b)$

Klausel

Finden einer Belegung für eine Formel in DNF ist trivial. Finden einer Belegung in CNF ebenso wie die Umwandlung von CNF nach DNF ist sehr aufwendig.

Daher wird SAT meist nur für CNF betrachtet. Es ist so, daß die Eigenschaft „Turing-Maschine“ terminiert akzeptierend mit polynomialem Aufwand als CNF dargestellt werden kann.

# SAT und 3-SAT

## Das 3-SAT-Problem

Gegeben eine Formel in CNF, wobei jede Klausel aus genau drei Variablen (bzw. Negaten) besteht.

Gesucht eine erfüllende Belegung der Variablen.

3-SAT sieht einfacher aus als SAT-allgemein. Ist es das wirklich?

Wenn nicht, dann:

- 3-SAT auch NP-vollständig
- 3-SAT eingeschränkteres Standardproblem für Reduktionen (evtl. einfachere Reduktion)

# Reduktion von SAT auf 3-SAT

Kann man SAT auf 3-SAT reduzieren? Ja, so:

Sei eine Formel gegeben in CNF (o.B.d.A. ohne Negationen)

$$E = (\dots \vee \dots) \wedge \dots \wedge (x_1 \vee x_2 \vee \dots \vee x_k) \wedge \dots \wedge (\dots \vee \dots)$$

Wir wollen die Klausel  $C = (x_1 \vee x_2 \vee \dots \vee x_k)$  schreiben als Konjunktion  $C'$  mehrerer 3er-Klauseln mit den Variablen  $x_1, \dots, x_k, y_1, \dots, y_{k-3}$  so daß gilt:

- wenn eine Belegung von  $x_1, \dots, x_k$   $C$  erfüllt, dann gibt es eine erweiternde Belegung von  $y_1, \dots, y_{k-3}$ , die  $C'$  erfüllt.
- wenn eine Belegung von  $x_1, \dots, x_k$   $C$  nicht erfüllt, dann gibt es auch keine Belegung, die  $C'$  erfüllt.

# Reduktion von SAT auf 3-SAT

$$C = (x_1 \vee x_2 \vee \dots \vee x_k)$$

$C' =$

$$(x_1 \vee x_2 \vee y_1) \wedge (x_3 \vee \neg y_1 \vee y_2) \wedge (x_4 \vee \neg y_2 \vee y_3) \wedge \dots \wedge (x_{k-2} \vee \neg y_{k-4} \vee y_{k-3}) \wedge (x_{k-1} \vee x_k \vee \neg y_{k-3})$$

x	1	+				
	2	+				
	3		+			
	4			+		
	5				+	
	6					+
	7					

y	1	+	-			
	2		+	-		
	3			+	-	
	4					+

Wenn alle  $x_i = 0$ , dann ist  $C' = (y_1) \wedge (\neg y_1 \vee y_2) \wedge (\neg y_2 \vee y_3) \wedge \dots \wedge (\neg y_{k-4} \vee y_{k-3}) \wedge (\neg y_{k-3})$  offensichtlich nicht erfüllbar.

Wenn ein  $x_i = 1$ , dann setze  $y_1 \dots y_{i-2} = 1$  und die anderen auf 0

$C$  hat nur 1 oder 2 Variablen => einfache Sonderfälle

# NP-vollständige Probleme

• Lassen sich die Knoten eines gegebenen Graphen so mit 3 Farben einfärben, daß keine benachbarten Knoten dieselbe Farbe haben?

• Gegeben  $X = \{x_1, x_2, \dots, x_k\}$ ,  $s(x_i)$ ,  $v(x_i)$

Gesucht  $B \subseteq X$  mit  $\sum_{x_i \in B} s(x_i) \leq s$  und  $\sum_{x_i \in B} v(x_i) \geq v$  (Knapsack)

• u.v.a.

# Umgang mit NP-vollst. Problemen

Oft kommt eine tatsächliche Berechnung aller exponentiell vielen Berechnungswege nicht in Frage.

(Man mache sich klar, daß eine 1000 GHz CPU seit Beginn des Universums gerade mal  $2^{100}$  Taktzyklen rechnen konnte.)

Daher versucht man oft Näherungslösungen zu finden:

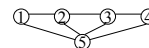
- Algorithmus funktioniert nur für bestimmte Eingaben perfekt
- gefundene Lösungen sind nicht optimal
- Algorithmus läuft im Durchschnitt „schnell“ aber nicht immer
- Algorithmus findet manchmal eine, manchmal keine Lösung
- Algorithmus ist schnell genug für kurze Eingaben

# Umgang mit NP-vollst. Problemen Backtracking

Backtracking

Aufgabe: Färbe Knoten eines Graphen mit 3 Farben ein, so daß keine zwei benachbarten Knoten dieselbe Farbe haben.

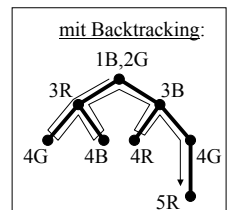
z.B.:



Extrem dummer Algorithmus:

für alle  $(f_1, \dots, f_n) \in \{R, G, B\}^n$   
wenn Färbung OK  
gib Lösung aus

(243 Schritte)



(11 Schritte)

# Umgang mit NP-vollst. Problemen

## Backtracking

Backtracking kann Aufwand drastisch reduzieren, aber

- keine Garantie für Aufwandsreduktion
- d.h. Worst-case-Verhalten weiterhin exponentiell

Eine Lösung wird immer gefunden.

Die gefundene Lösung ist eine korrekte Lösung  
(keine Näherungslösung).

# Umgang mit NP-vollst. Problemen

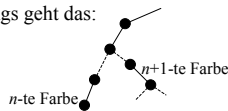
## Branch-and-Bound

Branch-and-Bound

Aufgabe: Färbe Knoten eines Graphen mit  $n$  Farben ein, so daß keine zwei benachbarten Knoten dieselbe Farbe haben und  $n$  minimal ist.

Backtracking geht jetzt nicht mehr so, da beliebig viele Farben wählbar.

Allerdings geht das:



Wenn Farbensumme auf einem Pfad größer als bisheriges Minimum (bound), dann setze zurück.

# Umgang mit NP-vollst. Problemen

## Näherungslösungen

Algorithmen für Näherungslösungen.

Aufgabe: Gegeben  $n$  Orte als  $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$   
Gesucht: Permutation  $(i_1, i_2, \dots, i_n, i_{n+1} = i_1)$  so, daß

$$\sum_{j=1}^n \sqrt{(x_{i_j} - x_{i_{j+1}})^2 + (y_{i_j} - y_{i_{j+1}})^2} \leq k$$

**Extrem dummer Algorithmus:**

für alle  $(i_1, i_2, \dots, i_n)$   
wenn  $\Sigma \leq k$   
gib Lösung aus

( $n!$  Schritte)

