



Algorithmen und Datenstrukturen

Kapitel 04: Algorithmenmuster

Prof. Dr. Adrian Ulges

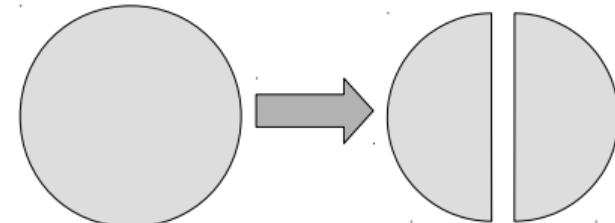
B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Algorithmenmuster

Verschiedene Informatik-Probleme werden oft mit **ähnlichen Verfahren** gelöst.
→ Wiederverwendung von **Lösungsmustern**.

Beispiel

- ▶ Binäre Suche
- ▶ Mergesort
- ▶ Quicksort



Diese Verfahren basierten alle auf der Idee, das gegebene Problem in (zwei) Teile zu zerlegen. → Entwurfsmuster **Divide-and-Conquer**.

In diesem Kapitel: Vier gängige Algorithmen-Muster

- ▶ Divide-and-Conquer (*“Teile und herrsche”*)
- ▶ Greedy-Verfahren (*“Gierige” Verfahren*)
- ▶ Backtracking (*“Versuch und Irrtum”*)
- ▶ Dynamische Programmierung



Outline

1. Divide-and-Conquer
2. Greedy-Verfahren
3. Backtracking
4. Dynamische Programmierung

Divide-and-Conquer

Genereller Ansatz: Zerlege das Problem in einzelne Teile, löse die Teile, setze Teillösungen zu Gesamtlösung zusammen.

```
# Generelles Muster
function loese(problem):

    if ist_trivial(problem):
        return triviale_loesung(problem)

    problem1,...,problemn := zerlege(problem)

    for i = 1,...,n:
        loesungi := loese(problemi)

    loesung = kombiniere(loesung1,
                          ...,
                          loesungn)

    return loesung
```

```
# Beispiel: Mergesort
Function mergesort(a[left],...,a[right]):

    if left >= right: return

    m = (left + right) / 2
    a1 = a[left],...,a[m]
    a2 = a[m+1],...,a[right]

    mergesort(a1)
    mergesort(a2)

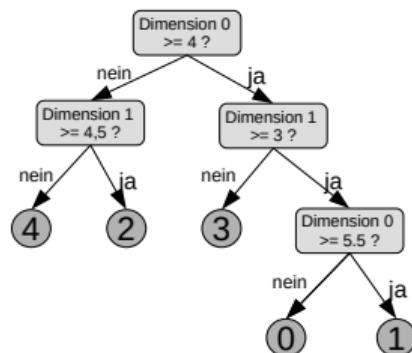
    Mische a1 und a2
```

Divide-and-Conquer

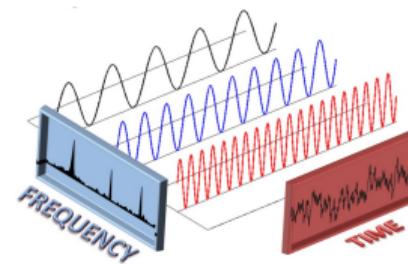
Wann ist ein Problem mit Divide & Conquer lösbar?

1. Das Ausgangsproblem ist in Teilprobleme **zerlegbar**.
2. Die Lösung ist aus Teillösungen **konstruierbar**.

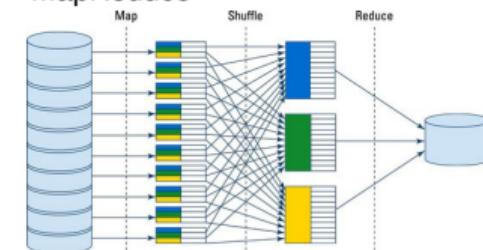
Aufbau von k-d-Bäumen



Die Fast Fourier Transform (FFT)



MapReduce

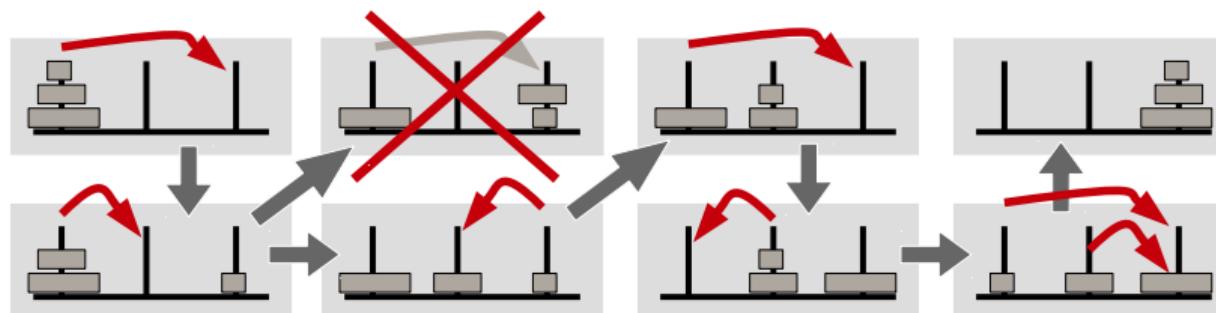
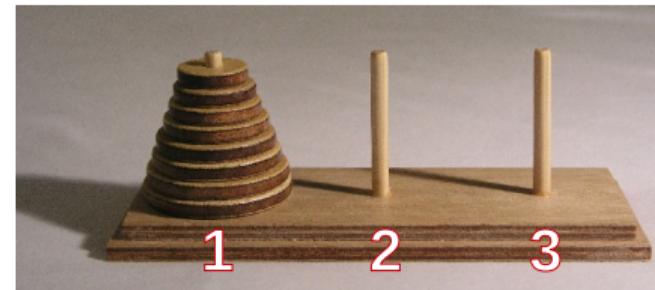


Divide-and-Conquer: Beispiel

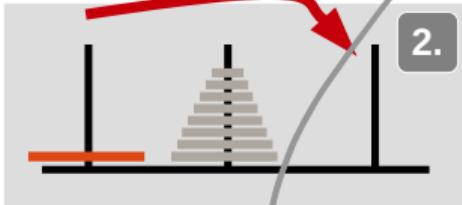
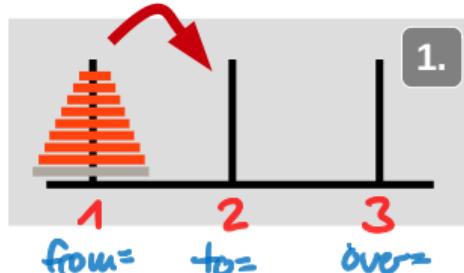


Die Türme von Hanoi

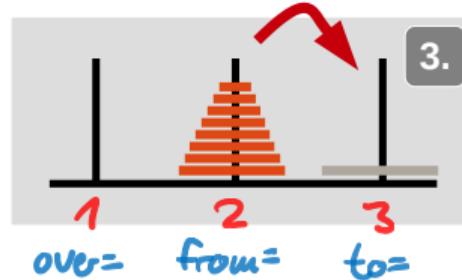
- ▶ **Gegeben:** Ein Turm aus **n Scheiben** und **3 Plätze** zum Stapeln von Scheiben.
- ▶ **Aufgabe:** Bewege den Turm von Platz 1 zu 3.
- ▶ **Erlaubte Züge:** Nehme die **oberste Scheibe** von einem Platz und lege sie **oben** auf einen anderen Platz.
- ▶ **Bedingung:** Eine größere Scheibe darf **niemals** auf einer kleineren liegen.



Türme von Hanoi: Pseudo-Code



move(100, from=1, to=3, over=2) *



Lösung mit Divide-and-Conquer

Idee: Verkleinere das Problem: Turm der Höhe n → Turm der Höhe n-1
(alle Scheiben bis auf die unterste!)

1. Bewege Teil-Turm der Höhe n-1 auf freien Platz (2)
2. Bewege die unterste Scheibe auf das Ziel (Platz 3)
3. Bewege Teil-Turm auf Ziel (Platz 3).

```
procedure move(n, from=1, to=3, over=2):
    # Bewege eine Scheibe
    if n==1:
        print ('move disk from' + from +
               'to' + to)
    else:
        (1.) move(n-1, from=from,
                  to=over,
                  over,to)
        (2.) move( 1, from=from, to=to, over=over)
        (3.) move(n-1, from=over,
                  to=to,
                  over=from)
```

Türme von Hanoi: Analyse

Laufzeit $f(n) := \#\text{Schritte} (\#\text{Scheiben-Bewegungen})$ für Turm der Höhe n .

Es gilt:

- $f(1) = 1$
- $f(n) = f(n-1) + 1 + f(n-1)$

$$= 2 \cdot f(n-1) + 1$$

$$= 2 \cdot \overbrace{(2 \cdot f(n-2) + 1)} + 1$$

$$= 2^2 \cdot f(n-2) + 2 + 1$$

$$= 2^2 \cdot \overbrace{(2 \cdot f(n-3) + 1)} + 2 + 1$$

$$= 2^3 \cdot f(n-3) + \underbrace{4}_{2^2} + \underbrace{2}_{2^1} + \underbrace{1}_{2^0}$$

= ...

$$= 2^{n-1} \cdot f(1) + \underbrace{\left(2^{n-2} + 2^{n-3} + \dots + 2^0 \right)}_{= 2^{n-1} - 1} = 2 \cdot 2^{n-1} - 1 = 2^n - 1$$

Exponentieller Aufwand
:(

Türme von Hanoi: Analyse



Türme von Hanoi: Analyse

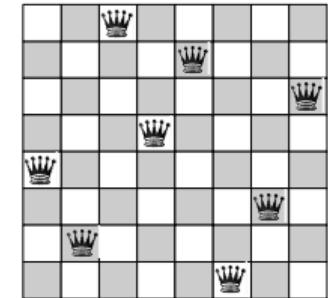
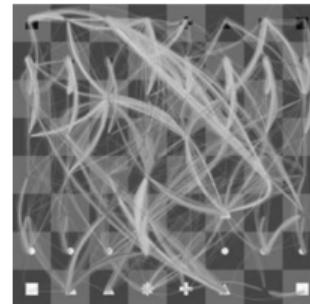
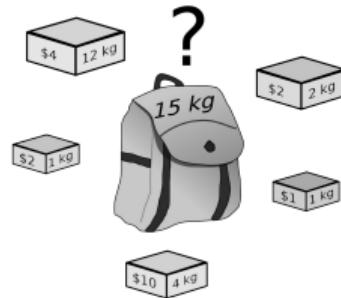
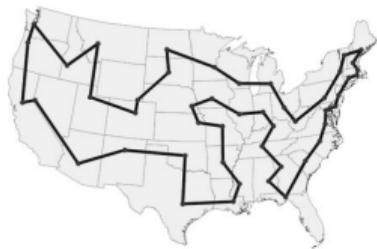




Outline

1. Divide-and-Conquer
2. Greedy-Verfahren
3. Backtracking
4. Dynamische Programmierung

Greedy-Verfahren: Motivation

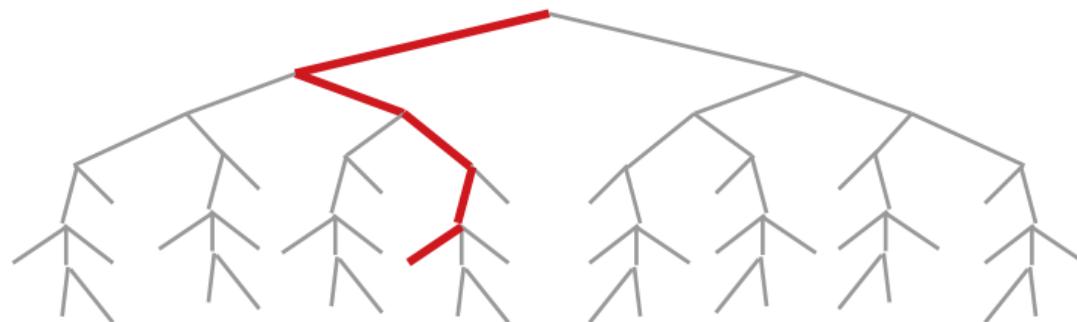


Oft **durchsuchen** Algorithmen einen **Lösungsraum** nach ("guten") Lösungen:

- ▶ **Traveling Salesman Problem (TSP)**: Wähle eine günstige Route/Permutation.
- ▶ **Knapsack**: Packe möglichst viele Objekte in begrenzten Raum.
- ▶ **Schach**: Wähle eine Sequenz von Zügen die zum Sieg führt.
- ▶ **N-Damen-Problem**: Ordne Damen so an, dass sie einander nicht schlagen.

Greedy-Verfahren: Motivation

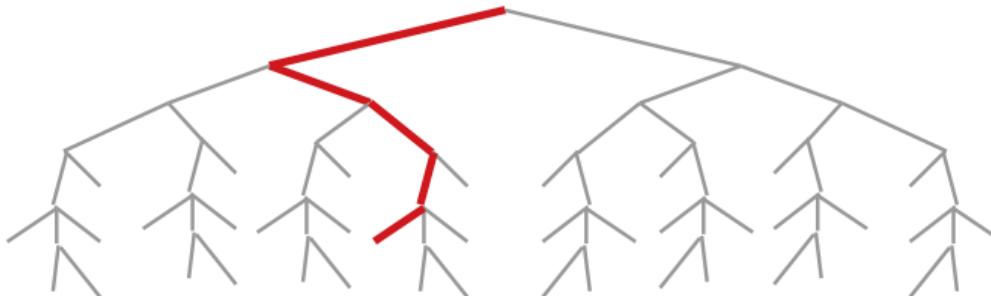
- ▶ Der Algorithmus beginnt in einem **Startzustand**.
- ▶ Unterschiedliche Änderungen führen zu unterschiedlichen neuen Zuständen.
- ▶ Es ergibt sich ein **Zustandsbaum**.



Struktur des Lösungsraums

- ▶ Manche Zustände sind nicht **valide** → **breche ab**.
- ▶ Manche Lösungen sind besser/kostengünstiger als andere → **optimale Lösung**.

Greedy-Verfahren



```
# Generelles Muster: Greedy
function greedy():

    zustand = startzustand()

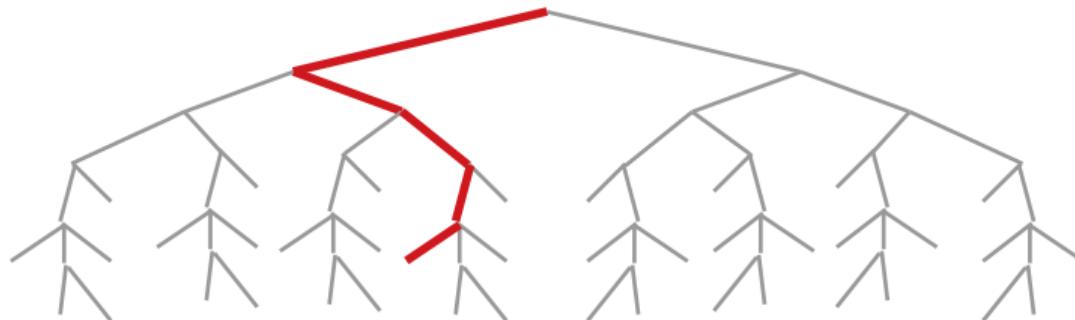
    while !ist_loesung(zustand):

        Z = folgezustände(zustand)
        zustand = argmaxz $\in$ Z gute(z)

    return zustand
```

- ▶ Greedy-Verfahren (engl. für *gierig*) gelangen in **möglichst wenig Schritten** zu einer Lösung.
- ▶ Beginne in Startzustand, berechne iterativ verbesserten Zustand.
- ▶ Ende wenn Problem gelöst.
- ▶ In jeder Iteration: **Lokal bestmögliche Änderung.**

Greedy-Verfahren



Was sind Voraussetzungen für ein Greedy-Verfahren?

- ▶ Zustände lassen sich einfach ineinander **überführen**.
- ▶ Zustände lassen sich **bewerten**.
- ▶ Ein “**optimaler**” **Zustand** ist gesucht.
- ▶ Eine **kontinuierliche Verbesserung** ist möglich.

Beispiel: Wechselgeld



- ▶ Ein Geldbetrag $x \in \mathbb{N}$ sei zu wechseln.
- ▶ **Gegeben:** Münzen mit Werten $W = \{1, 2, 5, 10, 20, 50\}$.
- ▶ Von jeder Münzsorte seien **ausreichend viele** Exemplare vorhanden.
- ▶ **Ziel:** Bilde eine **möglichst kleine** Menge
an Münzen mit Gesamtwert x .

Wechselgeld: Formalisierung

Ein **Zustand** $s = (n(1), n(2), n(5), n(10), n(20), n(50))$ gibt an, wieviele Münzen von welcher Sorte verwendet werden.

Beispiel: Wechsle $x=79$

- **Startzustand:** $(0, 0, 0, 0, 0, 0)$.
- Zustand s ist eine **Gesamtlösung** falls

$$x = \sum_{w \in W} n(w) \cdot w$$

- **Lösungsgüte** = Anzahl der Münzen
(je höher desto schlechter):

$$\sum_{w \in W} n(w)$$

- Optimale Lösung: $(0, 2, 1, 0, 1, 1)$.

Startzustand	0	0	0	0	0	0
valider Zustand	0	2	0	4	1	0
nicht valider Zustand	0	2	0	8	0	0
Lösung (aber nicht optimal)	2	1	3	0	3	0
Lösung (optimal)	0	2	1	0	1	1

Wechselgeld: Pseudo-Code

- Solange ein **Restbetrag** existiert ($x > 0$),
nehme die größte Münze, die in x hinein passt. Ziehe deren Wert von x ab.
- **Beispiel:** $79 = 50 + 20 + 5 + 2 + 2$ (*optimale Lösung*).

```
# Wechselgeld (Greedy)
function wechselgeld(x, W):
    # Zustand s als Array
    # (enthält Anzahl je Münze)
    s = [0,0,...,0]

    while x > 0:
        # Füge 'bestmögliche' Münze hinzu
        bestw      = max { w∈W | w≤x }
        s[bestw] += 1
        x         -= bestw

    return s
```

Greedy-Charakteristika

- **Zustandsübergang:** Nehme Münze hinzu.
- **Verbesserung** des Zustands: Restgeld x wird kleiner.
- **Greedy:** In jedem Schritt **möglichst große “Verbesserung”** des Zustandes.



Greedy: Optimalität

Findet das Greedy-Verfahren immer die optimale Lösung?

Das hängt von der **Struktur des Problems** ab.

Beispiel: Anderer Münzsatz

- ▶ Wir führen eine 11-Cent-Münze ein: $T = (1, 5, 10, 11)$.
- ▶ Für $x = 22\dots$ findet Greedy die optimale Lösung $(0, 0, 0, 2)$.
- ▶ Für $x = 15\dots$
 - ▶ ... lautet die Greedy-Lösung $(4, 0, 0, 1)$.
 - ▶ ... lautet die optimale Lösung $(0, 1, 1, 0)$.

$$15 = \text{11} \text{ } \text{5} \text{ } \text{10} \text{ } \text{10}$$

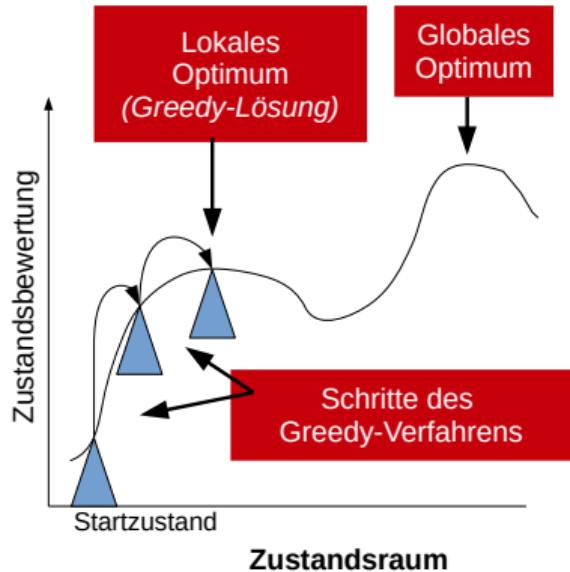
Greedy-Lösung

$$15 = \text{10} \text{ } \text{5}$$

Optimale Lösung

Greedy: Fazit

- ▶ Greedy-Verfahren findet im allgemeinen **nicht** die optimale Lösung.
- ▶ Sie wählen den **lokal** besten Schritt (*z.B. die größtmögliche Münze*).
- ▶ Aber: Greedy-Verfahren sind meist sehr **effizient**.



Greedy-Beispiel: TSP

Das “Traveling Salesman Problem” (TSP)

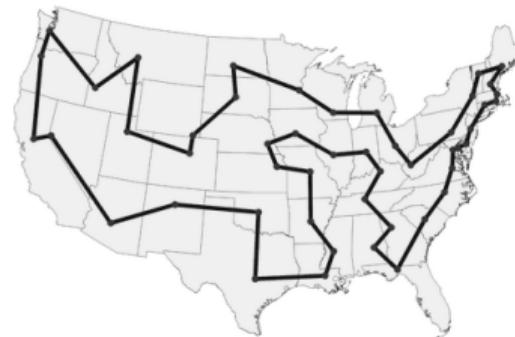
- ▶ In welcher Reihenfolge sollen die Städte besucht werden?
- ▶ **Greedy:** Erweitere die Route in jedem Schritt um die nächstgelegene Stadt mit minimaler Distanz.
- ▶ **Ist das Verfahren optimal?**

```
# Routenplaner
# Gegeben:
# - S (die Menge zu besuchender Städte)
# -  $s_0$  (die Anfangsstadt der Route)
# - cost:  $S \times S \rightarrow R$  (Kostenfunktion)

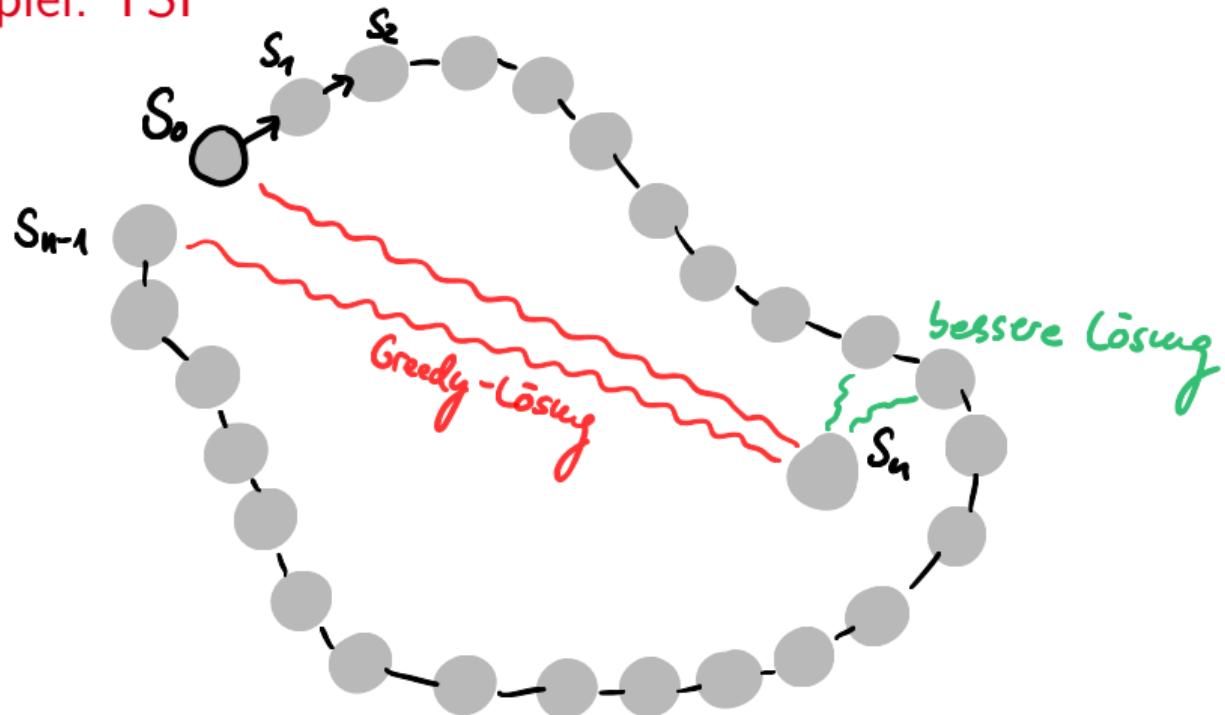
stadt :=  $s_0$ 
route := [] # leere Liste

while S != {}:
    stadt := argmin $_{s \in S}$  cost(stadt,s)
    route := route + [stadt]
    S      := S\{stadt}

return route
```



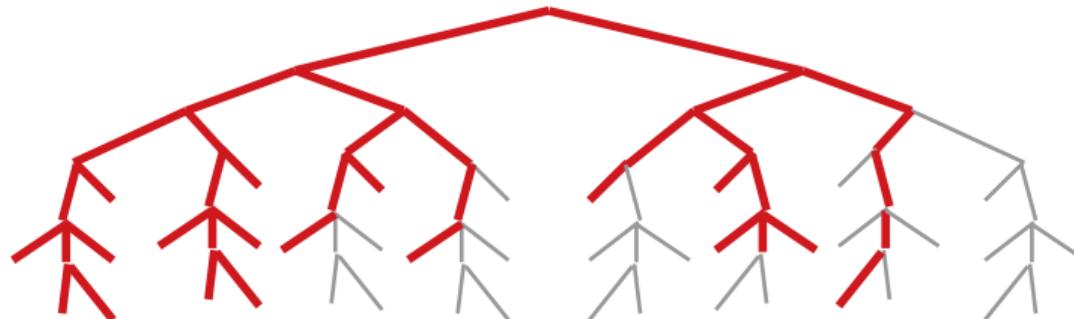
Greedy-Beispiel: TSP





Outline

1. Divide-and-Conquer
2. Greedy-Verfahren
3. Backtracking
4. Dynamische Programmierung



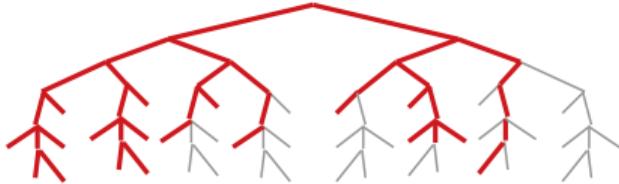
Greedy-Verfahren sind effizient, führen aber **nicht** immer zur **besten** Lösung.

Backtracking

... nimmt eine globale Sicht ein:

- ▶ Der **komplette Lösungsraum** wird durchsucht.
- ▶ Es wird **immer die optimale Lösung** gefunden!
- ▶ Einzelne Tiefen können aber **übersprungen** werden,
wenn sie zu nicht-validen oder suboptimalen Lösungen
führen.

Backtracking: Ansatz



- Wenn Lösung erreicht: Beende Suche.
- Wenn Zustand nicht valide oder nicht mehr vielversprechend, breche Suche ab.

Ansonsten:

- Probiere **jeden** (!) möglichen Folgezustand z .
- Suche von z aus nach einer Lösung.
- Wenn von z aus eine **bessere Lösung** erreicht wird, nehme diese.

Wir erhalten die **optimale Lösung** aller Möglichkeiten.

```
# Generelles Muster: Backtracking
function backtrack(zustand,
                  beste_loesung):

    if ist_loesung(zustand):
        return zustand

    if !ist_valide(zustand) or
       !verbesserung_moeglich(zustand,
                               beste_loesung):
        return None

    Z = folgezustaende(zustand)
    fuer jedes z in Z:
        loesung = backtrack(z, beste_loesung)

        if (gute(loesung) >
            gute(beste_loesung)):
            beste_loesung = loesung

    return beste_loesung

backtrack(startzustand(), None)
```

Backtracking

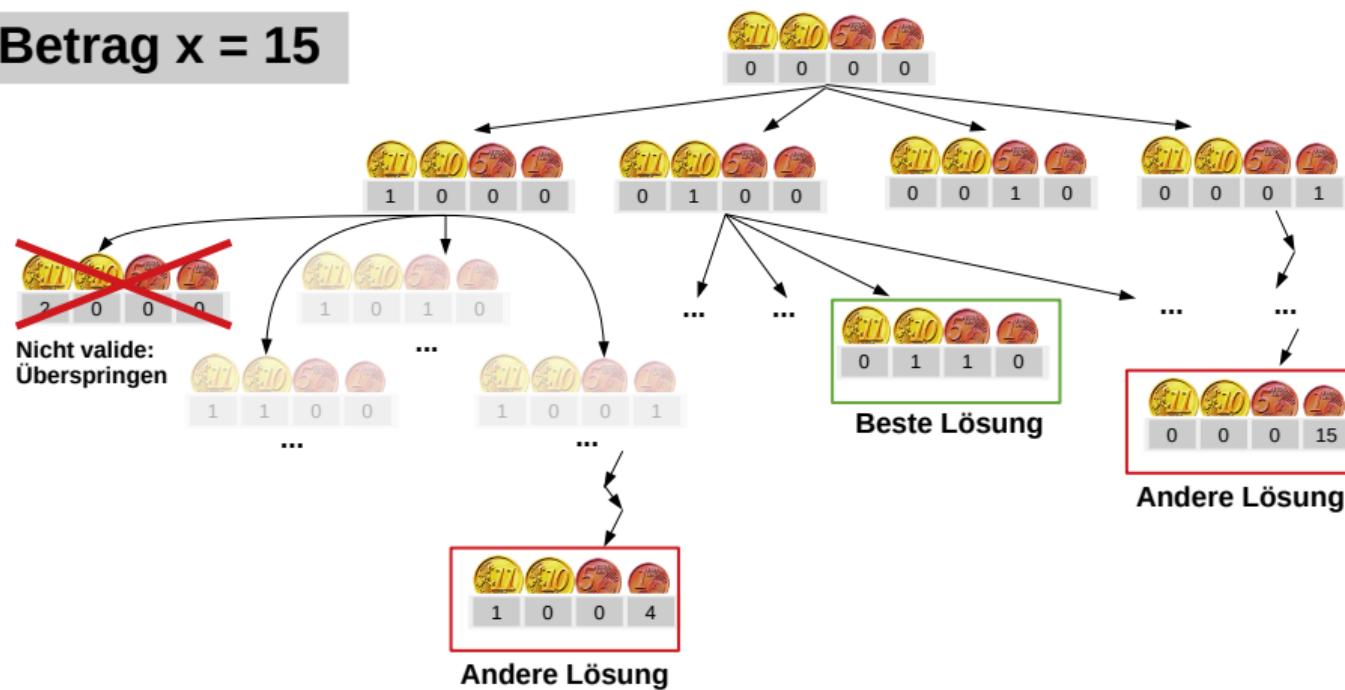
Anmerkungen

- ▶ Backtracking testet alle (*sinnvollen*) Lösungsmöglichkeiten.
- ▶ Die Laufzeit im Worst Case ist oft **exponentiell**
*(Durchsuchen des **kompletten Suchbaums**).*

Backtracking: Wechselgeld-Beispiel

- Gegeben seien die Münzen $[11, 10, 5, 1]$ (siehe oben) und der Betrag $x=15$.
 - **Greedy** fand eine suboptimale Lösung ($1 \times 11, 4 \times 1$).
 - **Backtracking** findet die optimale Lösung ($1 \times 10, 1 \times 5$).

Betrag x = 15





```
# Wechselgeld-Backtracking
function backtrack(x, # Restbetrag
                  zustand, [0,1,3,1,0]
                  beste_loesung):

    if x==0:
        return zustand

    if x<0 or
        sum(zustand) > sum(beste_loesung):
        return None

    Z = {}
    für jede muenze in W:
        z = zustand.copy()
        z[muenze] += 1
        Z.add( (z, muenze) )

    für jedes (z, muenze) ∈ Z:
        loesung = backtrack(x - muenze,
                            z,
                            beste_loesung)

        if sum(loesung) < sum(beste_loesung):
            beste_loesung = loesung

    return beste_loesung

muenzen = [1,5,10,11]
backtrack(x=15, startzustand=[0,0,0,0],
          beste_loesung=None)
```

```
# Generelles Muster: Backtracking
function backtrack(zustand,
                   beste_loesung):

    if ist_loesung(zustand):
        return zustand

    if !ist_valide(zustand) or
        !verbesserung_moeglich(zustand,
                               beste_loesung):
        return None

    Z = folgezustände(zustand)

    für jedes z ∈ Z:
        loesung = backtrack(z, beste_loesung)

        if (güte(loesung) >
            güte(beste_loesung)):
            beste_loesung = loesung

    return beste_loesung

backtrack(startzustand(), None)
```

Backtracking: Wechselgeld-Beispiel

```
# Wechselgeld-Backtracking
function backtrack(x, # Restbetrag
                  zustand,
                  beste_loesung):

    if x==0:
        return zustand

    if x<0
        # bereits zu viele Münzen verwendet
        sum(zustand)>sum(beste_loesung):
            return None

    # Folgezustände
    Z = {}
    für jede muenze in muenzen:
        z = zustand.copy()
        z[muenze] += 1
        Z.add( (z,muenze) )

    für jedes (z,muenze)∈Z:
        loesung = backtrack(x-muenzen[muenze],
                            z,
                            beste_loesung)

        if sum(loesung)<sum(beste_loesung):
            beste_loesung = loesung

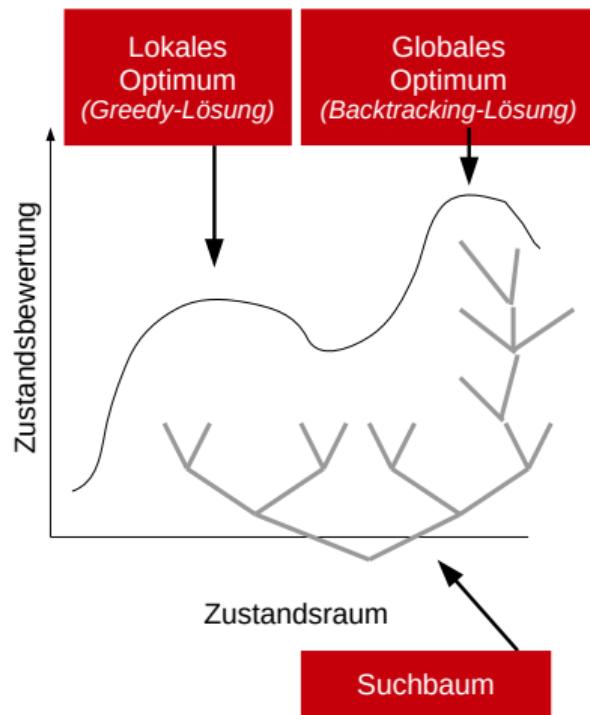
    return beste_loesung

muenzen = [1,5,10,11]
backtrack(x=15, startzustand=[0,0,0,0],
          beste_loesung=None)
```

- ▶ Erhalte Folgezustände indem **jede Münze** einmal hinzugefügt wird.
 - ▶ **Zähle** die Münze hoch.
 - ▶ **Reduziere** den Restbetrag x entsprechend.
- ▶ **Lösung** falls $x == 0$ (*Betrag komplett gewechselt*).
- ▶ **Stoppe** Suche falls $x < 0$ (*Betrag zu groß*), oder falls Lösung mehr Münzen enthält als die optimale Lösung.
- ▶ **Initialer Aufruf:**
 - ▶ Startbetrag x
 - ▶ leere Lösung (*keine Münzen*)
 - ▶ noch keine beste Lösung.

Backtracking: Fazit

Backtracking = Lösen durch potenzielles **Probieren** aller (sinnvollen) Kombinationen.

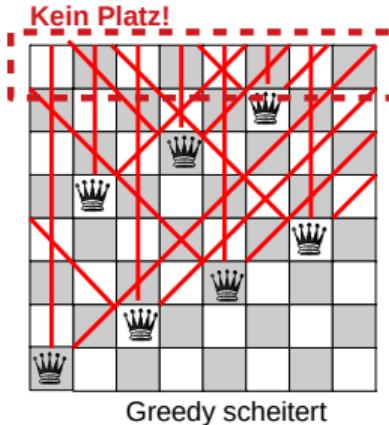


- ▶ **Globale Sicht** ergibt globales Optimum.
- ▶ Für manche Probleme existieren effizientere Lösungen, z.B. **Dynamic Programming** (*Teillösungen in Tabelle gespeichert, deutlich effizienter*).

Backtracking: N-Damen-Problem

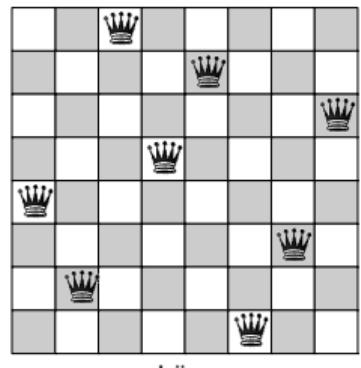
Das N-Damen-Problem

- Gegeben: Ein $N \times N$ -Schachbrett, N Damen.
- Verteile die Damen so auf dem Schachbrett, dass **keine** Dame eine andere **bedroht**.
- Es gibt **keine Gütefunktion**; Alle Lösungen sind gleich gut.
- **Keine analytische Lösung** bekannt → Backtracking.



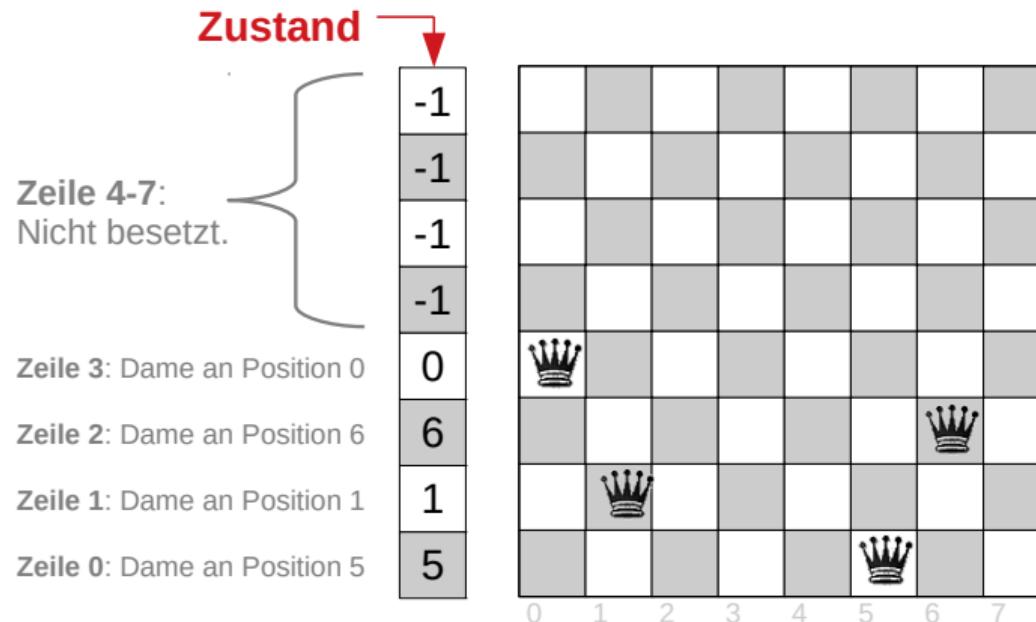
Backtracking-Strategie

- Wir plazieren **eine Dame je Zeile**.
- Durchlufe alle möglichen Zeilen (*z.B. von unten nach oben*).
- Je neuer Zeile werden alle noch nicht bedrohten Positionen **ausprobiert**.



N-Damen-Problem: Formalisierung des Zustandsraums

- Ein **Zustand** besteht aus N Zahlen: Die Positionen der Damen in jeder Zeile.
- Zeile noch **nicht besetzt**: Wert=-1.





N-Damen-Problem: Pseudo-Code

```
# Prüfe ob Dame an Position (zeile,spalte)
# plaziert werden kann.
function belegbar(zustand, zeile, spalte):
    for z in 0,...,zeile-1:
        # schlägt senkrecht
        if zustand[z] == spalte:
            return False
        # schräg von links unten
        if zustand[z] == spalte-(zeile-z):
            return False
        # schräg von rechts unten
        if zustand[z] == spalte+(zeile-z):
            return False

    return True
```

- ▶ Funktion `belegbar()` prüft, ob Dame an Position `(zeile,spalte)` **plaziert werden kann** (*d.h., nicht bedroht wird*).
- ▶ Backtracking geht **zeilenweise** vor. Findet das Backtracking keine Lösung, wird `None` zurückgegeben.

```
# N-Damen-Backtracking
function backtrack(zustand, zeile):

    # Alle Zeilen durchlaufen -> Lösung
    if zeile==N:
        return zustand

    # Folgezustände z: Alle Positionen
    # in nächster Zeile durchprobieren
    for spalte = 0,...,N-1:

        if belegbar(zustand, zeile, spalte):

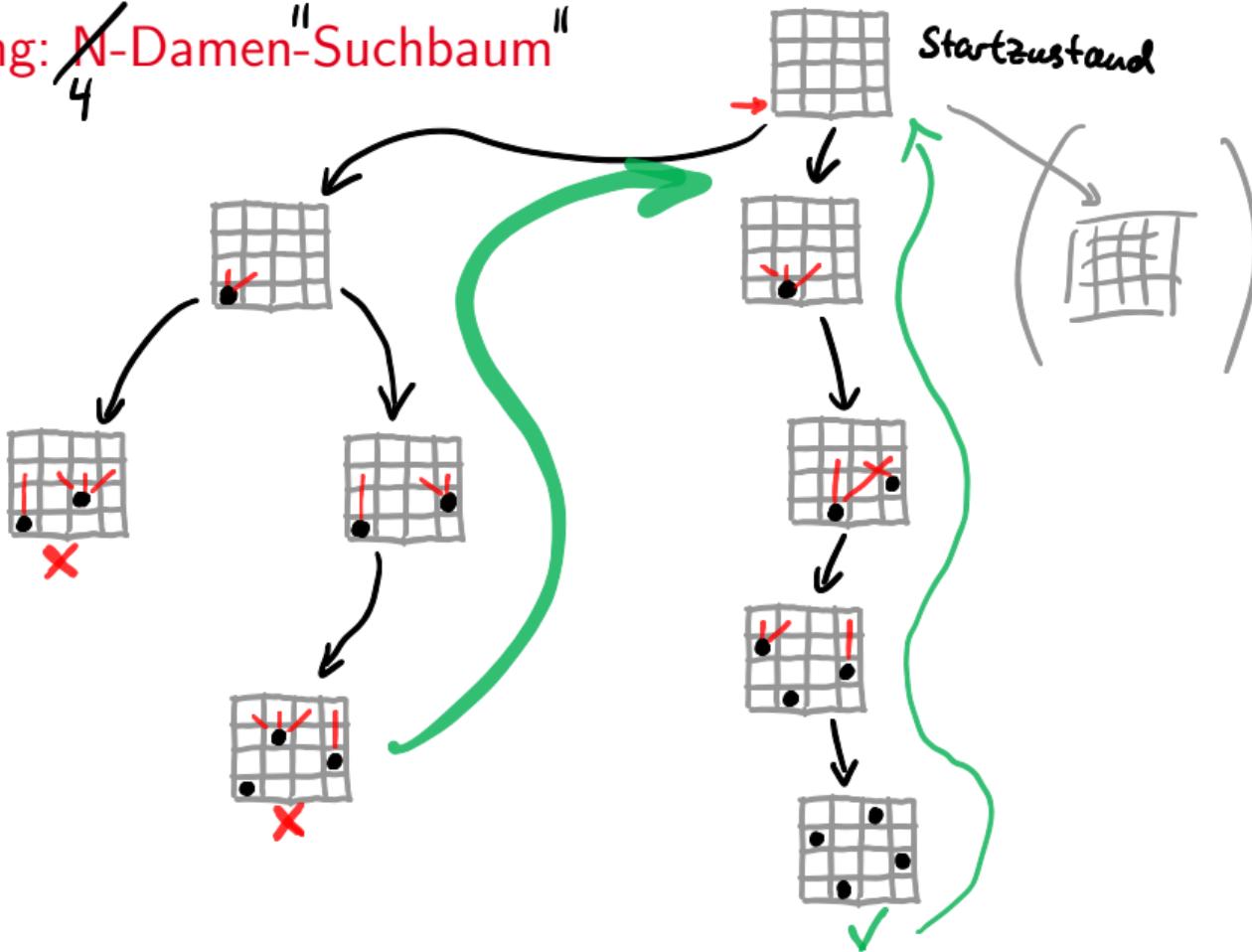
            z = zustand.copy()
            z[zeile] = spalte

            loesung = backtrack(z, zeile+1)

            if loesung <> None:
                return loesung

    # Kein Folgezustand z hat zu Lösung geführt
    return None
```

Backtracking: "N-Damen-Suchbaum"



*

Backtracking: N-Damen-Suchbaum



Outline

1. Divide-and-Conquer
2. Greedy-Verfahren
3. Backtracking
4. Dynamische Programmierung

References I

