Grundlagen der Algorithmen und Datenstrukturen

Kapitel 12

Christian Scheideler + Helmut Seidl

SS 2009

Generische Optimierungsverfahren

Techniken:

- Systematische Suche
  – lass nichts aus
- Divide and Conquer
  – löse das Ganze in Teilen
- Dynamische Programmierung
  – mache nie etwas zweimal
- Greedy Verfahren
  – schau niemals zurück
- Lokale Suche
  – denke global, handle lokal

Systematische Suche

Prinzip: durchsuche gesamten Lösungsraum

Auch bekannt als „Brute Force“

Vorteil: sehr einfach zu implementieren

Nachteil: sehr zeitaufwendig und sollte daher nur für kleine Instanzen verwendet werden

Systematische Suche

Beispiele:

- Suche in unsortierter Liste
- Broadcasting in unstrukturierten verteilten Systemen (Peer-to-Peer Systeme)
- Selection Sort (nimm wiederholt kleinstes Element aus Eingabesequenz)
- Rucksackproblem (siehe nächste Folie)

Systematische Suche

Rucksackproblem:

- Eingabe: n Objekte mit Gewichten $w_1,\ldots,w_n$ und Werten $v_1,\ldots,v_n$ und Rucksack mit Kapazität W
- Ausgabe: Objektmenge M maximalen Wertes, die in Rucksack passt

Anwendungen: Nikolaus, Räuber, …

Systematische Suche

Lösung zum Rucksackproblem:

Probiere alle Teilmengen von Objekten aus und merke die Menge M von Objekten mit $\sum_{i \in M} w_i \leq W$, die bisher den maximalen Wert hatte

Aufwand: $O(2^n)$, da es 2^n Möglichkeiten gibt, Teilmengen aus einer n-elementigen Menge zu bilden.

Divide and Conquer

Prinzip: teile Problem in Teilprobleme auf, die unabhängig voneinander gelöst werden können

Vorteil: Implementierung oft einfach, da rekursiv

Nachteil: eventuell werden Teilprobleme doppelt gelöst

Divide and Conquer

Beispiele:

- Mergesort
- Quicksort
- Binary Search
- Arithmische Operationen wie Matrixmultiplikation oder Multiplikation großer Zahlen
- Nächstes-Paar-Problem

Erinnerung: Master Theorem

Für positive Konstanten a,b,c,d mit $n=b^k$ und eine natürliche Zahl k sei

$$r(n) = a \qquad \text{falls } n=1$$

$$r(n) = c*n + d*r(n/b) \qquad \text{falls } n > 1$$

Dann gilt:

- $r(n) = \Theta(n)$ falls $d < b$
- $r(n) = \Theta(n \log n)$ falls $d = b$
- $r(n) = \Theta(n^{\log_b d})$ falls $d > b$

Master Theorem (komplett)

Für positive Konstanten a,b,d mit $n=b^k$ und eine natürliche Zahl k sei

$$T(n) = a \quad \text{falls } n=1$$

$$T(n) = f(n) + d*T(n/b) \quad \text{falls } n > 1$$

Dann gilt mit $q=\log_b d$:

- $T(n) = \Theta(n^q)$ falls $f(n)=n^p$ und $p<q$
- $T(n) = \Theta(n^q \log n)$ falls $f(n) = \Theta(n^q)$
- $T(n) = O(n^p)$ falls $g(n)=O(n^p)$ und $p>q$

Divide and Conquer

Matrixmultiplikation nach Strassen:

- Eingabe: nxn-Matrizen A und B
- Ausgabe: nxn-Matrix C mit $C=A \cdot B$

Wir nehmen vereinfachend an, n sei Zweierpotenz. Dann A und B darstellbar durch jeweils 4 (n/2)x(n/2)-Matrizen:

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} \qquad B = \begin{pmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{pmatrix}$$

Divide and Conquer

Für Matrix C gilt:

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}$$

mit

- $C_{00} = A_{00} B_{00} + A_{01} B_{10}$
- $C_{01} = A_{00} B_{01} + A_{01} B_{11}$
- $C_{10} = A_{10} B_{00} + A_{11} B_{10}$
- $C_{11} = A_{10} B_{01} + A_{11} B_{11}$

d.h. wir brauchen 8 rekursive Aufrufe!

Geht das eventuell besser ??!

Divide and Conquer

Für Matrix C gilt auch:

$$C = \begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix}$$

mit

- $C_{00} = M_1 + M_4 - M_5 + M_7$
- $C_{01} = M_3 + M_5$
- $C_{10} = M_2 + M_4$
- $C_{11} = M_1 + M_3 - M_2 + M_6$

mit Matrizen $M_1, \ldots, M_7$ wie in nächster Folie

Divide and Conquer

Matrizen $M_1,\ldots,M_7$:

- $M_1=(A_{00}+A_{11})\cdot(B_{00}+B_{11})$
- $M_2=(A_{10}+A_{11})\cdot B_{00}$
- $M_3=A_{00}\cdot(B_{01}-B_{11})$
- $M_4=A_{11}\cdot(B_{10}-B_{00})$
- $M_5=(A_{00}+A_{01})\cdot B_{11}$
- $M_6=(A_{10}-A_{00})\cdot(B_{00}+B_{01})$
- $M_7=(A_{01}-A_{11})\cdot(B_{10}+B_{11})$

Nur 7 rekursive Aufrufe !!!

Divide and Conquer

Laufzeit von Strassens Algorithmus:

$$T(1)=O(1),\ T(n) = 7T(n/2) + O(n^2)$$

Also Laufzeit $O(n^{\log_2 7}) = O(n^{2,81})$

Laufzeit der Schulmethode (brute force):
$O(n^3)$

Divide and Conquer

Nächstes-Paar-Problem:

- Eingabe: Menge S von n Punkten $P_1=(x_1,y_1),\ldots,P_n=(x_n,y_n)$ im 2-dimensionalen Euklidischen Raum
- Ausgabe: Punktpaar mit kürzester Distanz

Annahme: n ist Zweierpotenz

Divide and Conquer

Algo für Nächstes-Paar-Problem:

- Sortiere Punkte gemäß x-Koordinate (z.B. Mergesort, Zeit O(n log n))
- Löse danach Nächstes-Paar-Problem rekursiv durch Algo ClosestPair

Divide and Conquer

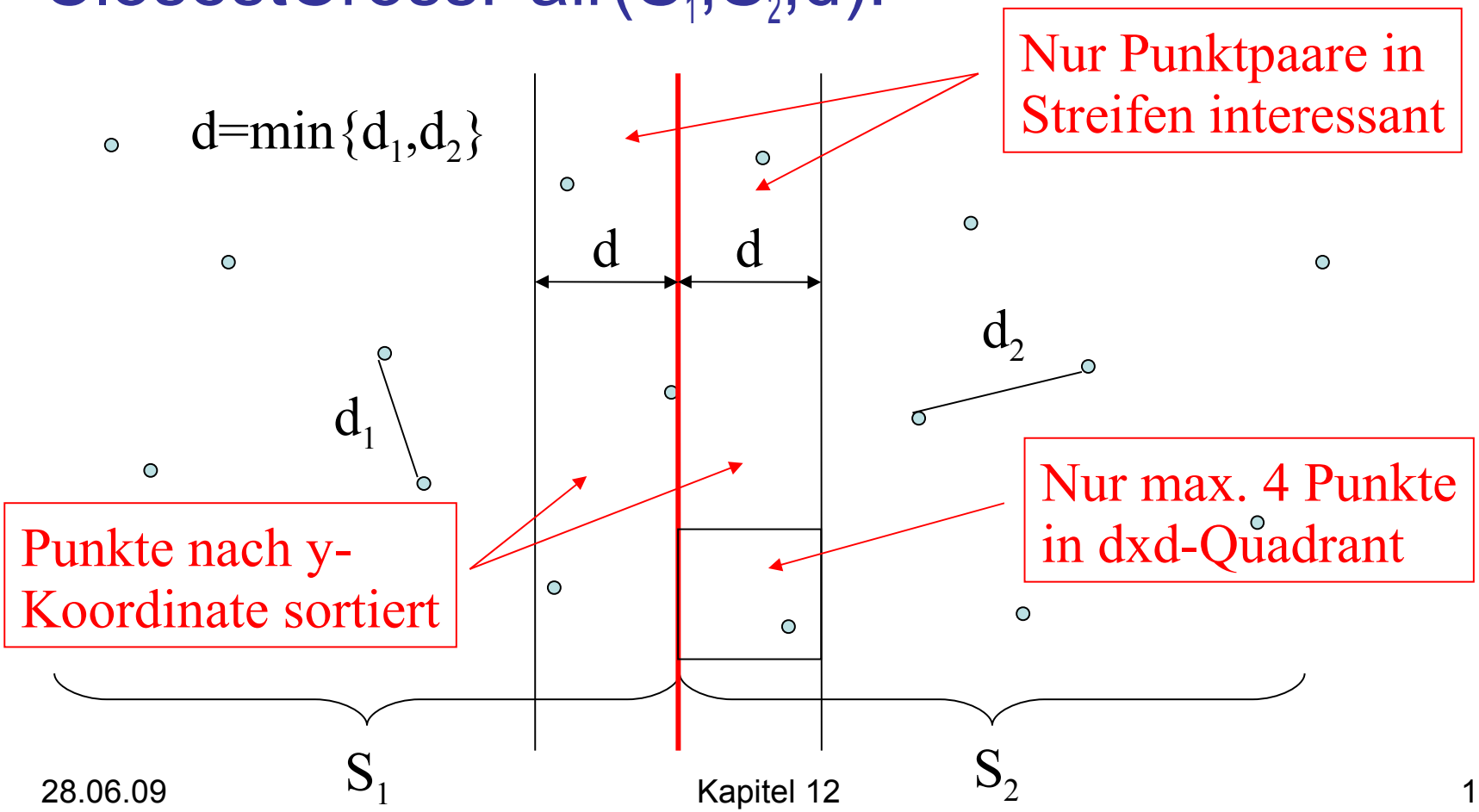
Algo ClosestPair(S):

Eingabe: nach x-Koordinate sortierte Punktmenge S

- |S|=2: sortiere S gemäß y-Koordinate und gib Distanz zwischen zwei Punkten in S zurück
- |S|>2:
  - teile S in der Mitte (Position n/2) in S_1 und S_2
  - d_1=ClosestPair(S_1); d_2=ClosestPair(S_2);
  - d=min{ClosestCrossPair(S_1,S_2,min(d_1,d_2)), d_1, d_2};
  - Führe merge(S_1,S_2) durch, so dass S am Ende nach y-Koordinate sortiert ist (S_1, S_2 bereits sortiert)
  - gib d zurück

Divide and Conquer

ClosestCrossPair(S_1,S_2,d):



Divide and Conquer

ClosestCrossPair:

- Durchlaufe die (nach der y-Koordinate sortierten) Punkte in S_1 und S_2 von oben nach unten (wie in merge(S_1,S_2)) und merke die Mengen M_1 und M_2 der 8 zuletzt gesehenen Punkte in S_1 und S_2 im d-Streifen
- Bei jedem neuen Knoten in M_1, berechne Distanzen zu Knoten in M_2, und bei jedem neuen Knoten in M_2, berechne Distanzen zu Knoten in M_1
- Gib am Ende minimal gefundene Distanz zurück

Divide and Conquer

Laufzeit für Nächstes-Paar-Algo:

- Mergesort am Anfang: $O(n \log n)$
- Laufzeit $T(n)$ von ClosestPair Algo:

$$T(1)=O(1),\ T(n)=2T(n/2)+O(n)$$

Gesamtlaufzeit: $O(n \log n)$

Brute force: $O(n^2)$

Dynamische Programmierung

Prinzip: rekursive Durchsuchung des Lösungsraums mittels Speicher

Vorteil: verhindert doppelte Berechnung von Teillösungen

Nachteil: benötigt eventuell sehr viel Speicher

Dynamische Programmierung

Beispiele:

- Berechnung des Binomialkoeffizienten
- Rucksackproblem

Dynamische Programmierung

Berechnung des Binomialkoeffizienten:

Eingabe: $n,k \geq 0,\ n \geq k$

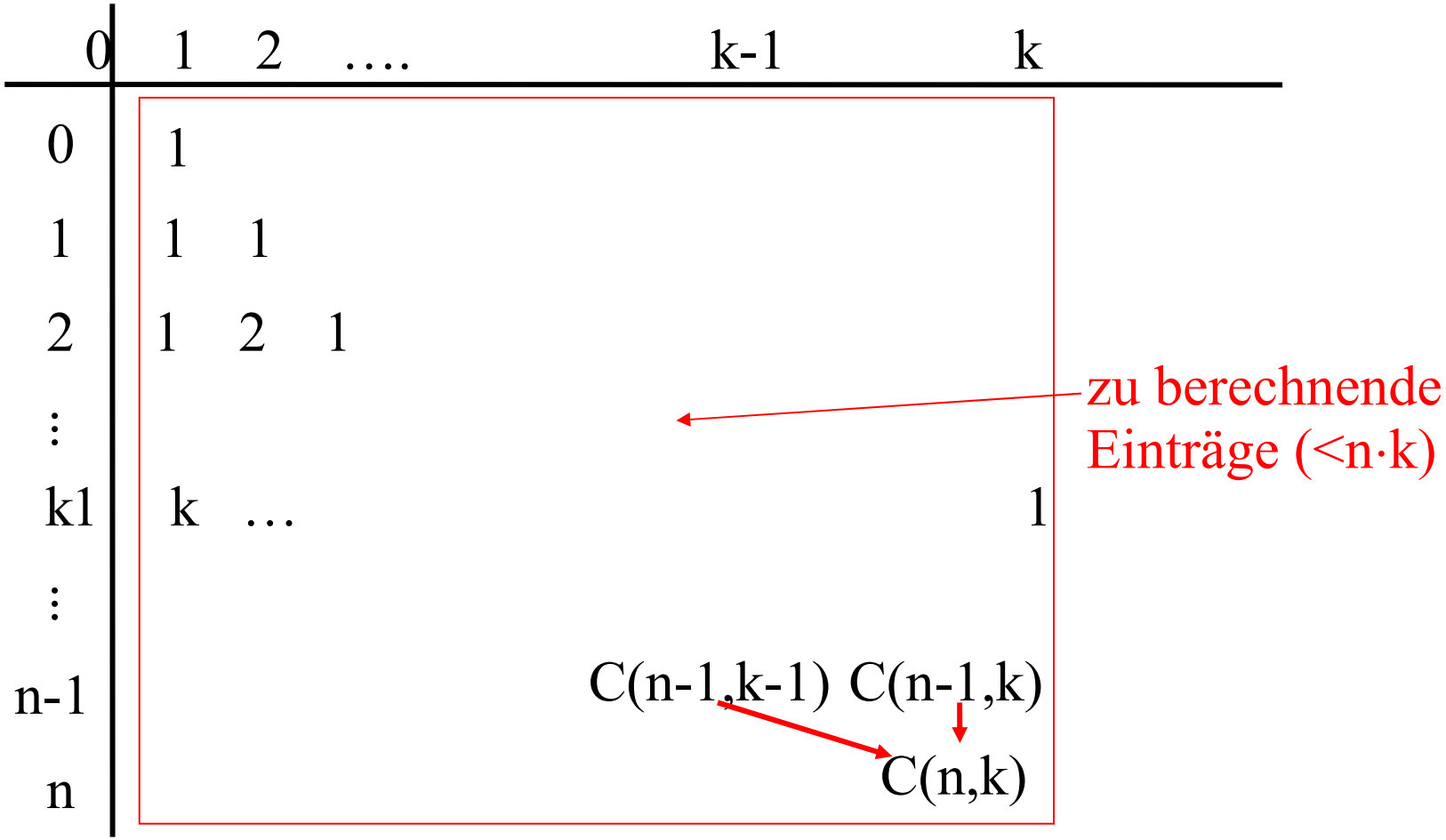
Ausgabe: $C(n,k)$ mit $C(n,k) = \binom{n}{k}$

Rekursive Formel für $C(n,k)$:

$C(n,0)=C(n,n)=1$ für alle $n \geq 0$

$C(n,k)=C(n-1,k-1)+C(n-1,k)$ für $n>k>0$

Dynamische Programmierung



Dynamische Programmierung

Berechnung des Binomialkoeffizienten:

```
for (i=0;i≤n;i++)
    for (j=0;j ≤ min(i,k);j++)
        if (j==0 || j==i) C[i][j] =1;
        else C[i][j] = C[i-1][j-1]+C[i-1][j];
return C[n][k];
```

Laufzeit: $O(n \cdot k)$

Dynamische Programmierung

Rucksackproblem:

- Eingabe: n Objekte mit Gewichten $w_1,\ldots,w_n$ und Werten $v_1,\ldots,v_n$ und Rucksack mit Kapazität W
- Ausgabe: Objektmenge M maximalen Wertes, die in Rucksack passt

Dynamische Programmierung

V[i][w]: optimaler Wert, der für die Objekte 1 bis i bei Rucksackvolumen w erreichbar ist

Rekursive Berechnung von V[i][w]:

- Für die Teilmengen, die das i-te Objekt nicht enthalten, ist der Wert der optimalen Lösung gleich V[i-1][w]
- Für die Teilmengen, die Objekt i enthalten, ist der Wert der optimalen Lösung gleich v_i+V[i-1][w-w_i]

Dynamische Programmierung

Rekursive Formel für V[i][w]:

V[0][w]=0 für alle $w \geq 0$

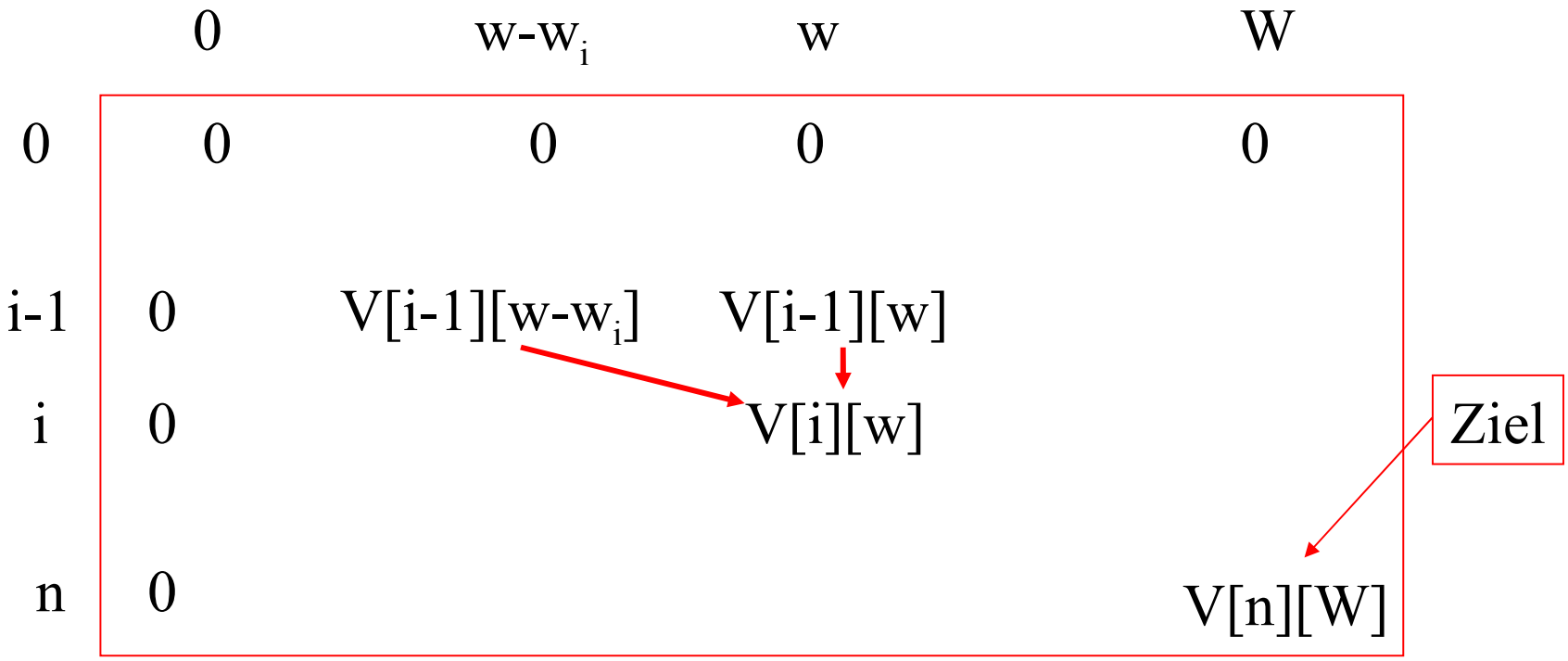
V[i][0]=0 für alle $i \geq 0$

V[i][w]=V[i-1][w] falls $w-w_i<0$

V[i][w]=max{V[i-1][w], v_i+V[i-1][w-w_i]} sonst

Dynamische Programmierung

Rekursive Berechnung von V[i][w]:



Dynamische Programmierung

Berechnung der Tabelle: $O(n \cdot W)$ Zeit

Brute force: $O(2^n)$

Fazit: solange W klein, viel bessere Laufzeit als brute force. Falls W groß, dann unterteile Objekte in große ($w_i \geq 1/(4\varepsilon^2 W)$) und kleine Objekte. Wende auf die großen Objekte dynamische Programmierung und auf die kleinen eine Greedy Methode an, die wir später noch beschreiben werden. Damit ist in Zeit $O(n \cdot \text{poly}(1/\varepsilon))$ Lösung ermittelbar, die um maximal $1+\varepsilon$ von Optimum abweicht.

Greedy Verfahren

Prinzip: teile Problem in mehrere kleine Entscheidungen auf, nimm immer lokales Optimum für jede kleine Entscheidung

Vorteil: sehr schnell

Nachteil: kann weit von optimaler Lösung entfernt sein

Greedy Verfahren

Beispiele:

- Dijkstra-Algorithmus für kürzeste Wege
- Jarnik-Prim-Algorithmus für minimale Spannbäume
- Huffman-Bäume
- Rucksackproblem

Greedy Verfahren

Huffman-Baum:

- Eingabe: Wahrscheinlichkeitsverteilung p auf einem Alphabet A
- Ausgabe: optimaler Präfixcode $f:A \rightarrow \{0,1\}^*$ für A, d.h. $\sum_{a \in A} f(a) \cdot p(a)$ minimal

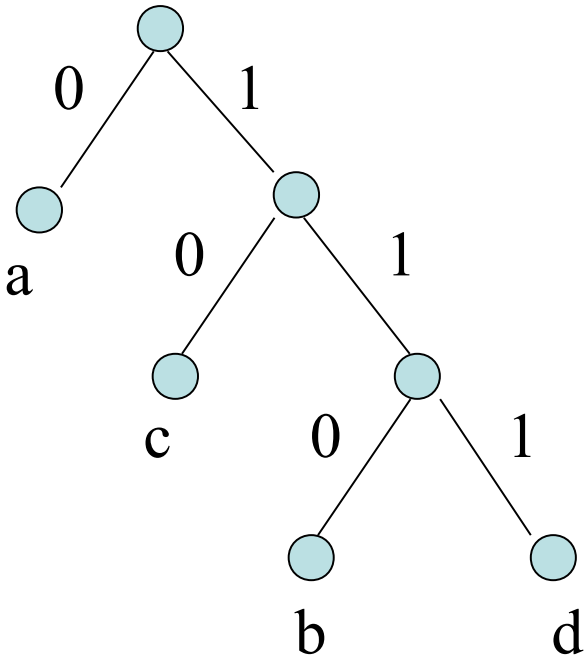
Präfixcode: Kodierung $f:A \rightarrow \{0,1\}^*$, für die es keine Zeichen $a,b \in \Sigma$ gibt, für die f(a) ein Präfix von f(b) ist (d.h. f(b) anfängt mit f(a))

Präfixcodes wichtig für eindeutige Dekodierung!

Greedy Verfahren

Einsicht: jeder Präfixcode lässt sich als Binärbaum darstellen mit Zeichen in A an Blättern

Beispiel: A={a,b,c,d}



Kodierung f:

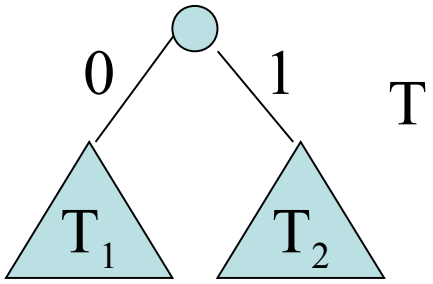
- f(a) = 0
- f(b) = 110
- f(c) = 10
- f(d) = 111

Greedy Verfahren

Huffman-Baum: Baum für optimale Kodierung

Strategie:

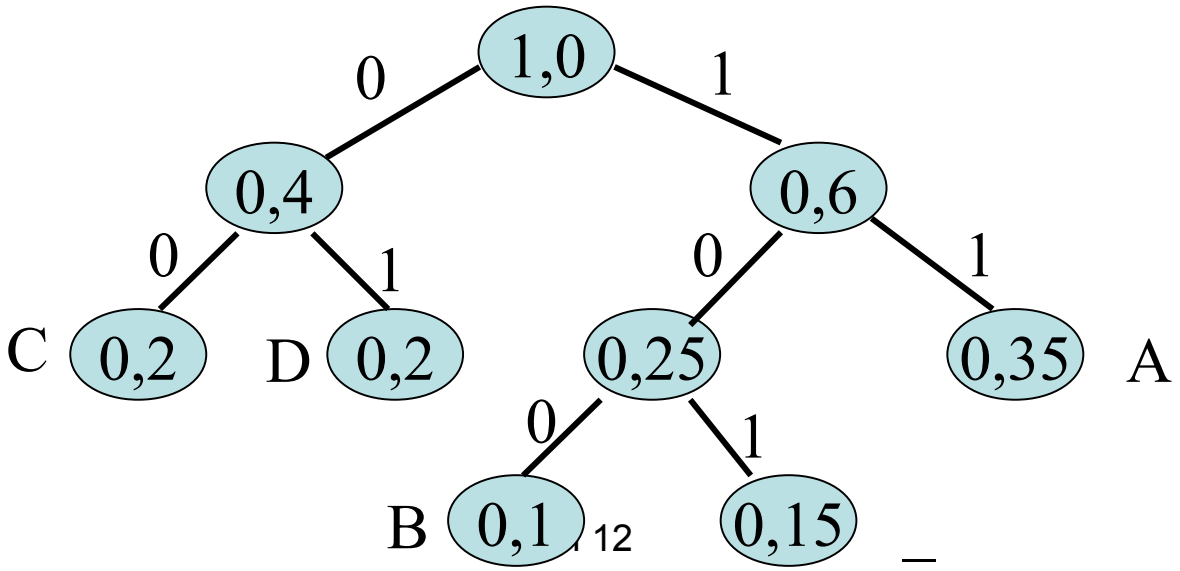
- Anfangs ist jedes Zeichen in A ein Baum für sich
- Wiederhole, bis ein einziger Baum übrig:
  - Bestimme die beiden Bäume T_1 und T_2 mit kleinsten Wahrscheinlichkeitswerten $\sum_a p(a)$ ihrer Zeichen
  - Verbinde T_1 und T_2 zu neuem Baum T



Greedy Verfahren

Beispiel für Huffman-Baum:

Zeichen	A	B	C	D	_
Wkeit p	0,35	0,1	0,2	0,2	0,15



Greedy Verfahren

Rucksackproblem:

- Eingabe: n Objekte mit Gewichten $w_1,\ldots,w_n$ und Werten $v_1,\ldots,v_n$ und Rucksack mit Kapazität W
- Ausgabe: Objektmenge M maximalen Wertes, die in Rucksack passt

Greedy Verfahren

Greedy Strategie:

- Berechne Profitdichten $d_1=v_1/w_1,..,\ d_n=v_n/w_n$
- Sortiere Objekte nach Profitdichten
- Angefangen von dem Objekt mit höchster Profitdichte, füge Objekte zu Rucksack hinzu, bis kein Platz mehr da

Problem: Greedy Strategie kann weit vom Optimum entfernt liegen

Greedy Verfahren

Beispiel: zwei Objekte mit $v_1=1$ und $v_2=W-1$ und $w_1=1$ und $w_2=W$, Rucksackkapazität ist W

Greedy-Methode: berechnet $d_1=1$ und $d_2 = 1-1/W$ und wird nur Objekt 1 in Rucksack packen, da Objekt 2 nicht mehr passt

Optimale Lösung: packe Objekt 2 in Rucksack (viel besser da Wert $W-1$ statt 1)

Greedy Verfahren

Verbesserte Greedy-Methode:

- Seien die Objekte 1 bis n absteigend nach Profitdichte sortiert
- Bestimme maximale Objektmenge {1,…,i} wie bisher mit $\sum_{j<=i} w_j \leq W$
- Gib entweder {1,…,i} oder {i+1} aus, je nachdem, welche Menge den maximalen Wert hat

Greedy Verfahren

Behauptung: die Lösung der verbesserten Greedy-Methode ist höchstens einen Faktor 2 von der optimalen Lösung entfernt

Beweis:

- Wenn beliebige Bruchteile der Objekte gewählt werden könnten, wäre die optimale Lösung {1, …,i+1}, wobei von Objekt i+1 nur der Bruchteil genommen wird, der noch in den Rucksack passt.
- Für den optimalen Wert OPT gilt demnach: $OPT \leq \sum_{j<=i+1} v_j$.
- Also ist $\max\{\sum_{j<=i} v_j, v_{i+1}\} \geq OPT/2$

Lokale Suche

Generische Verfahren:

- Backtracking
- Branch-and-Bound
- Hill climbing
- Simulated annealing
- Tabu search
- Evolutionäre Verfahren

Backtracking

Prinzip: systematische Tiefensuche im Lösungsraum, bis gültige Lösung gefunden

Vorteil: einfach zu implementieren

Nachteil: kann sehr lange dauern

Backtracking

Beispiele:

- n-Damen Problem
- Hamiltonscher Kreis Problem

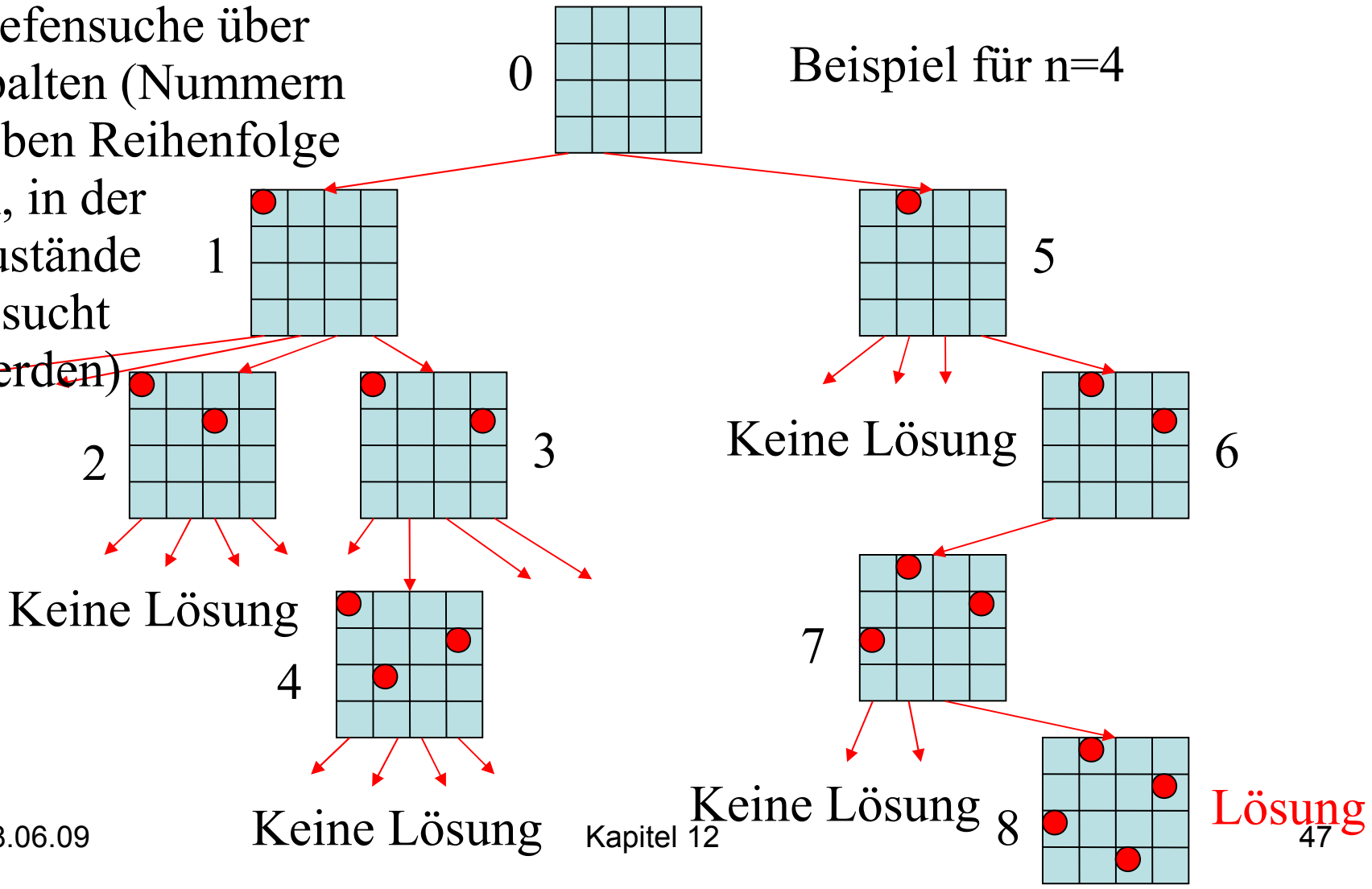
Backtracking

n-Damen Problem:

- Eingabe: $n \in \mathbb{N}$
- Ausgabe: Antwort auf die Frage, ob sich n Damen so auf einem nxn-Schachbrett stellen lassen, dass keine die andere schlagen kann

Backtracking

Tiefensuche über Spalten (Nummern geben Reihenfolge an, in der Zustände besucht werden)

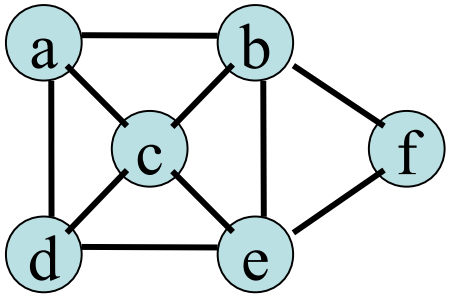


Backtracking

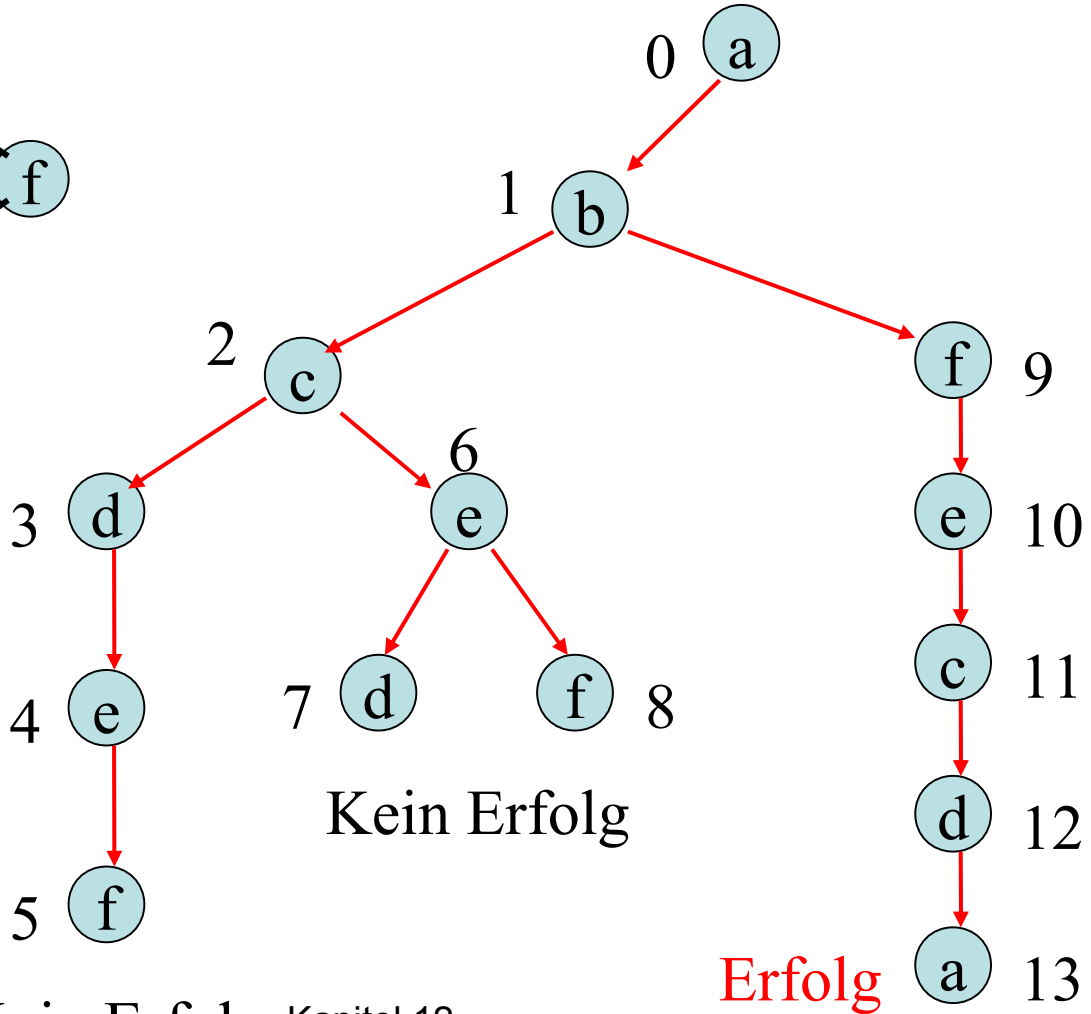
Hamiltonscher Kreis Problem:

- Eingabe: ungerichteter Graph G=(V,E)
- Ausgabe: Kreis $C \subseteq E$, der jeden Knoten genau einmal besucht, falls dieser existiert

Backtracking



G=(V,E)



Kein Erfolg

Branch and Bound

Prinzip: systematische Breitensuche im Lösungsraum (mit Bewertung der Zustände), bis optimale Lösung gefunden

Vorteil: relativ einfach zu implementieren

Nachteil: kann sehr lange dauern