# Grundlagen Theoretischer Informatik 3

## SoSe 2012 in Trier

Henning Fernau
Universität Trier
fernau@uni-trier.de

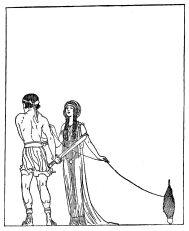
## Exaktes Lösen *NP*-harter Probleme

- In manchen Anwendungen ist das garantierte Auffinden exakter Lösungen unabdingbar.

- Das bedeutet, dass man (theoretisch) mit nicht-polynomiellen Algorithmen auskommen muss.

- Auch hiervon kann es “mehr oder weniger gute” geben.
  Dies haben wir bereits beim Abschnitt über Dynamisches Programmieren gesehen.

## **Backtracking**: Mit dem Ariadnefaden durchs Labyrinth



- Durchlauf durch (großen) Suchbaum
- Baum i.d.R. nur implizit gegeben
- wichtiges Teilziel: Ausschluss von Teilbäumen
- i.d.R. rekursive Implementierung

Grundstruktur der Rekursion:

```
PROCEDURE backtrack(Lösungsansatz);
  IF (Lösungsansatz == vollständige Lösung)
    THEN gib Lösung aus
    ELSE FOR (wichtige Erweiterungen des Lösungsansatzes)
             backtrack(Lösungsansatz mit Erweiterung)
         ENDFOR
  ENDIF
  RETURN
ENDPROCEDURE
```

Aufruf mit `backtrack(leere Lösung)`

## Bemerkungen zu Backtracking

- Rekursion führt implizit zu Baumstruktur!
- Anzahl der Erweiterungen der Lösungsansätze klein halten!
- zur ‘Wichtigkeit’ der Erweiterungen:
  - Jede Erweiterung, die noch zu Lösung werden könnte, ist wichtig!
  - Leicht erkennbar, dass keine Lösung mehr entstehen kann: unwichtige Erweiterung
  - ansonsten: vorsichtshalber als wichtige Erweiterung einstufen...
- falls keine Erweiterung wichtig $\leadsto$ Teilbaum wird ausgelassen!

## Beispiel KNF-SAT:

- Gegeben: Formel $F$ in KNF mit $n$ Variablen
- Gesucht: Erfüllende Belegung $\Phi$

Naiver Algorithmus: Durchsuche alle möglichen Belegungen, z.B.:

```
Gegeben F mit n Variablen
FOR b := 0 TO power(2,n) - 1 DO
    Interpretiere die Einzelbits von b als Belegung Φ.
    Teste, ob F durch ϕ erfüllt wird.
    Wenn ja: RETURN TRUE
RETURN FALSE
```

Zeitaufwandsabschätzungsüberlegungen:

- Formel nicht erfüllbar: $2^n \cdot$ (Bestimmung des Wertes von $F$ bei $\phi$)
- Formel erfüllbar: Zeitaufwand schrumpft mit wachsender Zahl an Lösungen...

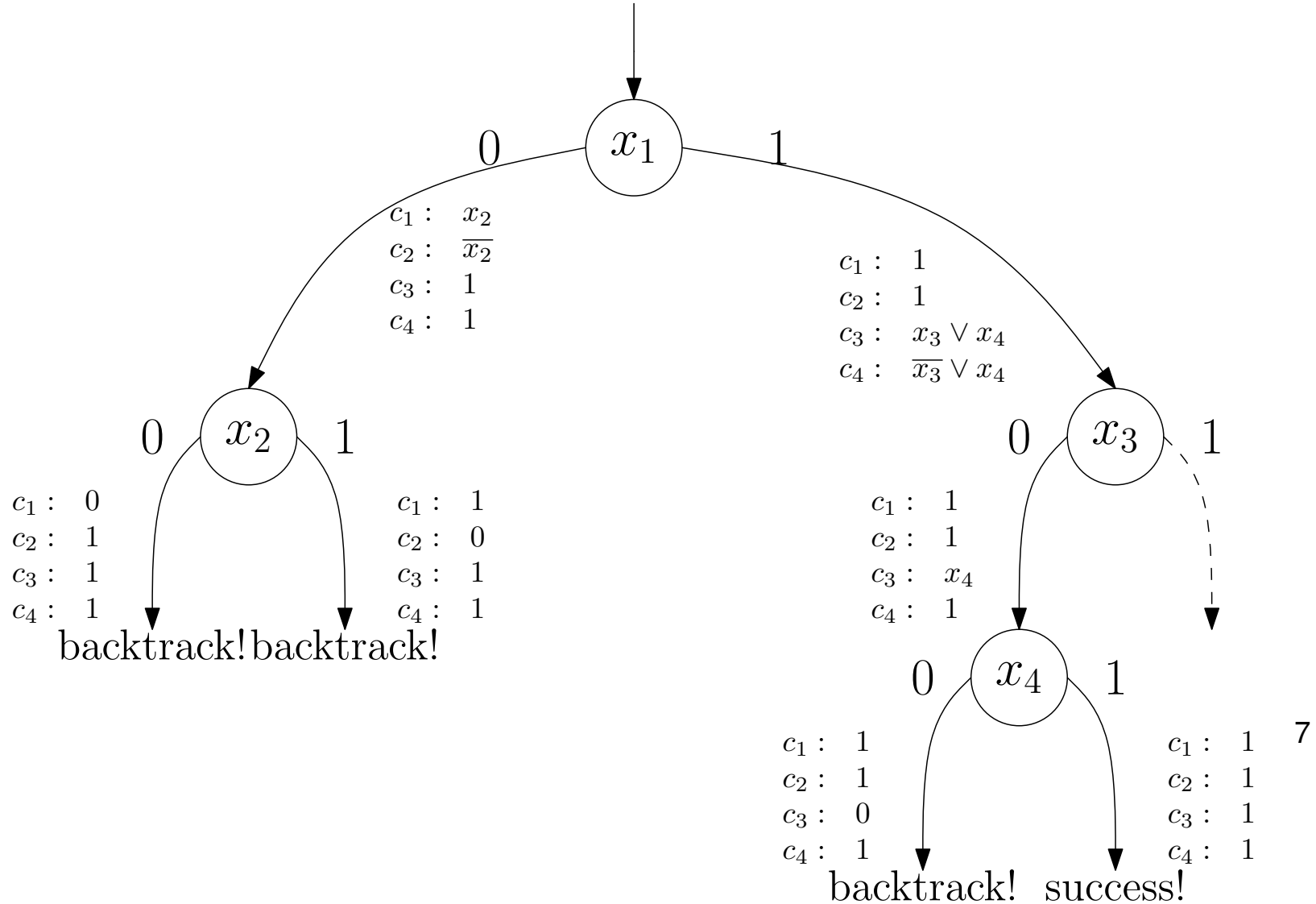
## **Backtracking-Algorithmus (einfachste Form)**:

Gegeben: Formel $F$ mit Variablen $\{x_1, ..., x_n\}$
Belegung $\Phi$ ist String der Länge $n$ über $\{0, 1, u\}$ (mit 0: falsch, 1: wahr, $u$: unbelegt)

```
BOOLEAN PROCEDURE backtrack(belegung Φ;
  IF (Φ enthält kein u)
    THEN RETURN F(Φ)
  ENDIF
  IF (Φ setzt eine Klausel in F auf 0 )
    THEN RETURN 0
  ENDIF
  wähle unbelegte Variable, Index sei i
  IF  backtrack( Φ[i:0] )
    THEN RETURN 1
  ENDIF
  RETURN backtrack( Φ[i:1] )
ENDPROCEDURE

main:
  backtrack(  u...u )
```

Beispiel: $F = \underbrace{(x_1 \vee x_2)}_{c_1} \wedge \underbrace{(x_1 \vee \overline{x_2})}_{c_2} \wedge \underbrace{(\overline{x_1} \vee x_3 \vee x_4)}_{c_3} \wedge \underbrace{(\overline{x_1} \vee \overline{x_3} \vee x_4)}_{c_4}$



## **Backtracking bei 2-KNF-SAT**:

- Pro Klausel von 4 Wahrheitswerten nur 3 wichtig
- Formal damit statt $2^n$ nur $(\sqrt{3})^n$
- Noch bessere Abschätzung:
  Falls ein Literal falsch, muss das andere wahr werden, also Rekursion
  $T(n) \leq T(n-1) + T(n-2)$ für Suchbaumgröße;
  Erinnerung: Fibonacci-Rekursion
- zudem: jede Variablen-Belegung legt Werte sehr vieler direkt oder auch indirekt verbundener Variablen fest!
- Damit nur noch 2 Versuche pro 'Zusammenhangskomponente'!
- Topologische Sortierung der Komponenten
  $\leadsto$ Polynomialer Algorithmus (bei Modifikation des Backtracking) [Alternative s.o.]

## Backtracking bei 3-KNF-SAT:

- Statt $2^n$ nur $\approx 7^{n/3} = 1{,}913^n$,
- formal: bei $n = 30$ nur 1/4 der Komplexität
- Hauptvorteil aber: Jede Variablenbelegung macht aus manchen 3er-Klauseln kleinere 2er-Klauseln, d.h. dieser Teil dann wie 2-KNF-SAT behandelbar!
- Vielleicht bessere Abschätzung:
  $T(n) \leq 2T(n-2) + 2T(n-3)$ (noch viel bessere sind möglich!)
  (Für Klausel $C$ mit Variablen $x, y, z$ analysiere sowohl für $x = 0$ und $x = 1$ die 2-Klausel mit $y, z$ wie vorher.)
  Mit $T(1) = c_1$, $T(2) = c_2$ und $T(3) = c_3$ folgt bei angenommener Gleichheit (schlimmster Fall):
  $T(n) = T(n-1) + T(n-2) + T(n-3) + [n = 1]c_1 + [n = 2](c_2 - c_1) + [n = 3](c_3 - c_2 - c_1)$.
  Das könnte man wieder mit einer $z$-Transformation lösen...
  Daraus würde dann folgen: $T(n) \in \mathcal{O}(1{,}84^n)$.

**Branch and Bound**: Kombination zweier Strategien bei *Optimierungsproblemen*:

- Aufgabe: suche beste von allen(!) Lösungen
- wie bei Backtracking: Ausschluss von Lösungsansätzen
- zudem: bevorzugte Behandlung aussichtsreicher Lösungsansätze

Dazu benötigt:

- Optimierungsproblem mit baumartigem Lösungsalgorithmus
- einfache Bewertungsmöglichkeit $g$ der Güte eines Lösungsansatzes
- evtl. (möglichst gute) Schranke $G$ für die beste Gesamtlösung

Vergleich:

- Backtracking: Suche mit ‘nächster’ Teil-Lösung fortgesetzt
- Branch and Bound: Suche mit ‘bestem’ Lösungsansatz fortgesetzt
- statt Rekursionsstack wird Priority Queue verwendet

## **Branch-and-Bound-Algorithmus (einfachste Form)**:

- Gegeben: Mimimierungsproblem
  - mit Bewertungsfunktion $\nu$ für Lösungen,
  - mit Gütefunktion $g$ ('Bound') für Lösungsansätze
- Bedingung an $\nu$ und $g$:
  Ist Lösung $L$ Erweiterung des Ansatzes $A$, dann $\nu(L) \geq g(A)$

```
Ansatz      A := leerer Lösungsansatz
Schranke    G := ∞
Lösung      L
```

```
Priority Queue pq; pq.insert( A, g(A) )

WHILE pq.not_empty DO
A := pq.extract_min;
  IF (A ist vollständige Lösung)  THEN
     IF ( v(A) < G ) THEN L := A; G := v(A) ENDIF
  ELSE
     FORALL ( E Erweiterung von A ) DO
        IF ( g(E) < G ) THEN  pq.insert( E, g(E) ) ENDIF
     ENDFOR
  ENDIF
ENDDO
RETURN L
```

## Beispiel: TSP

- $m_{i,j}$ Kosten/Gewicht der Kante $(ij)$
- $g$ Gewicht eines Pfades $i_1 i_2 ... i_n$

$$g(i_1 i_2 ... i_n) = m_{i_1,i_2} + ... + m_{i_{n-1},i_n}$$

- $\nu$ Gewicht einer kompletten Rundreise $i_1 i_2 ... i_n$ mit $i_1 = i_n$

$$\nu(i_1 i_2 ... i_n) = g(i_1 ... i_n)$$

- bei nichtnegativen(!) Kosten $m_{i,j}$ gilt für $k \leq n$

$$\nu(i_1 i_2 ... i_n) \geq g(i_1 i_2 ... i_k)$$

Betrachte TSP mit $M = \begin{bmatrix} - & 10 & 15 & 20 \\ 5 & - & 9 & 10 \\ 6 & 13 & - & 12 \\ 8 & 8 & 9 & - \end{bmatrix}$, Start mit $G = \infty$

| Inhalt (Pfad,Gewicht) der Priority Queue: |
|---|
| $(1, 0)$ |
| $(12, 10)$ $(13, 15)$ $(14, 20)$ |
| $(13, 15)$ $(123, 19)$ $(124, 20)$ $(14, 20)$ |
| $(123, 19)$ $(124, 20)$ $(14, 20)$ $(134, 27)$ $(132, 28)$ |
| $(124, 20)$ $(14, 20)$ $(134, 27)$ $(132, 28)$ $(1234, 31)$ |
| $(14, 20)$ $(134, 27)$ $(132, 28)$ $(1243, 29)$ $(1234, 31)$ |
| $(134, 27)$ $(132, 28)$ $(142, 28)$ $(143, 29)$ $(1243, 29)$ $(1234, 31)$ |
| $(132, 28)$ $(142, 28)$ $(143, 29)$ $(1243, 29)$ $(1234, 31)$ $(1342, 35)$ |
| $(142, 28)$ $(143, 29)$ $(1243, 29)$ $(1234, 31)$ $(1342, 35)$ $(1324, 38)$ |
| $(143, 29)$ $(1243, 29)$ $(1234, 31)$ $(1342, 35)$ $(1423, 37)$ $(1324, 38)$ |
| $(1243, 29)$ $(1234, 31)$ $(1342, 35)$ $(1423, 37)$ $(1324, 38)$ $(1432, 42)$ |
| erste Lösung gefunden: $L = (12431)$, neue Schranke $G = 35$<br>$(1234, 31)$ $(1342, 35)$ $(1423, 37)$ $(1324, 38)$ $(1432, 42)$<br>Tatsächlich zufällig bereits das Optimum! |
| ab hier: $\mathrm{pq}$ wird nur noch geleert... |

**Verbesserungen des Algorithmus**:

(1) Suche besseren Startwert der Schranke $G$

- z.B. durch Vorab-Bestimmung einer (nicht zu schlechten) Lösung

(2) Bessere Datenstrukturen:
Für TSP statt Pfad besser Paare (Knotenmenge, letzter Knoten)

- z.B. statt $(1243, 29)$ und $(1423, 37)$ nur $(\ (\{2,4\},3)\ , 29)$ in `pq`
- benötigt `decreaseKey`-Operation für Priority Queues

(3) Versuche die Unterschiede zwischen guten und schlechten Pfaden zu vergrößern...

- Reduziere Gewichte der Pfade um Anteile, die allen Pfaden gemeinsam sind
- von Reduktion profitieren insbesondere gute Lösungen

**Anwendung von (3)** im Beispiel TSP: Reduktion von $M$ wie folgt

- Subtrahiere von jeder Zeile in $M$ das Minimum $z_i$ der Zeile $i$,
  damit wird Gewicht jeder(!) Lösung um $\sum k_i$ erniedrigt.

$$M = \begin{bmatrix} - & 10 & 15 & 20 \\ 5 & - & 9 & 10 \\ 6 & 13 & - & 12 \\ 8 & 8 & 9 & - \end{bmatrix} \quad \rightsquigarrow \quad M' = \begin{bmatrix} - & 0 & 5 & 10 \\ 0 & - & 4 & 5 \\ 0 & 7 & - & 6 \\ 0 & 0 & 1 & - \end{bmatrix}$$

mit Zeilen-Minima $10, 5, 6, 8$, d.h. mit Erniedrigung um $10+5+6+8 = 29$.

- Subtrahiere von jeder Spalte in $M'$ das Minimum $s_i$ der Spalte $i$,
  damit wird Gewicht jeder(!) Lösung weiter um $\sum s_i$ erniedrigt.

$$M' = \begin{bmatrix} - & 0 & 5 & 10 \\ 0 & - & 4 & 5 \\ 0 & 7 & - & 6 \\ 0 & 0 & 1 & - \end{bmatrix} \quad \rightsquigarrow \quad M'' = \begin{bmatrix} - & 0 & 4 & 5 \\ 0 & - & 3 & 0 \\ 0 & 7 & - & 1 \\ 0 & 0 & 0 & - \end{bmatrix}$$

mit Erniedrigung um $0+0+1+5 = 6$.

- Optimale Lösungen bei $M$ und $M''$ gleich,
  aber im Gewicht verschieden, sogar um $29 + 6 = 35$.

**Anwendung von (3)** im Beispiel TSP:

Betrachte TSP mit $M'' = \begin{bmatrix} - & 0 & 4 & 5 \\ 0 & - & 3 & 0 \\ 0 & 7 & - & 1 \\ 0 & 0 & 0 & - \end{bmatrix}$, Start mit $G = \infty$

| Inhalt (Pfad,Gewicht) der Priority Queue: |
|---|
| $(1, 0)$ |
| $(12, 0)$ $(13, 4)$ $(14, 5)$ |
| $(124, 0)$ $(123, 3)$ $(13, 4)$ $(14, 5)$ |
| $(1243, 0)$ $(123, 3)$ $(13, 4)$ $(14, 5)$ |
| erste Lösung gefunden: $L = (12431)$, neue Schranke $G = 0$<br>$(123, 3)$ $(13, 4)$ $(14, 5)$ |
| ab hier: $pq$ wird nur noch geleert... |

Optimale Lösung für $M''$ ist $L = (12431)$ mit Gewicht $0$, damit wieder:

Optimale Lösung für $M$ ist $L = (12431)$ mit Gewicht $35 + 0 = 35$

## Bestimmung guter Schrankenfunktionen:

- Gegeben: Mimimierungsproblem $P$
  - mit Bewertungsfunktion $v$ für Lösungen,
  - benötigt: Gütefunktion $g$ ('Bound') für Lösungsansätze
- bestimme $g$ mit:
  Ist Lösung $L$ Erweiterung des Ansatzes $A$, dann $v(L) \geq g(A)$.

Oft wird dazu **Relaxation** benutzt:

- Aufweichung der Aufgabenstellung von $P$ zu $P'$,
  z.B. Weglassen eigentlich notwendiger Bedingungen
- $\leadsto$ Lösungsraum von $P'$ umfasst Lösungsraum von $P$
- evtl. effizienter Algorithmus für aufgeweichtes Problem $P'$ bekannt
- Gütefunktion $g'$ für $P'$ ist auch Gütefunktion $g$ für $P$...

Hinweis: Auch TSP-Beispiel als Relaxation interpretierbar: Betrachte zyklenfreie Pfade als Lösung (statt Permutationen); dann: beste Lösung zu Ansatz $i_1...i_k$ ist gerade $i_1...i_k$ selbst.

## Lineares Programmieren (aus Wikipedia)

Bei einem *linearen Programm* (LP) sind eine Matrix $A \in \mathbb{R}^{m,n}$ und zwei Vektoren $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^n$ gegeben. Eine *zulässige Lösung* ist ein Vektor $x \in \mathbb{R}^n$ mit nichtnegativen Einträgen, der die linearen Bedingungen

$$\begin{array}{cccc} a_{11}x_1 & +\ldots & +a_{1n}x_n & \leq b_1 \\ a_{21}x_1 & +\ldots & +a_{2n}x_n & \leq b_2 \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1}x_1 & +\ldots & +a_{mn}x_n & \leq b_m \end{array}$$

erfüllt. Ziel ist es, unter allen zulässigen Vektoren x einen zu finden, der das Skalarprodukt

$$c^T x = c_1x_1 + \ldots + c_nx_n$$

maximiert. Dieses Optimierungsproblem in der sogenannten Standardform wird oft abkürzend als $\max\{c^T x \mid Ax \leq b, x \geq 0\}$ geschrieben, wobei die Bedingungen $Ax \leq b$ und $x \geq 0$ komponentenweise zu verstehen sind.

## Lineares Programmieren

Ein Maximierungs-Problem der Form

$$\max\{c^T x \mid Ax \leq b, x \geq 0\}$$

ist offenbar äquivalent zu einem Minimierungsproblem der Form

$$\min\{d^T x \mid Ex \geq f, x \geq 0\}$$

Mitteilung Lineare Programme lassen sich in Polynomialzeit lösen!
Viele praktisch relevante Probleme lassen sich so darstellen.
In eben solchen praktischen Fällen ist der im schlimmsten Fall exponentielle Simplex-Algorithmus häufig bis heute der schnellste aller “einfachen Verfahren”.
Es gibt sehr gute professionelle Software zum Lösen solcher Aufgaben.

## Beispiel aus der Produktionsplanung (zweidimensional)

Eine Firma stellt zwei verschiedene Produkte her, für deren Fertigung drei Maschinen A, B, C zur Verfügung stehen.
Diese Maschinen haben eine maximale monatliche
Laufzeit von 170 Stunden (A), 150 Stunden (B) bzw. 180 Stunden (C).
Eine Mengeneinheit (ME) von Produkt 1 liefert einen Deckungsbeitrag von 300 Euro,
eine ME von Produkt 2 dagegen 500 Euro.
Fertigt man eine ME von Produkt 1, dann benötigt man dafür eine Stunde die Maschine A und eine Stunde die Maschine B.
Eine Einheit von Produkt 2 belegt zwei Stunden lang Maschine A, eine Stunde Maschine B und drei Stunden Maschine C.
Ziel ist es, Produktionsmengen zu bestimmen, die den Deckungsbeitrag der Firma maximieren, ohne die Maschinenkapazitäten zu überschreiten.
Fixkosten können in dem Optimierungsproblem ignoriert und anschließend dazuaddiert werden, da sie per Definition unabhängig von den zu bestimmenden Produktionsmengen sind.

## Mathematische Modellierung

Angenommen, der Betrieb fertigt pro Monat $x_1$ ME von Produkt 1 und $x_2$ ME von Produkt 2. Dann beträgt der Gesamtdeckungsbeitrag

$G(x_1, x_2) = 300x_1 + 500x_2$.

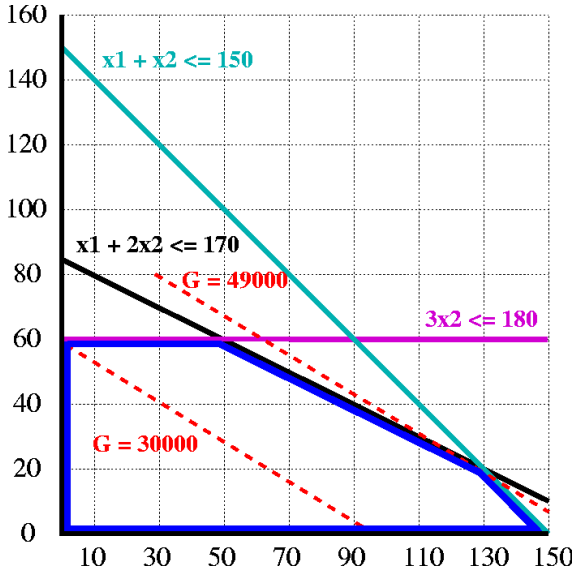
Diesen Wert möchte die Firma maximieren.
Da die Maschinenkapazitäten eingehalten werden müssen, ergeben sich die Nebenbedingungen:

$$x_1 + 2x_2 \leq 170 \text{ (Maschine A, rechts schwarz dargestellt)} \quad (1)$$

$$x_1 + x_2 \leq 150 \text{ (Maschine B, rechts türkis dargestellt)} \quad (2)$$

$$3x_2 \leq 180 \text{ (Maschine C, rechts violett dargestellt)} \quad (3)$$

Da außerdem keine negativen Produktionsmengen möglich sind, muss $x_1, x_2 \geq 0$ gelten.




$G = 49000$ zeigt (unabhängig von den Nebenbedingungen) Möglichkeiten auf, 49000 Euro Gewinn zu erzielen.

## Das Knotenüberdeckungsproblem als Lineares Programm

Ordne jedem Knoten $v_i$ im Graphen eine Variable $x_i$ zu.
Diese Variable soll den Wert 1 annehmen, wenn der zugehörige Knoten in die Überdeckung aufgenommen wird, und sonst bekommt sie den Wert 0.
Das Abdecken der Kante $\{v_i, v_j\}$ können wir jetzt ausdrücken durch:
$x_i + x_j \geq 1$.
Also erhalten wir zu einem Graphen $G = (V, E)$ als VC-Instanz ein LP der Form:
$\min\{\sum x_i \mid \forall\{v_i, v_j\} \in E : x_i + x_j \geq 1\}$.

Welche Probleme würde uns eine Lösung eines LP-Lösers bescheren?

## Ganzzahliges Lineares Programmieren

Das folgende (hier in Maximierungsform formulierte) Problem ILP (Integer Linear Programming) ist aufgrund der bekannten $\mathrm{NP}$-Härte von VC ebenfalls $\mathrm{NP}$-hart:
Gibt es $x \in \mathbb{N}^n$ oder sogar $x \in \{0,1\}^n$, so dass $c^T x \geq k$, vorausgesetzt $Ax \leq b$ für vorgelegte $A, b, C, k$?

Wir haben also VC auf ILP reduziert, i.Z.: VC $\leq_p$ ILP,
indem wir gezeigt haben, wie wir VC mit ILP modellieren können.

Es ist aber oft eine gute Heuristik, die Integrabilitätsbedingung zu relaxieren, also die Ganzzahligkeitsbedingung fallenzulassen, und dann (i.allg.) nicht-ganzzahlige Lösungen als Schranken zu gewinnen.
Dies hilft auch bei Branch&Bound-Ansätzen.

Leichte Übung: Warum liegt ILP in $\mathrm{NP}$?