Algorithmen für Chip-Entwurfswerkzeuge

Exakte Optimierungsverfahren

TECHNISCHE UNIVERSITÄT DARMSTADT

Vorlesung
WS 2013/2014

Florian Stock, Andreas Koch

Eingebette Systeme und Anwendungen
Technische Universität Darmstadt

Embedded Systems & Applications

Platzierungsproblem allgemein
Design Stile

TECHNISCHE UNIVERSITÄT DARMSTADT

- Design Stile:
  - Standardzellen
  - Gate Arrays
  - Makroblock
  - Mixed-Size
  - "*UPP*" - Einfaches Beispielproblem
- Alle Gleich:
  Platziere die zur Verfügung stehenden Module Überlappungsfrei!
- Unterschiedlich in Zielfunktionen und Randbedingungen

Design Stil
Standarzellen-Design

TECHNISCHE UNIVERSITÄT DARMSTADT

- Semi-Custom
- Bibliothek mit Logikmodule mit einfachen Funktionen (AND, OR, Inverter, . . .)
  - Alle Module haben die gleiche Höhe
  - Module habe variable Breite
- Es gibt für das Platzieren vordefinierte Reihen
- Sehr beliebter Design-Ansatz

$\Rightarrow$ Viele Algorithmen gehen von Standardzellen Design aus

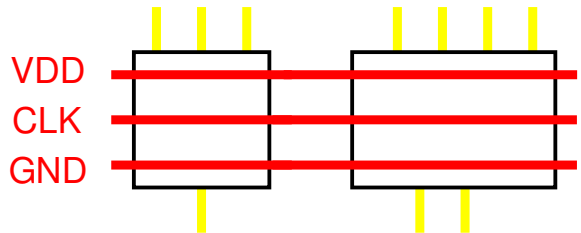
- Platzierung überlappungsfrei innerhalb der Reihen

Design Stil
Standarzellen-Design

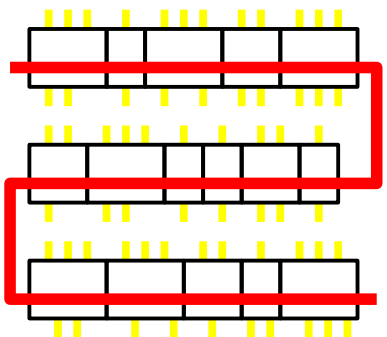
TECHNISCHE UNIVERSITÄT DARMSTADT

Routing:

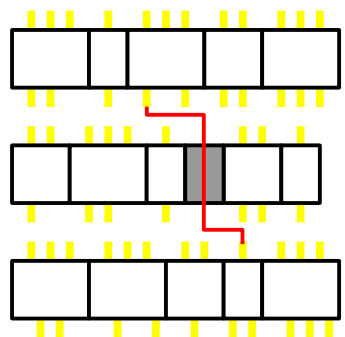
- Infrastruktur (V_{DD}, *CLK*, *GND*) durch alle Reihen



- Verdrahtung zwischen Reihen
- Ausnahmen:



Angrenzende Verbindungen (abutment)

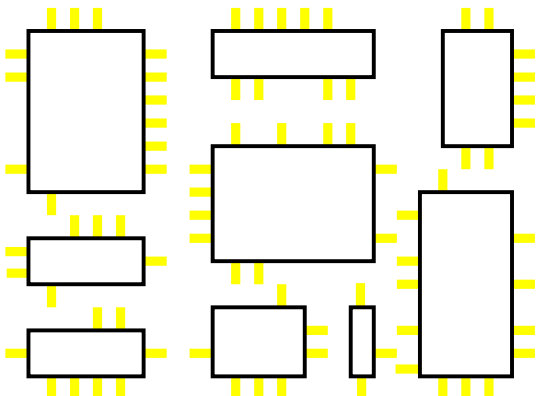


Durchleitungen (feedthroughs)

Design Stil
Makroblock-Design

TECHNISCHE UNIVERSITÄT DARMSTADT

- Alle Module sind Makroblöcke (Building-Blocks) fester Größe, Form und Ausrichtung
  - Kann auch Full-Custom Teile enthalten
  - Automatisch generierte Blöcke (z.B. RAM)
- Verdrahtungskanäle an allen Seiten
- Alle sind überlappungsfrei zu platzieren
- Ähnlich zu Floorplanning (dort sind i.d.R. Form und Orientierung variabel).



Design Stil
Mixed-Size-Design

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sehr häufig benutzt
- Vereint
  - Makroblöcke und
  - Standardzellen
- Makroblöcke $\gg$ Standardzellen
  $\Rightarrow$ Schwer Überlappungen zu vermeiden

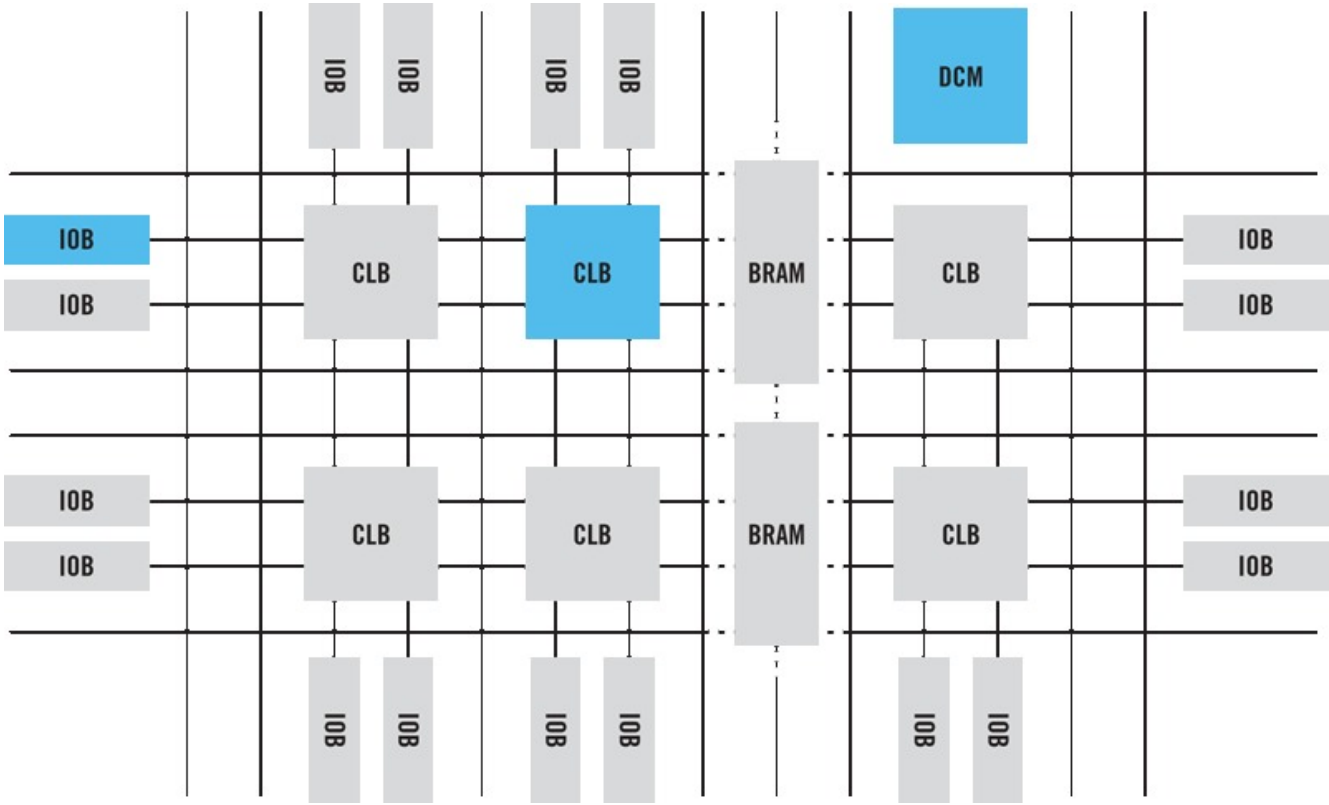
Zielarchitektur
MPGA/FPGA

TECHNISCHE UNIVERSITÄT DARMSTADT

- Gate Arrays
  - Field Programmable Gate Array (FPGA)
    Wird beim Anwender an Funktion angepasst ⇒ Programmierung
  - Mask Programmable Gate Array (MPGA)
    Andere Bezeichnung: Structured ASIC
    Wird beim Hersteller an Funktion angepasst ⇒ Metallagen Herstellung
- Reguläre Struktur mit fester Anordnung von
  - Programmierbarer Logik
  - Festen Funktionsblöcken
  - Speicher
  - Verdrahtung

Zielarchitektur
MPGA/FPGA

TECHNISCHE UNIVERSITÄT DARMSTADT



[Quelle: Xilinx]

Zielarchitektur
MPGA/FPGA

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sehr ähnlich zu UPP
- Aber: Segmentierte Verbindungen
  $\Rightarrow$ Mehrere Verdrathungslängen
- Verzögerung abhängig von
  - Anzahl durchlaufener Switch Boxes
  - Last (Fan-Out)
- Feste Verdrahtungskapazität
  $\Rightarrow$ Nicht jede Platzierung verdrahtbar
- Verdrahtbarkeit in Kostenfunktion

Exakte Lösungsverfahren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Erschöpfende Suche
  - Durchlaufen des gesamten Lösungsraums
  - Beispiel: Backtracking
- Eliminierung *schlechter* Ansätze
  - Abschätzung aus Teillösung
  - Beispiel: Branch-and-Bound
- Wiederverwendung alter Ergebnisse
  - Beispiel: Dynamic Programming
- Mathematisches Modell
  - Beispiel: Linear Programming

Lösungsdefintion

TECHNISCHE UNIVERSITÄT DARMSTADT

- Probleminstanz $I = (F, c)$
  - Lösungsraum F
  - Kostenfuntion $c : F \mapsto \mathbf{R}$
- Lösung $\vec{f} \in F : \vec{f} = [f_1, ... f_n]^T$
  - Explizite Einschränkungen: Wertebereiche der f_i
  - Implitizite Einschränkunge: Abhängigkeiten
- Teillösung $\underline{\vec{f}}$
  - Einige f_i undefiniert
  - Spannt Unterraum von F auf

Backtracking

TECHNISCHE UNIVERSITÄT DARMSTADT

- Den gesamten Lösungsraum enumerieren
- Jede Lösung wird dabei schrittweise aufgebaut (f_1, f_2, ...)
- Zu jeder Lösung die Kosten bestimmen und ggf. BSF aktualisieren
- Üblicherweise: Rekursiv

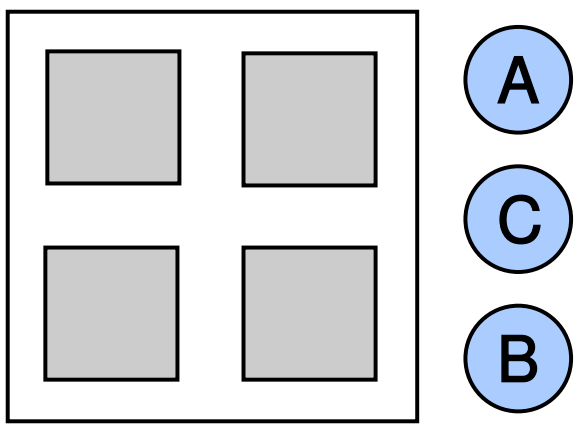
```
float bsfCost ;
solution_element current[n], bsf[n] ;
main(): begin
    bsfCost := ∞ ;
    backtrack(0) ;
    report(bsf) ;
```

```
backtrack(int k): begin
    float newCost ;
    solution_element sol_el ;
    if k = n then
        new_cost := cost(current) ;
        if newCost < bsfCost then
            bsfCost := newCost ;
            bsf := copy(bsf) ;
    else
        foreach (sol_el ∈
        allowed(current, k)) do
            current[k] := sol_el ;
            backtrack(k+1) ;
```

Backtracking
UPP Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lösung: Bei n Zellen auf m möglichen Positionen:
  $\vec{f} = [f_1, \ldots, f_n]^T$
  Zelle i wird auf Position f_i platziert
- Explizite Einschränkungen: $f_i \in 1, \ldots, m$
- Implizite Einschränkungen: $i \neq j \Rightarrow f_i \neq f_j$
- Konkretes Beispiel: $n = 3$, $m = 4$, HPWL-Kosten



Netzliste:

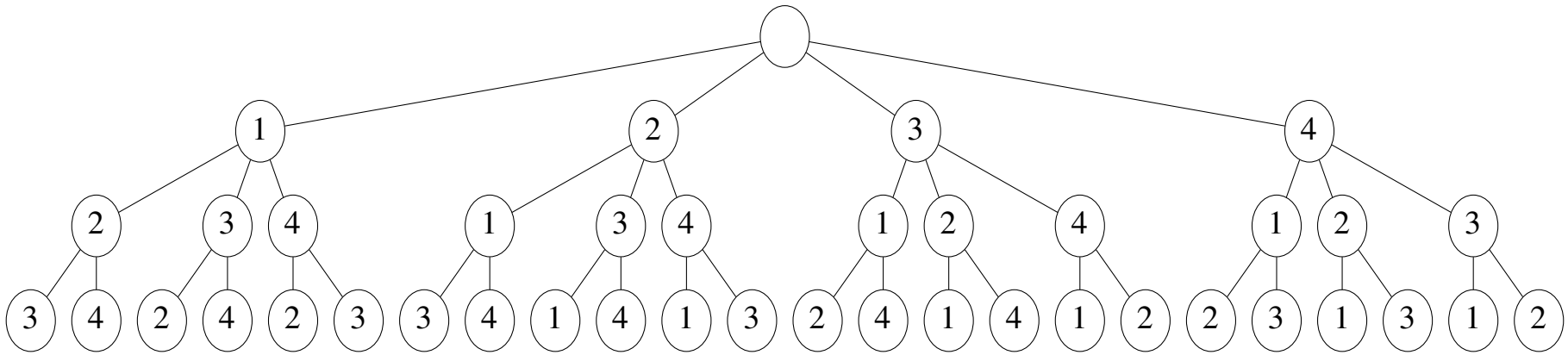
n_1 : A, B, C

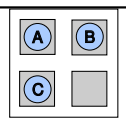
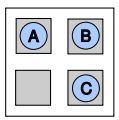
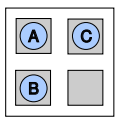
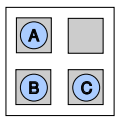
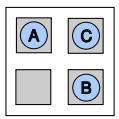
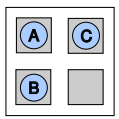
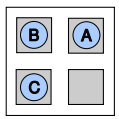
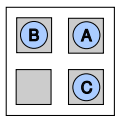
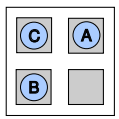
n_2 : A, B

n_3 : B, C

Backtracking
UPP Suchbaum

TECHNISCHE UNIVERSITÄT DARMSTADT



Lsg.	$\vec{f}$	Kosten
	123	5
	124	5
	132	5
	134	4
	142	5
	143	5
	213	4
	214	5
	124	5
⋮	⋮	⋮

Branch-and-Bound

TECHNISCHE UNIVERSITÄT DARMSTADT

- Branch = Verzweigen des Suchbaums
- Bound = Beschneiden der Äste wenn möglich
- Teillösung $\underline{\vec{f}}^{(k)} = [f_1, \ldots, f_k, \perp, \ldots, \perp]$
- $D(\underline{\vec{f}}^{(k)})$: Menge aus $\underline{\vec{f}}^{(k)}$ herleitbarer Lösungen
- Minimale mögliche Kosten $\underline{cost}$ einer Teillösung $\underline{\vec{f}}^{(k)}$ abschätzen
  Verwerfe $\underline{\vec{f}}^{(k)}$ falls $\underline{cost}(\underline{\vec{f}}^{(k)}) > cost(BSF)$
  $\Rightarrow$ Suchbaum wird gestutzt

Branch-and-Bound
Abschätzfunktion *cost*

TECHNISCHE UNIVERSITÄT DARMSTADT

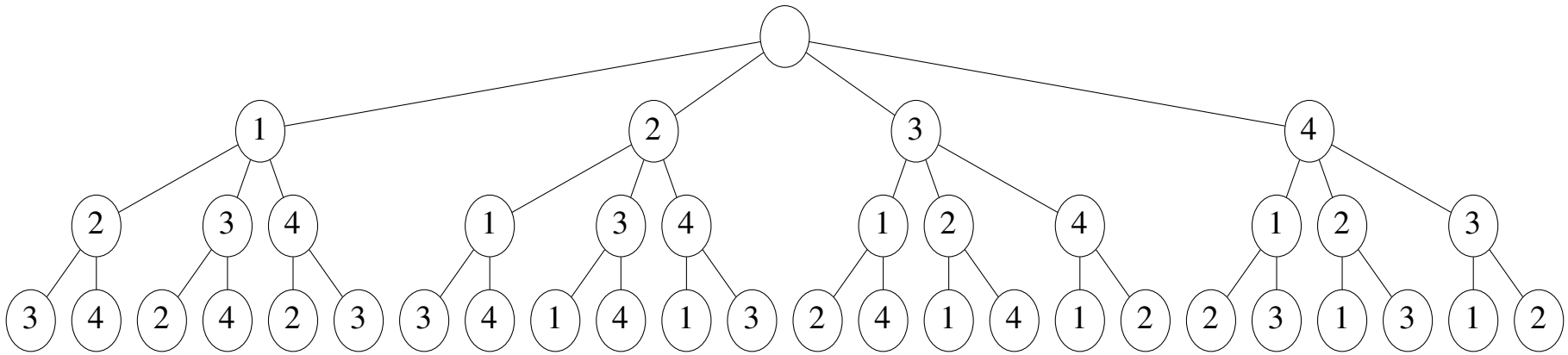
- Effekt der Abschätzung
  - Reale Kosten höher als geschätzte Kosten
    - ⇒ Zu optimistisch (“Ja, es lohnt sich weiterzumacen”)
    - ⇒ Überflüssige Schritte
  - Reale Kosten niedriger als geschätzte Kosten
    - ⇒ Zu pessimistisch (“Nein, das bringt nichts mehr”)
    - ⇒ Optimum wird möglicherweise weggestutzt
    - ⇒ Keine exakte Lösung mehr!
  - *cost* sollte möglichst genau sein

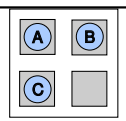
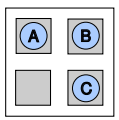
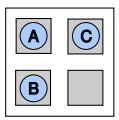
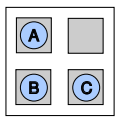
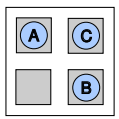
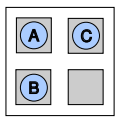
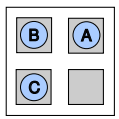
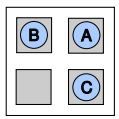
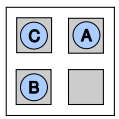
UPP Beispiel: Kostenabschätzung

- Gute Abschätzung: Minimal mögliche Kosten eines Netzes,

Branch-and-Bound
UPP Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT



Lsg.	$\vec{f}$	Kosten
	123	5
	124	5
	132	5
	134	4
	142	5
	143	5
	213	4
	214	5
	124	5
⋮	⋮	⋮

Dynamisches Programmieren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wiederverwenden von Lösungen
- Prinzip der Optimalität:
  - Annahme: Lösung eines komplex Problems kann optimal aus dem optimalen Lösungen von Teilproblemen zusammengesetzt werden (ebenfalls Idee bei Teile-Und-Herrsche)
  - Gilt aber nicht für alle Probleme!
- p Parameter für Problemkomplexität

  $p = k$: Gesamtproblem

  $p < k$: Teilproblem

  $p = 0$ oder $= 1$: Kleinstes Problem
- Für viele Probleme anwendbar. Beispielsweise:
  - Klassisches Beispiel: Fibonacci-Zahlen
  - Dijkstra SP
  - Aber auch für NP-vollständige Probleme

Dynamisches Programmieren
Beispiel Fibonnacci-Zahlen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Fibonacci-Zahlen: 0, 1, 1, 2, 3, 5, ...
- $F_n = F_{n-1} + F_{n-1}$ mit $F_0 = 0, F_1 = 1$

Klassische Implementierung:

```
fib(int n): begin
    if (n=0) then
        return 0
    if (n=1) then
        return 1
    else
        return (fib(n-1)+fib(n-2))
```

- $\frac{F_{n+1}}{F_n} \rightarrow 1.6$ (genau $\frac{1+\sqrt{5}}{2}$) $\Rightarrow F_n > 1.6^n$
- Da immer 1 aufsummiert werden $\Rightarrow \mathcal{O}(1.6^n)$

Dynamisches Programmieren

```
fib(int n): begin
    Fib[0] := 0 ;
    Fib[1] := 1 ;
    for (i=2; i ≤ n; i++) do
        Fib[i] := Fib[i-1] + Fib[ i-2 ]
    return Fib[n] ;
```

- Teillösungen bei $p = i$
- Gesamtlösung bei $p = n$

Mathematische Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nahezu alle Probleme Optimierungsprobleme
- Entsprechende mathematische Modelle wählen und bekannte mathematische Verfahren anwenden

Mathematische Optimierung
Lineare Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Verfahren aus dem Operations Research
- Beispiel:

	Produkt 1	Produkt 2	Liefermenge
Rohstoff A	42	14	100
Rohstoff B	23	53	200
Preis	550	250	

Ziel: Optimiere auf maximalen Umsatz

Mathematische Optimierung
Lineare Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Modell: x_1, x_2 Anzahl Produkte 1 und 2
- Kanonische Form (üblicherweise max und $x_i \geq 0$ implizit)

$$\begin{aligned} \texttt{max:} \quad & 550x_1 + 250x_2 \\ \texttt{subject to:} \quad & 42x_1 + 14x_2 \leq 100 \\ & 23x_1 + 53x_2 \leq 200 \end{aligned}$$

$$\texttt{and} \quad x_i \geq 0$$

- Lineares Programm
- Matrixformulierung:

$$\begin{aligned} \texttt{max:} \quad & \vec{c} \cdot \vec{x} \\ \texttt{s.t.:} \quad & A\vec{x} \leq \vec{b}, x_i \geq 0 \end{aligned}$$

Mathematische Optimierung
Lineare Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lösung mittels Simplex-Verfahren (Danzig, 1947)
  - Darstellung der Kostenfunktion und Restriktionen als Hyperebene im Raum
  - Restriktionen bilden ein Simplex (Polyeder), Lösung eine der Ecken
  - Worst-Case-Komplexität: Exponentiell
  - Praktisch immer Polynomial
- Problem ist in **P**
  - Ellipsoid-Verfahren (1979)
  - Innere Punkte Methoden (1984)
    wandern nicht die Ecken ab, sondern gehen durch den Simplex
- Lösung durch *LP Solver*
  - lp_solve, GLPK, CLP, CPLEX, ...

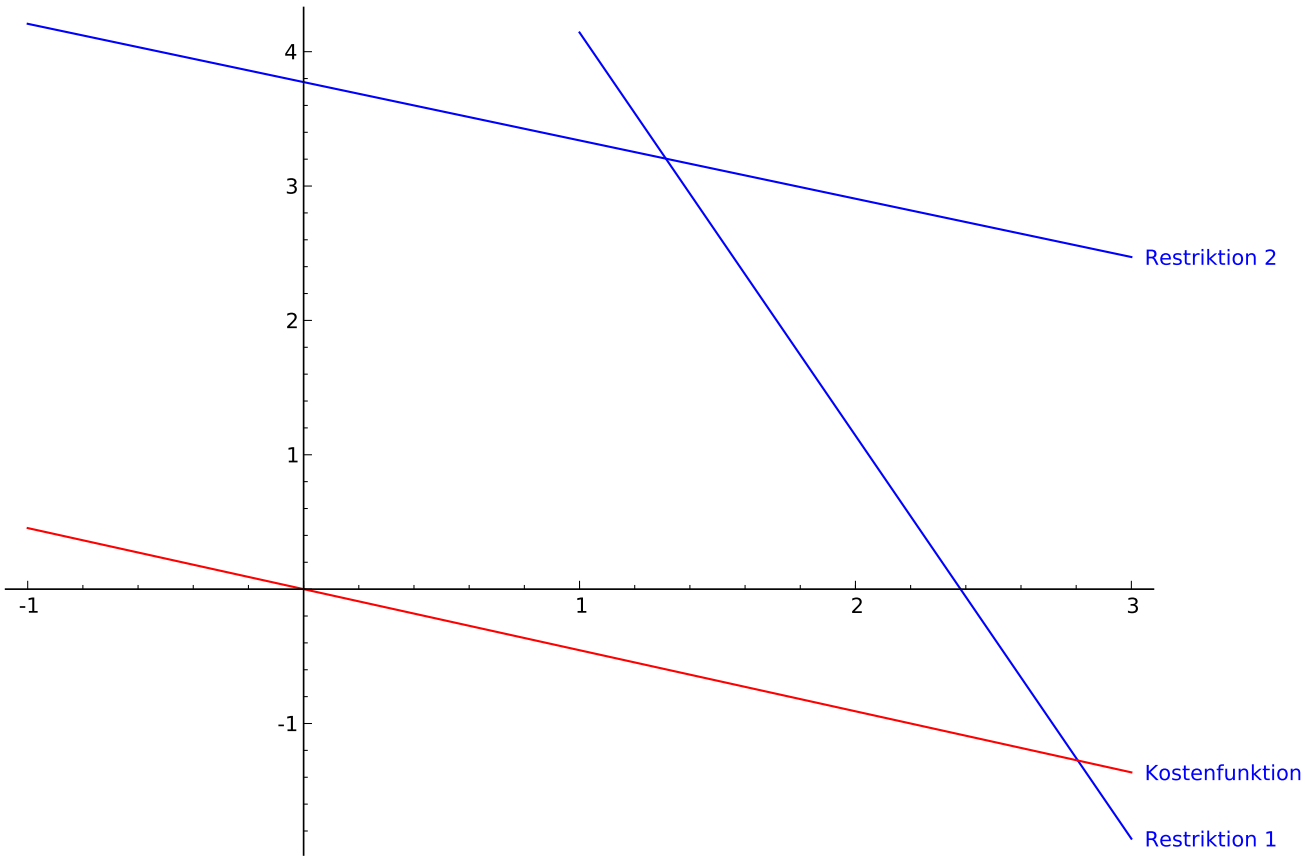
Beispiel

Optimum: $x_1 = 1.31303, x_2 = 3.20378$

Umsatz: 1523.11

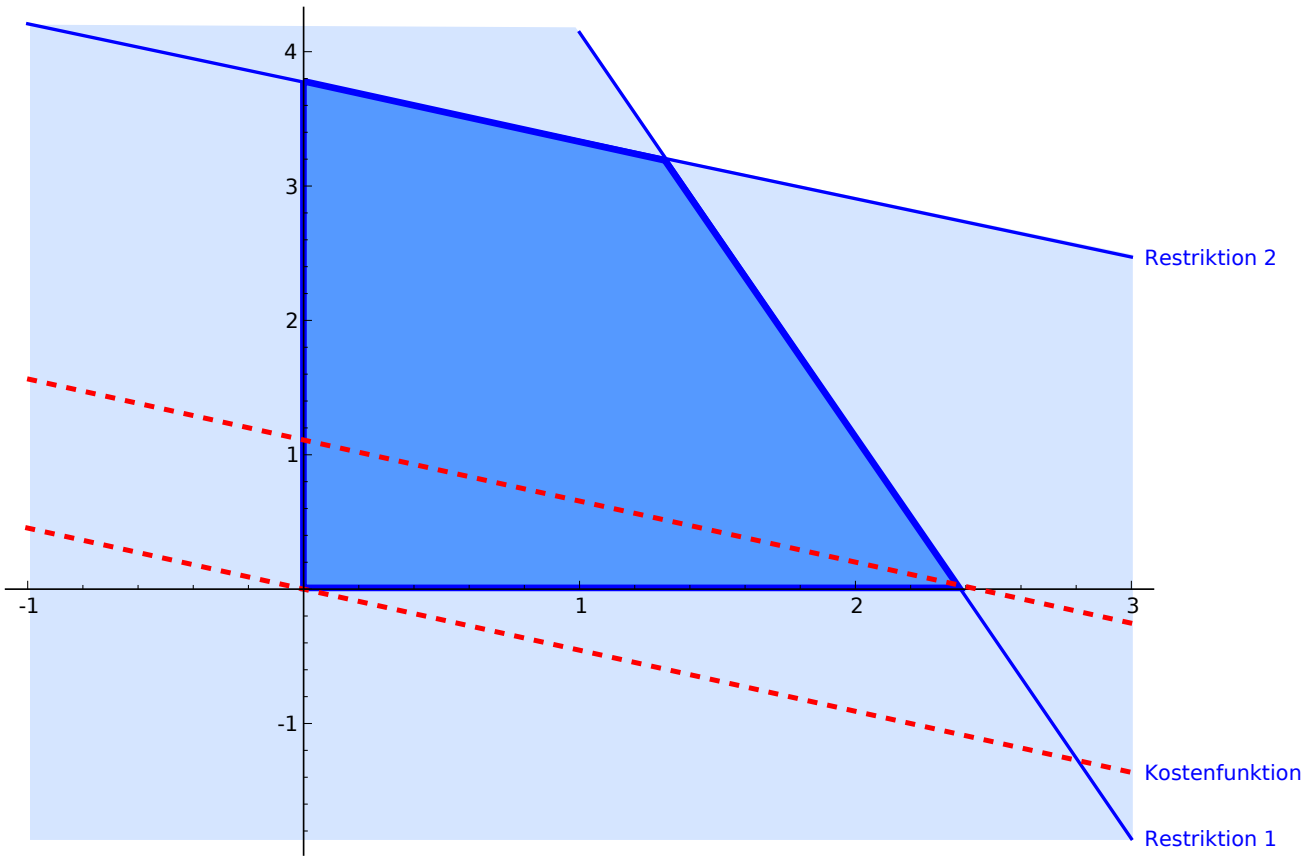
Mathematische Optimierung
Lineare Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT



Mathematische Optimierung
Lineare Programmierung

TECHNISCHE UNIVERSITÄT DARMSTADT



Mathematische Optimierung
Integer Lineare Optimierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Problem: Oft nur ganzzahlige Variablen erlaubt!
  Es lassen sich nicht 1.31303 Produkte herstellen/verkaufen

$\Rightarrow$ Integer Lineare Programmierung (ILP)

- Lösungsmethoden komplizierter
  Lösungsverfahren NP-vollständig
- Einfachste Heuristik: Rundung
- Nicht sinnvoll
  - Sub-Optimial
  - Unzulässige Lösung

Beispiel

$x_1 = 2, x_2 = 1$
Umsatz: 1350

Mathematische Optimierung
ILP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Lösungverfahren:
  - LP kombiniert mit branch-and-bound (auf- und abrunden)
  - SAT (Erfüllbarkeitsproblem) (nur bei $0-1$-ILP)
  - Schnittebenenverfahren (Cutting-Planes)
    Wiederholt normales LP lösen und Restriktionen für Ganzzahligkeit hinzufügen
  - Manchmal durch Matrixstruktur implizit garantiert, dass Lösung ganzzahling ist:
    Total unimodulare Matrizen

Beispiele

- Inzidenzmatrix eines bipartiten Graphen
- Restriktionsmatrix von Max-Flow-Problemen

$\Rightarrow$ Standard Löser benutzen

Mathematische Optimierung
LP Varianten

TECHNISCHE UNIVERSITÄT DARMSTADT

Häufige Varianten:

- Variablen nur 0 oder 1:
  0 – 1-ILP
- Gemischt LP und ILP:
  Mixed Linear Programm (MLP)
- Quadratische Zielfunktion (weiterhin Restriktionen linear):
  Quadratisches Programmieren (QP)
  Vorteil bei Quadratisch-Konvexen-Funktionen: Lokale Extrema = Globale

ILP
UPP Beispiel

TECHNISCHE UNIVERSITÄT DARMSTADT

- Modell:

  $x_{ij} \in \mathbf{B}$: Zelle $i = 1, \ldots, N$ wird auf Position $j = 1, \ldots, M$ platziert.

  $n_k \in \mathbf{R}$: Kosten für ein Netz $k = 1, \ldots, K$.

  Zielfunktion: min: $\sum_k n_k$

  Restriktionen:
  - Alle Platzieren: $\forall i \sum_j x_{ij} = 1$ ($2N$ Restriktionen)
  - Überlappungsfreiheit: $\forall j : \sum_i x_{ij} \leq 1$ ($2M$ Restriktionen)
  - Kostenfunktion: $n_k = c_{kV} + c_{kH}$
  - Netz k:

    $c_{kV} = c_{kMaxV} - c_{kMinV}$

    $c_{kMaxV} = max_{i \in \text{ Netz } k} \{ x_{ij} \cdot$ 'zu j gehöriger x-Wert' $\}$

    $c_{kMinV} = min_{i \in \text{ Netz } k} \{ x_{ij} \cdot$ 'zu j gehöriger x-Wert' $\}$

    c_{kH} analog

    ($2K(2 + 2 \cdot \sum_k |\text{Netz } k|)$ Restriktionen)

Mathematische Optimierung
Konjugiertes Gradientenverfahren (CG)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Eigentlich benutzt um große LGS zu lösen
- Vorteilhaft bei dünnbesetzten (sparse) Matrizen
- Minimieren einer Funktion $f(\vec{x}) = \frac{1}{2} \cdot \vec{x}^T A\vec{x} - b \cdot \vec{x} + c$,
  A symmetrisch, positiv definit
- Gradient: Vektor der partiellen Ableitungen $\nabla f(\vec{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial x_3}, ... \right)$
  in unserem Fall $f'(\vec{x}) = A\vec{x} - b$.
- $f'(\vec{x}) = 0 \Leftrightarrow$ lösen des LGS $Ax = b$

Mathematische Optimierung
Konjugiertes Gradientenverfahren

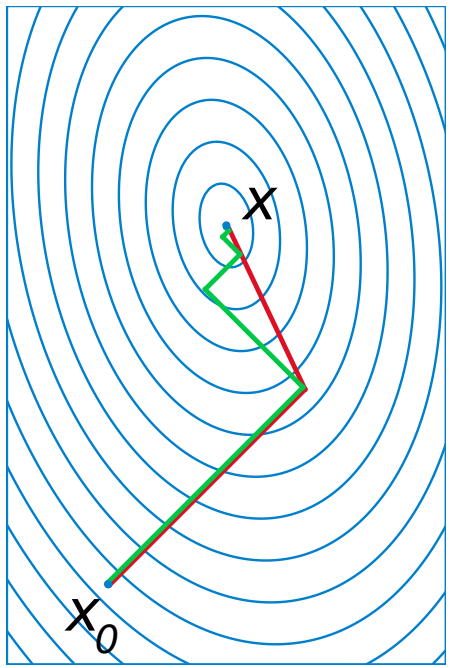
TECHNISCHE UNIVERSITÄT DARMSTADT

- Krylow-Unterraum-Verfahren
- Steilster Anstieg in Punkt x_k ist Gradient $f'(\vec{x_k}) = A\vec{x_k} - b$
- Residuum $r_k := -(A\vec{x_k} - b)$
- Minimiere Funktion in Richtung d_k
  d_i sind bzgl. A orthogonal zueinander (konjugiert)

$\Rightarrow$ α_k in entsprechende Richtung gehen

$\Rightarrow$ Neues x_k

- Neues Residuum r_{k+1} aus Altem und der Richtung d_k (mit zugehöriger Schrittweite β_k bestimmen)



[Quelle: Wikimedia]

Mathematische Optimierung
Konjugiertes Gradientenverfahren

TECHNISCHE UNIVERSITÄT DARMSTADT

cg(A,b) **begin**

Wähle $x_o \in \mathbf{R^n}$;

$r_0 := b - Ax_0$; $d_0 := r_0$;

k := 0 ;

repeat

$\alpha_k := \frac{r_k^T r_k}{d_k^T A d_k}$;

$x_{k+1} := x_k + \alpha_k \cdot d_k$;

$r_{k+1} := r_k + \alpha_k \cdot A d_k$;

$\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$;

$d_{k+1} := r_{k+1} + \beta_k \cdot d_k$;

k++ ;

until *Residuum* $||r_k|| < \epsilon$;

return x_k ;

- x_0 beliebig
  schnellere Konvergenz, falls x_0 nahe an Lösung
- Optimierung: In der Schleife $A \cdot d_k$ nur einmal berechnen
- Auch andere Abbruchbedingungen möglich:
  - $\Delta x_k < \epsilon$
  - Max. Anzahl Iterationen
  - ...

Mathematische Optimierung
Konjugiertes Gradientenverfahren (nichtlinear)

TECHNISCHE UNIVERSITÄT DARMSTADT

cg(A,b) **begin**

Wähle $x_o \in \mathbf{R^n}$;

$r_0 := -\nabla f(x_0)$; $d_0 := r_0$;

k := 0 ;

repeat

$\alpha_k := \frac{r_k^T r_k}{d_k^T \nabla^2(f(x_k)) d_k}$;

$x_{k+1} := x_k + \alpha \cdot d_k$;

$r_{k+1} := -\nabla f(x_{k+1})$;

$\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$;

$d_{k+1} := r_{k+1} + \beta_{k+1} \cdot d_k$;

k++ ;

until *Residuum* $||r_k|| < \epsilon$;

return x_k ;

- Hesse-Matrix zur α-Bestimmung
  Üblicherweise $\mathcal{O}(n^2)$,
  problemabhängig ggf. nur $\mathcal{O}(n)$
- Andere Möglichkeit:
  α mittels Linesearch
- Andere Abbruchbedingungen

Mathematische Optimierung
Konjugiertes Gradientenverfahren

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kovergenz abhängig von Matrix (Konditionszahl)
- Konvergenz:
  - Theoretisch: Nach n Schritten
  - Praktisch I: Numerische Ungenauigkeiten machen mehr notwendig
  - Praktisch II: Schneller, abhänging von Anzahl verschiedener Eigenwerte
- Wenn nötig Matrix konditionieren
  $\Rightarrow$ Preconditioner
- Verschiedene Varianten, andere Bestimmung von β
  $\Rightarrow$ ggf. besseres Konvergenzverhalten
- Optimierung von Funktionen mit Nebenbedingungen:
  - Lagrange-Multiplikatoren
  - Nebenbedingungen werden mittels Kostenfunktion berücksichtigt

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Allgemeines Platzierungsproblem
- Design-Stile & Zielarchitekturen
  - Standardzellen
  - Makro-/Buildingblock
  - FPGA/MPGA
  - Mixed-Mode
- Exakte Optimierungsverfahren
  - Backtracking
  - Branch-and-Bound
  - Dynamisches Programmieren
  - Mathematische Modelle
    - LP/ILP/MLP
    - CG-Verfahren