Kapitel 6 - Algorithmische Paradigmen

Backtracking
Divide and Conquer
Sweeplinetechnik
Heuristiken
Greedy Algorithmen
Dynamische Programmierung
Branch and Bound
Online-Algorithmen

Paradigma: Backtracking

Idee:

- Lösung eines Problems durch Versuch und Irrtum (trial and error) .
- Schrittweises „Herantasten“ an die Gesamtlösung.
- Kann eine Teillösung nicht zu einer Gesamtlösung erweitert werden:
  - Letzten Schritt rückgängig machen.
  - Weitere, alternative Schritte probieren.

Voraussetzung:

- Die Lösung setzt sich aus „Komponenten zusammen“
- Für jede Komponente gibt es mehrere Wahlmöglichkeiten
- Teillösungen können getestet werden.

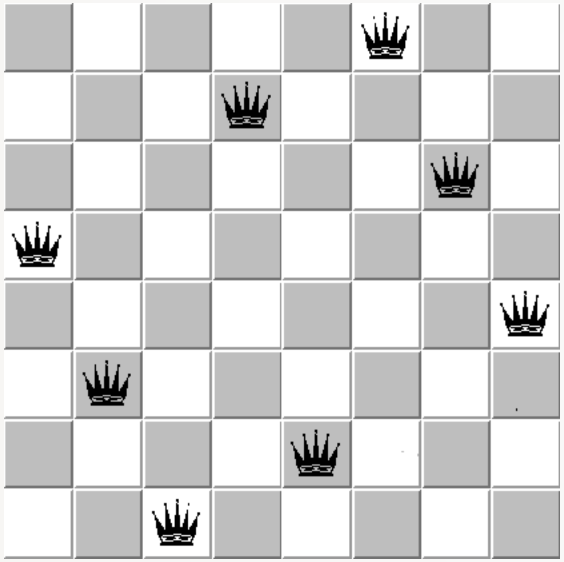
Geeignete Probleme:

- *Damenproblem*
- Sudoku
- Solitär-Brettspiel
- Wegsuche von A nach B

Damenproblem

Problem:

Auf einem Schachfeld der Größe n * n sollen n Damen so positioniert werden, dass sie sich gegenseitig nicht schlagen können.



Für *n = 8 gibt es:*

- $\binom{64}{8} = 4\,426\,165\,368$ Positionierungen.
- 92 Lösungen.
- 12 nicht unter Symmetrie äquivalente Lösungen

Optimierte Suche:

In jeder Spalte und in jeder Zeile kann jeweils nur eine Dame stehen.
Somit lassen sich die Positionierungen auf *8! = 40 320* reduzieren.

n-Damenproblem

Für n = 26 wurden im Jahre 2009 alle Lösungen berechnet:

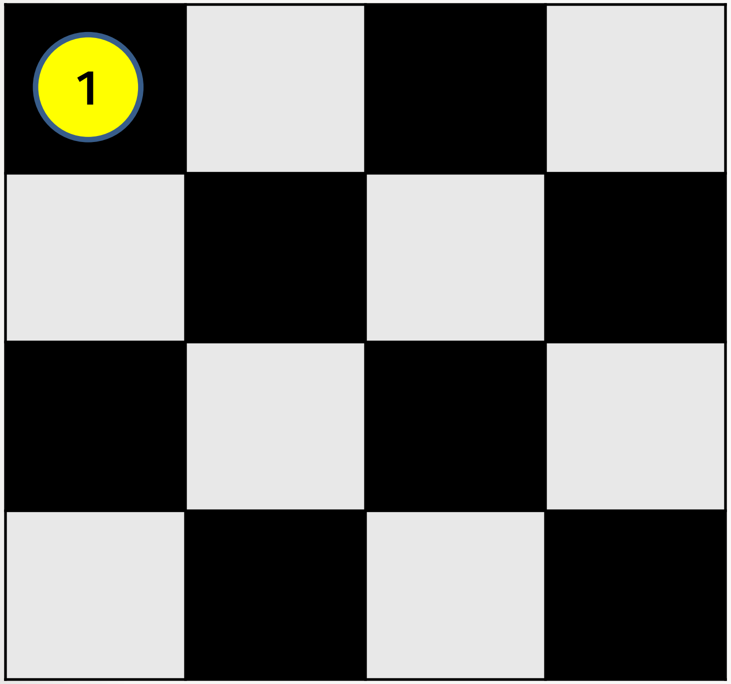
• Lösungen insgesamt	22.317.699.616.364.044
• Eindeutige Lösungen	2.789.712.466.510.289

Für n = 27 wurden im Jahre 2016 alle Lösungen berechnet:

• Lösungen insgesamt	234.907.967.154.122.528
• Eindeutige Lösungen	29.363.495.934.315.694

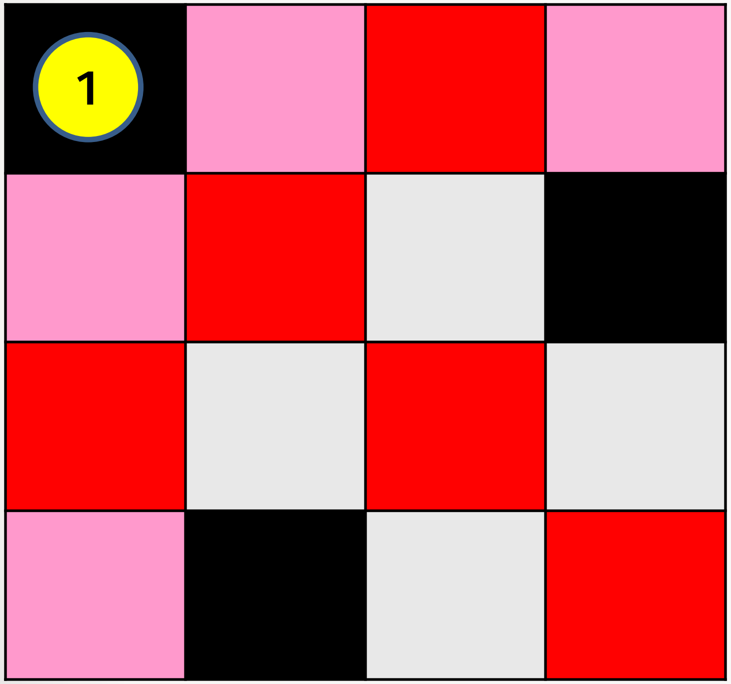
- Die Berechnung aller Lösungen für n = 28 steht noch aus.

4 Damenproblem



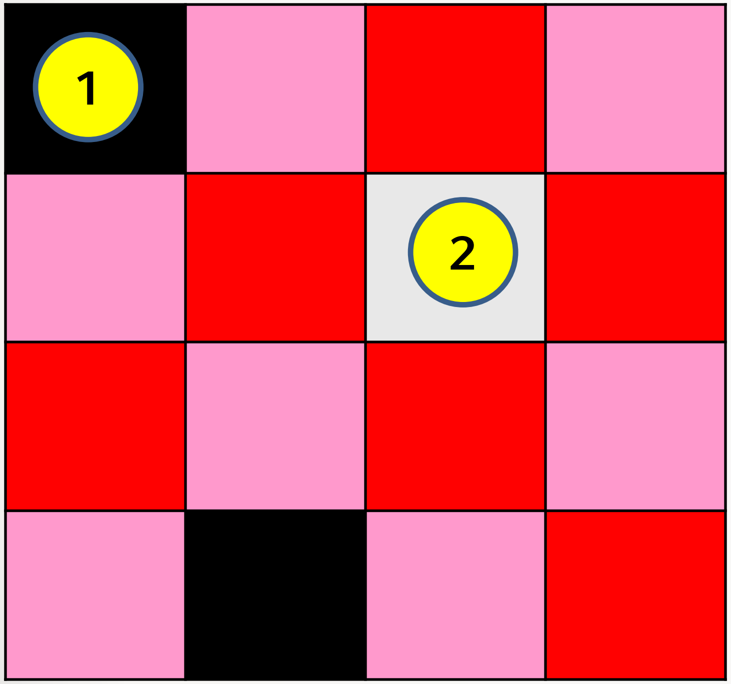
- Erste Dame auf das erste Feld setzen

4 Damenproblem



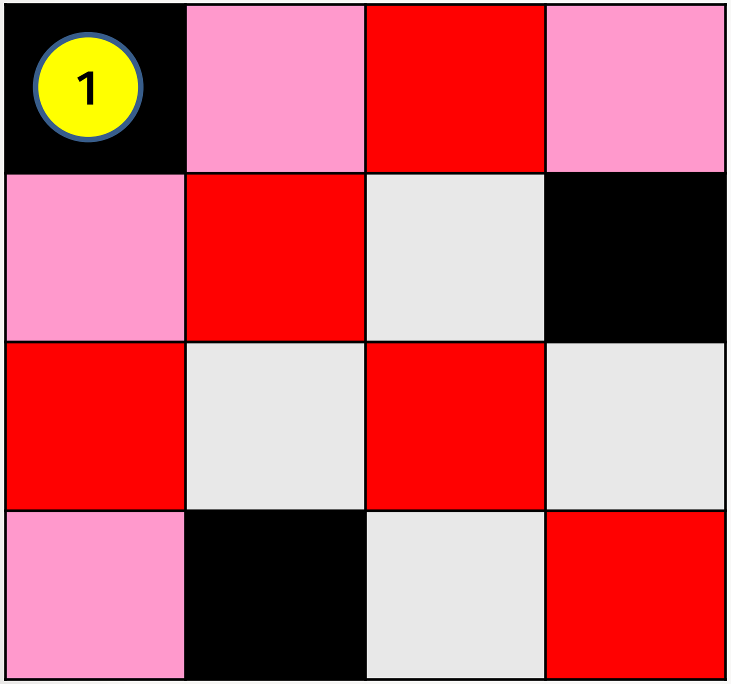
- Erste Dame auf das erste Feld setzen
- Markierte Felder sind bedroht

4 Damenproblem



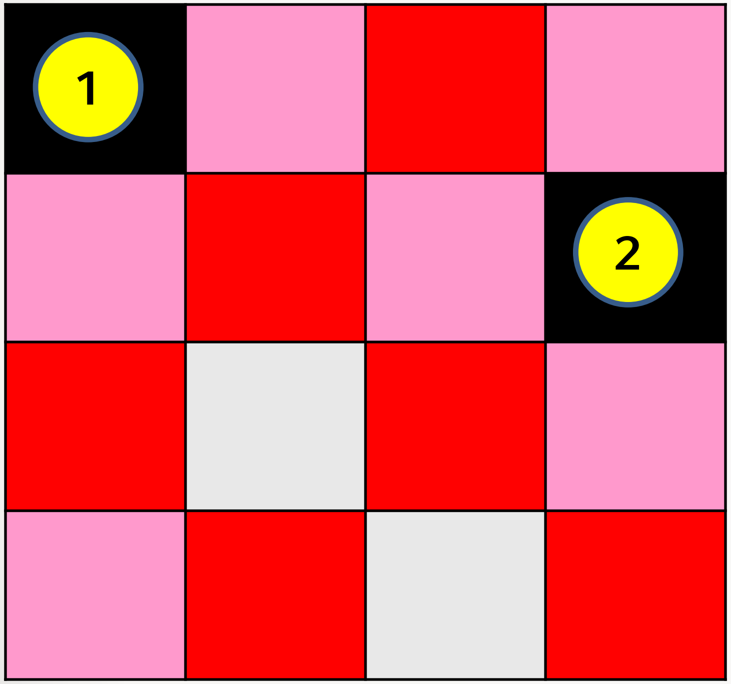
- Erste Dame auf das erste Feld setzen
- Markierte Felder sind bedroht
- Zweite Dame auf erstes freies Feld setzen
- Dritte Reihe komplett bedroht!
- Keine Lösung möglich.
- Backtracking ist nötig

4 Damenproblem



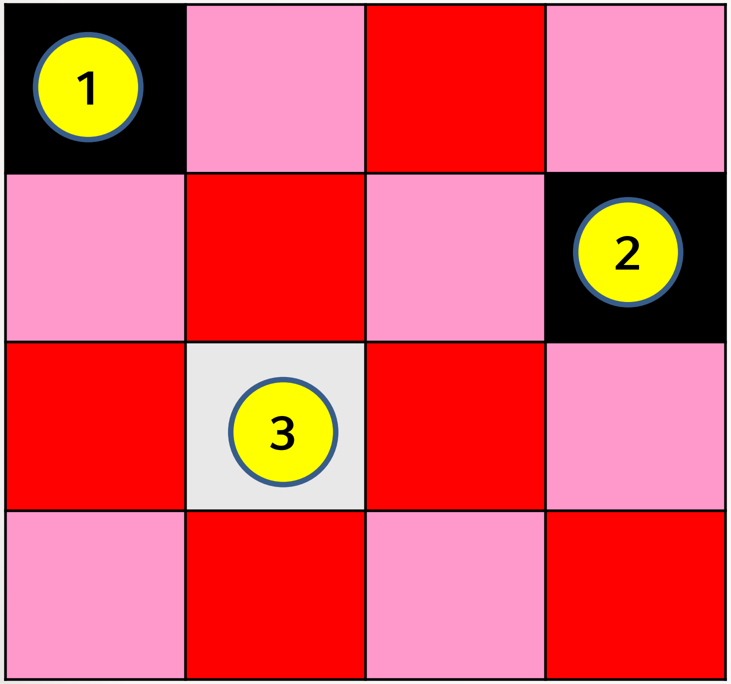
- …
- Backtracking ist nötig
- Dame zwei wieder entfernen

4 Damenproblem



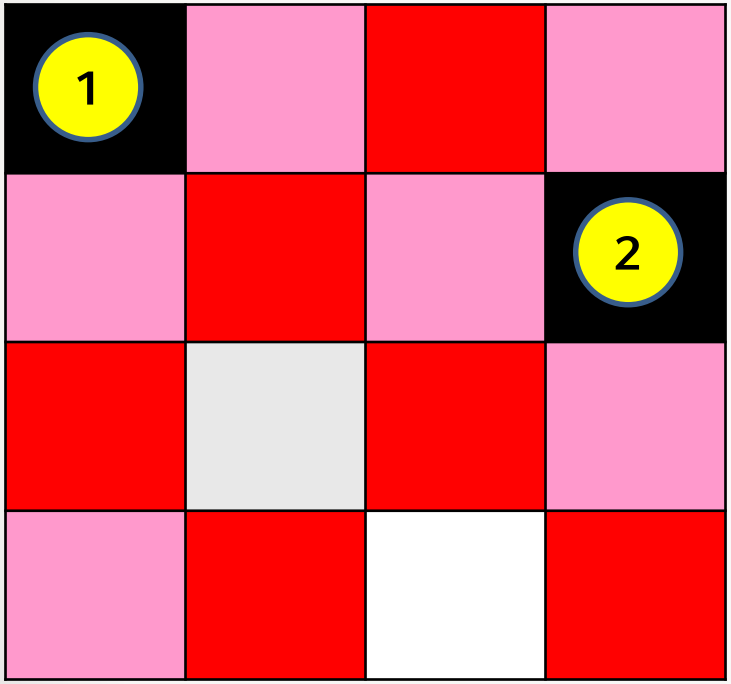
- …
- Backtracking ist nötig
- Dame zwei wieder entfernen
- Zweite Dame auf das nächste freie Feld

4 Damenproblem



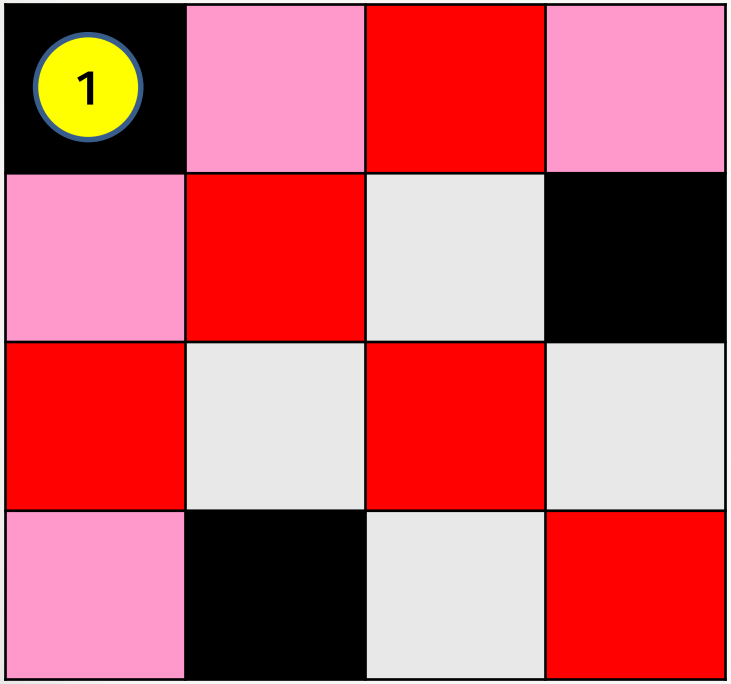
- …
- Backtracking nötig
- Dame zwei wieder entfernen
- Zweite Dame auf das nächste freie Feld
- Dritte Dame auf das erste freie Feld
- Kein freies Feld für vierte Dame!
- Backtracking …

4 Damenproblem



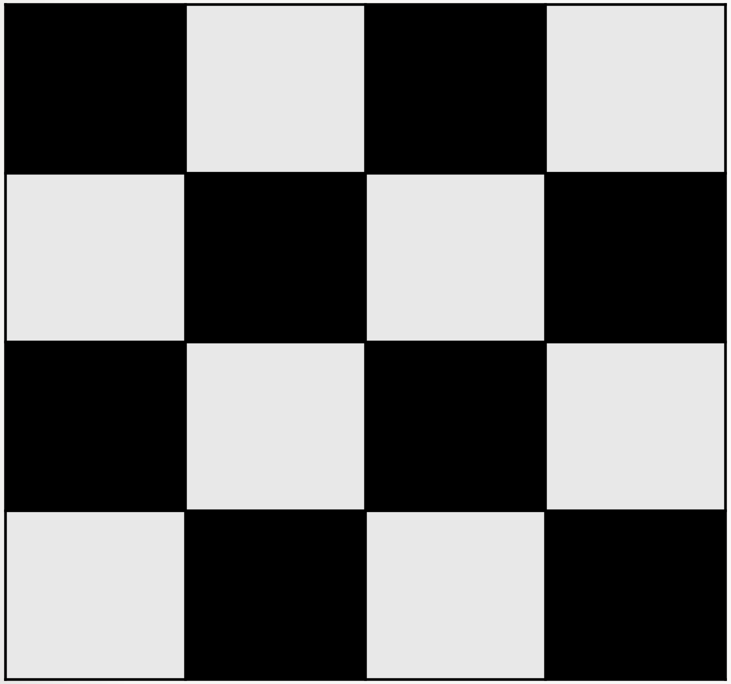
- …
- Kein Weiteres Feld in der dritten Reihe frei
- Weiteres Backtracking

4 Damenproblem



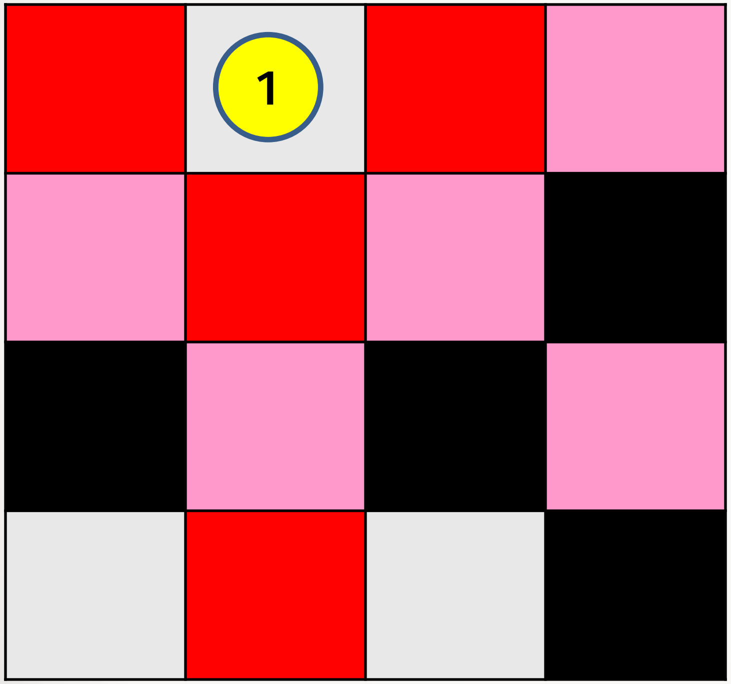
- …
- Kein Weiteres Feld in der dritten Reihe frei
- Weiteres Backtracking
- Kein weiteres Feld in der zweiten Reihe frei
- Weiteres Backtracking …

4 Damenproblem



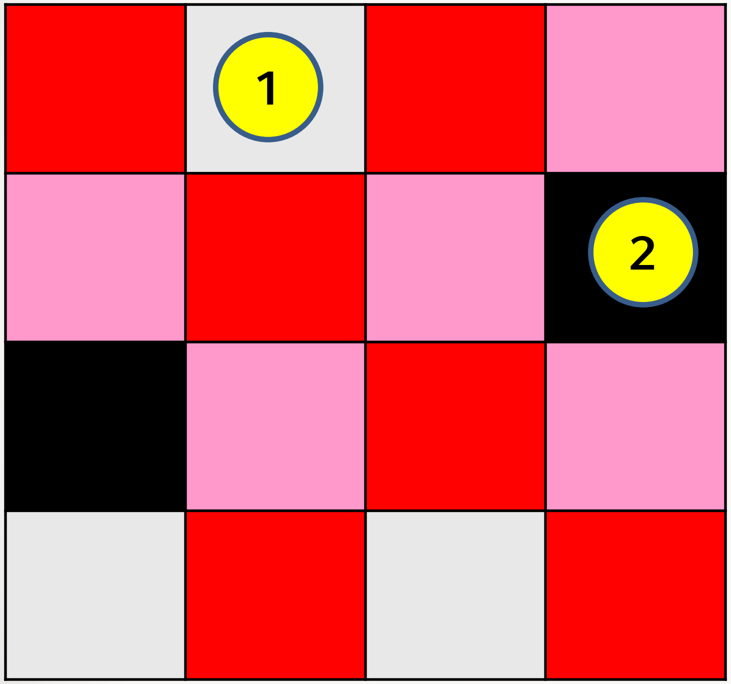
- …
- Kein Weiteres Feld in der dritten Reihe frei
- Weiteres Backtracking
- Kein weiteres Feld in der zweiten Reihe frei
- Weiteres Backtracking …

4 Damenproblem



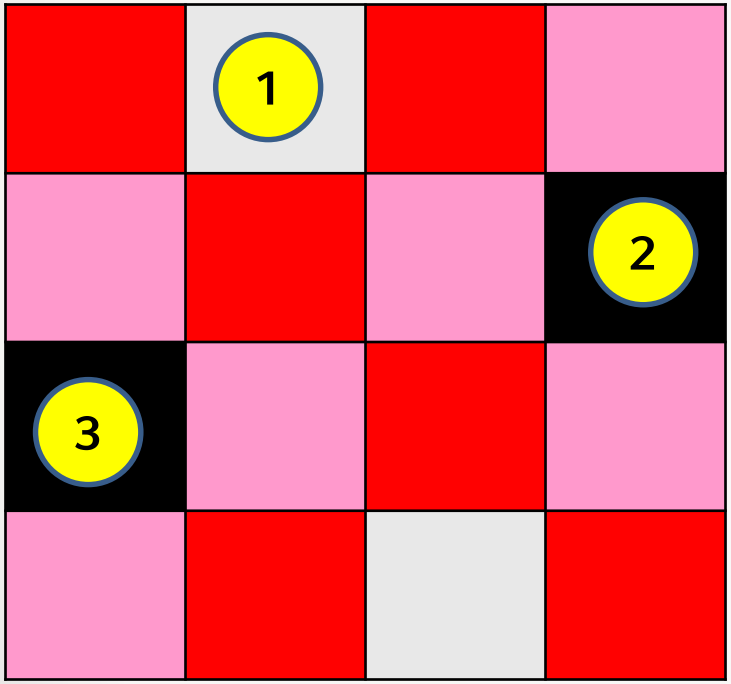
- …
- Erste Dame auf zweites Feld setzen

4 Damenproblem



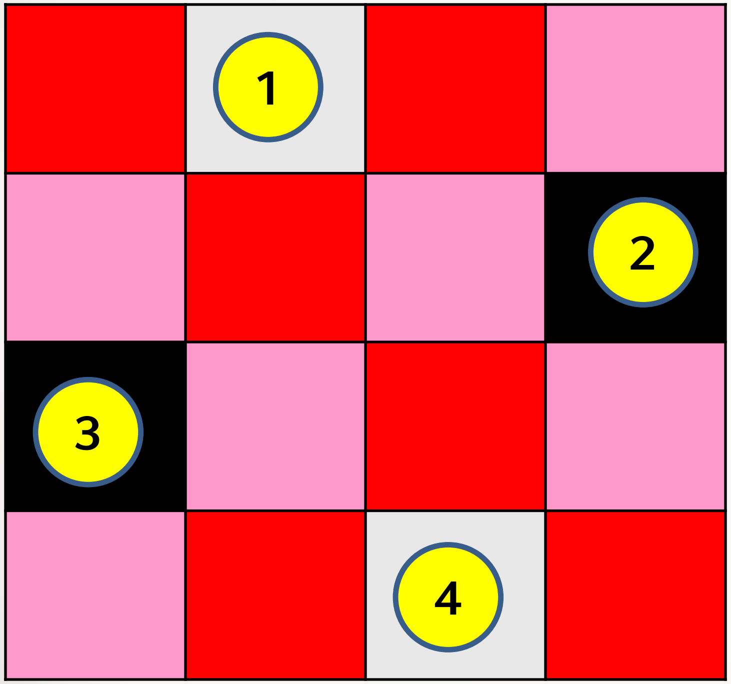
- …
- Erste Dame auf zweites Feld setzen
- Zweite Dame auf erstes freies Feld setzen

4 Damenproblem



- …
- Erste Dame auf zweites Feld setzen
- Zweite Dame auf erstes freies Feld setzen
- Dritte Dame auf erstes freies Feld setzen

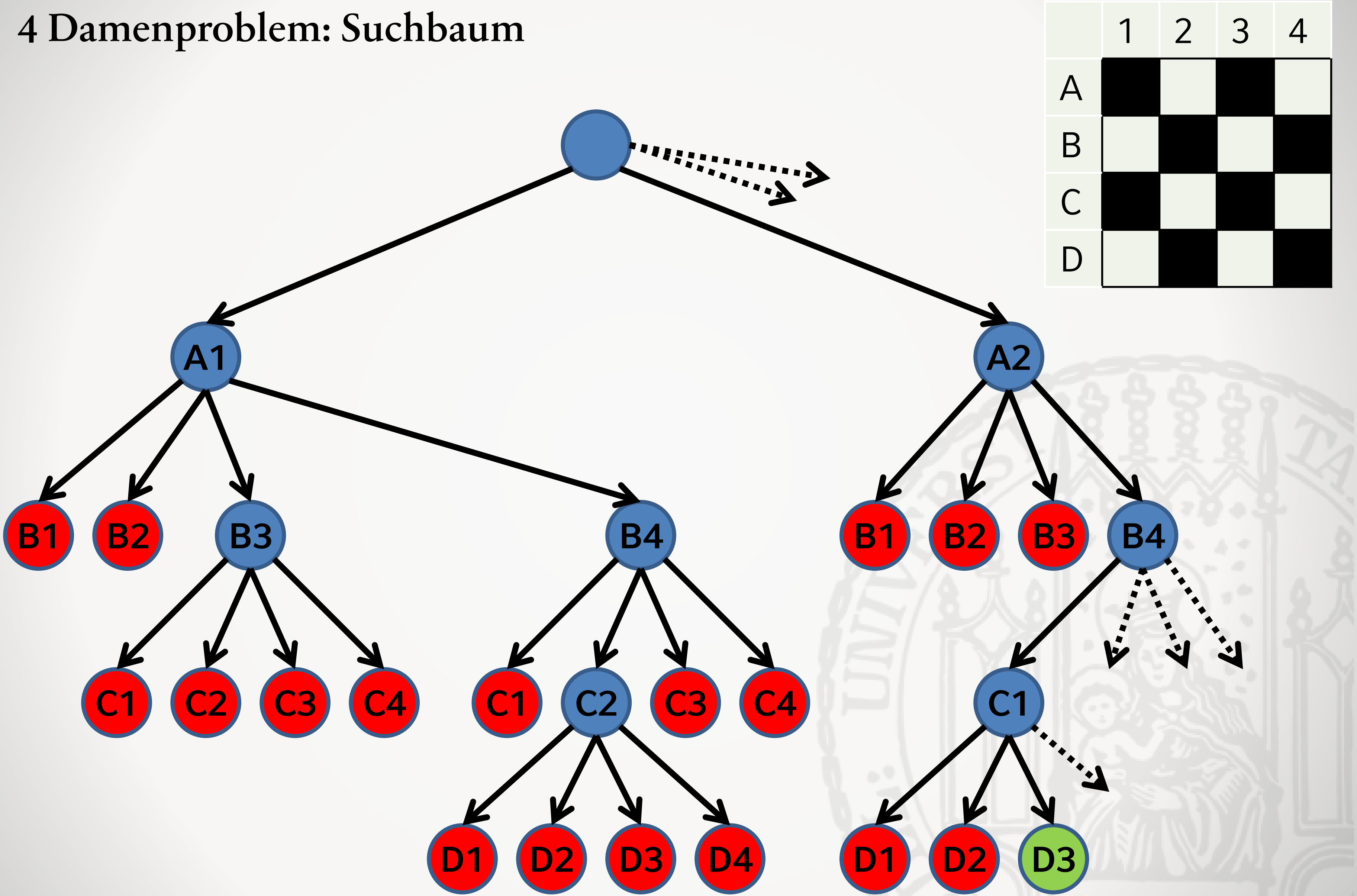
4 Damenproblem



- …
- Erste Dame auf zweites Feld setzen
- Zweite Dame auf erstes freies Feld setzen
- Dritte Dame auf erstes freies Feld setzen
- Vierte Dame auf erstes freies Feld setzen

- Es wurde eine Lösung gefunden!

4 Damenproblem: Suchbaum



n- Damenproblem in Java

```
public static boolean damen( boolean[][] feld, int reihe){

  if(bedroht (feld))              // testen ob Teillösung ungültig
    return false;
  if(reihe == feld.length)        // testen ob Gesamtlösung erreicht
    return true;

  for(int i = 0; i < feld[reihe].length; i++){

      feld[reihe][i] = true;        // versuchen …
      if(damen(feld, reihe+1))
        return true;
      else
        feld[reihe][i] = false;   // backtracking

  }
  return false;
}
```

Aufruf mit:
damen(feld, 0)

Backtracking- Allgemeiner Algorithmus

```
BackTracking ( Stufe, Lösungsvektor )
    falls Teillösung ungültig gib falsch zurück
    falls alle Komponenten gesetzt sind gib richtig zurück
    solange es auf der aktuellen Stufe  noch Wahlmöglichkeiten gibt:
        wähle einen neuen Teil-Lösungsschritt
        setze die entsprechende Komponente
        falls BackTracking ( Stufe+1,  Lösungsvektor ) gib richtig zurück
        sonst mache Wahl rückgängig
    Wenn es keinen neuen Teil-Lösungsschritt mehr gibt: Keine Lösung!
```

Optimierungsprobleme

Unter Optimierungsproblemen versteht man Probleme mit den folgenden Eigenschaften:

- Es gibt eine Menge von Lösungen
- Es gibt eine Bewertungsfunktion für Lösungen
- Gesucht ist eine beste Lösung (beste Bewertung)

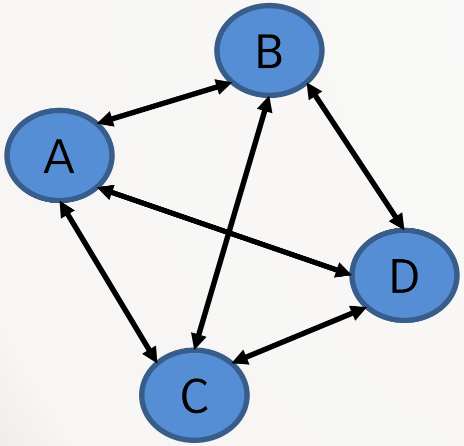
Bekannte Optimierungsprobleme:

- Kürzester Weg im Graph
- *Rucksackproblem*
- Maximaler Fluss in einem Netzwerk
- *Traveling Salesperson Problem (TSP)*
- Minimaler Spannbaum

Traveling Salesperson Problem (TSP)

Kombinatorisches Problem:

- Ein Handlungsreisender plant eine Rundreise durch mehrere Städte.
- Start und Ziel der Rundreise ist eine vorgegebene Stadt.
- Jede Stadt soll nur einmal besucht werden (Ausnahme Start und Ziel)
- Die Kosten der Reise sollen minimal sein.
- In welcher Reihenfolge müssen die Städte besucht werden?



Entfernungstabelle:

nach

von

	A	B	C	D
A	---	10	15	20
B	5	---	9	10
C	6	13	---	12
D	8	8	9	---

Eine optimale Route mit Kosten von ***35*** verläuft über: ***A -> B -> D -> C -> A***

Traveling Salesperson Problem (TSP)

Anwendung:

- Tourenplanung in der Logistik
- Design von Mikrochips
- Genom-Sequenzierung
- …

Problem:

- Anzahl der Rundreisen: $(n-1)\cdot(n-2)\cdot\ldots\cdot 2\cdot 1 = (n-1)!$

1.307.674.368.000	für 15 Knoten
15.511.210.043.330.985.984.000.000	für 25 Knoten

→ Durchprobieren (Brute Force) für großes n praktisch nicht möglich

- Tatsächlich existiert bis heute kein exakter Algorithmus, der das TSP effizient (in nicht-exponentieller Zeit) löst
- Man geht in der Informatik davon aus, dass ein solcher auch nicht existieren kann („NP vollständig“)

Wie gehen wir vor um ein solches NP-vollständiges Problem zu lösen?

Paradigma: Branch and Bound

Idee:

- Methode zur Lösung von (NP-vollständigen) Optimierungsproblemen, bei denen keine anderen effizienten Verfahren bekannt sind.
- Wie bei Backtracking (trial and error).
- Schrittweises Annähern an die Gesamtlösung.
- Für die Teillösungen werden Schranken (Bounds) berechnet.
- Teillösungen werden dann verworfen, wenn das Optimum nicht mehr erreicht werden kann.

Vorgehensweise:

- Berechne Bound b der leeren Lösung v und füge $\mathrm{x} = (v, b)$ in die Liste Q ein.
- *Wiederhole:*
  - Entferne (v, b) mit minimalem Bound b aus Q
  - Falls x vollständige Lösung *gib diese aus*
  - Sonst: Erweitere x, bestimme Bound b_w für jede Erweiterung w und füge (w, b_w) in Q ein.

Branch für TSP

Aufteilung des Lösungsraums in:

- Rundreisen, die nicht direkt von i nach j gehen und
- Rundreisen, die direkt von i nach j gehen

Wird in der Entfernungstabelle $M[i,j]$ auf „–“ gesetzt, sind nur noch Rundreisen möglich, die Kante (i,j) nicht enthalten.

Somit kann die Aufteilung des Lösungsraums wie folgt realisiert werden:

Branch 1: Rundreisen die nicht direkt von i nach j gehen:

Setze: $M[i,j] = -$, d.h. Entfernen der Kante (i,j)

Branch 2: Rundreisen die direkt von i nach j gehen:

Setze: $M[i,k] = -$, d.h. Entfernen der Kante (i,k) für $k \neq j$

Setze: $M[l,j] = -$, d.h. Entfernen der Kante (l,j) für $l \neq i$

Setze: $M[j,i] = -$, d.h. Entfernen der Kante (j,i)

→ Ein Baum entsteht, dessen Blätter die möglichen Pfade repräsentieren.

Entscheidungsbaum: Branch and Bound

- Ziel: Verwerfen von Teillösungen

  → Zweige des Baumes werden nicht weiter verfolgt

- Zwei Möglichkeiten:
  - Tiefensuche um erste vollständige Lösung zu erhalten (in Blatt)
    - Die Lösung dient als obere Schranke des Ergebnisses
    - Alle Teillösungen deren Bound (untere Schranke) bereits über der Schranke liegen werden verworfen
  - Heuristische obere Schranke bereits in inneren Knoten des Baums

Einfacher Bound für TSP

- Gesucht ist eine möglichst große **untere Schranke** für die Kosten.
  - →Je genauer, desto näher am Optimum.
- **Idee:** Jede Stadt wird genau ein Mal **verlassen.**
- Aus jeder Zeile wird *genau ein* Wert entnommen.
- Die Summe der minimalen Werte jeder Zeile ist eine untere Schranke
- Gleiches gilt für die minimalen Werte der Spalten.

nach

von

	A	B	C	D	***min***
A	---	10	15	20	10
B	5	---	9	10	5
C	6	13	---	12	6
D	8	8	9	---	8
				Summe	***29***

Kann man diese Schranke noch verbessern?

Verbesserter Bound für TSP

Idee: Gibt es Städte, bei welchen noch nicht die minimalen Kosten für das **Betreten** angenommen wurden?

Reduzierung der Matrix:

- Bestimmung des Minimums m_i der Einträge der Zeile `i`
- m_i wird von allen Einträgen in der Zeile `i` abgezogen
- Wir reduzieren jede Zeile der Entfernungsmatrix `M` und erhalten die Matrix `M‘` sowie die Werte $m_i, \ldots, m_n$ als Minima der Zeilen von `M` und $m_{n+1}, \ldots, m_{2n}$ als Minima der Spalten von `M‘`.
- Der Bound entspricht $\sum_{i=1}^{2n} m_i$.

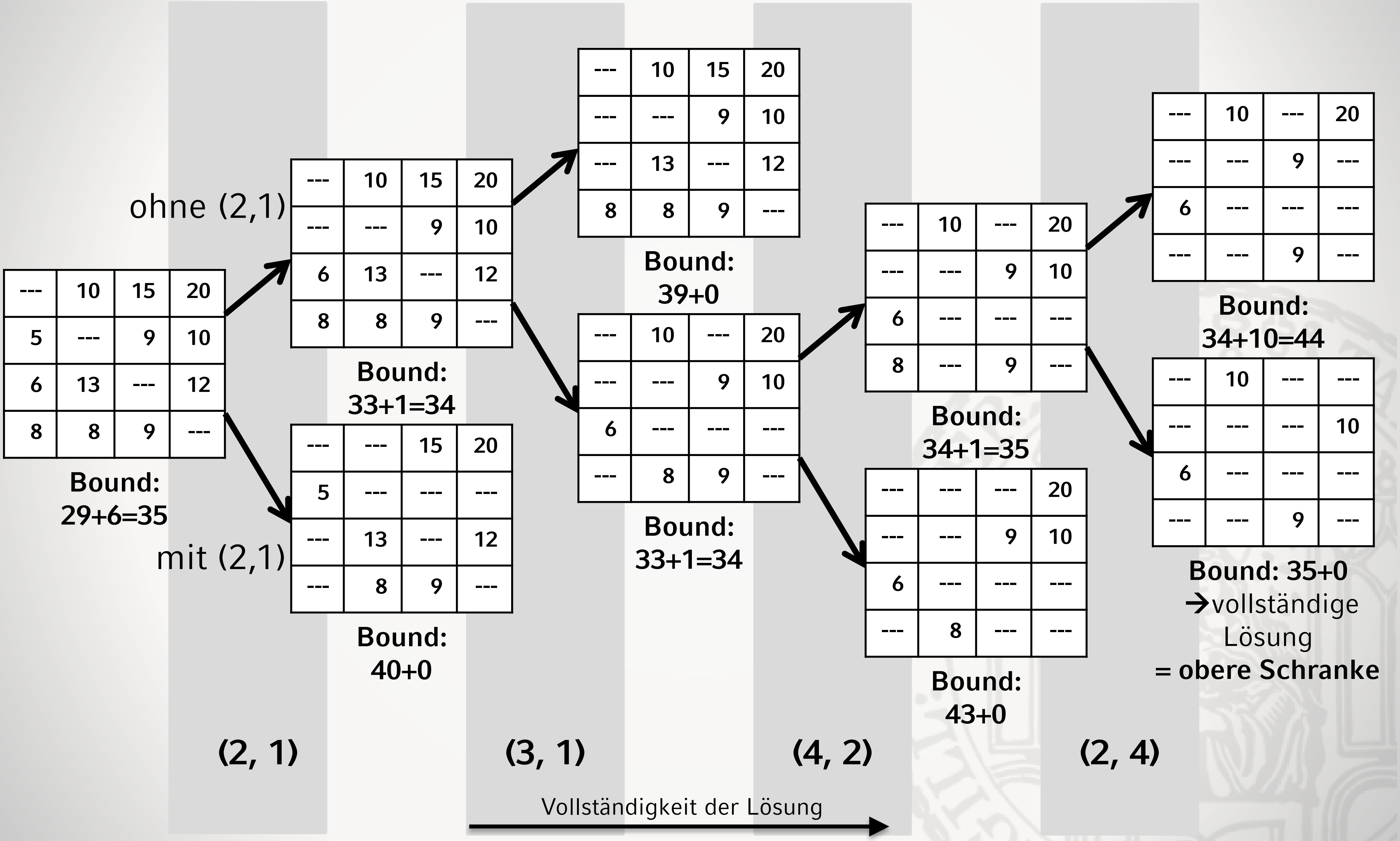
Beispiel:

---	10	15	20	***10***
5	---	9	10	***5***
6	13	---	12	***6***
8	8	9	---	***8***
				29

---	0	5	10	
0	---	4	5	
0	7	---	6	
0	0	1	---	
0	***0***	***1***	***5***	**6**

Bound: ***29 + 6 = 35***

Branch and Bound für TSP



---	10	15	20
5	---	9	10
6	13	---	12
8	8	9	---

Bound: 29+6=35

ohne (2,1)

---	10	15	20
---	---	9	10
6	13	---	12
8	8	9	---

Bound: 33+1=34

mit (2,1)

---	---	15	20
5	---	---	---
---	13	---	12
---	8	9	---

Bound: 40+0

---	10	15	20
---	---	9	10
---	13	---	12
8	8	9	---

Bound: 39+0

---	10	---	20
---	---	9	10
6	---	---	---
---	8	9	---

Bound: 33+1=34

---	10	---	20
---	---	9	10
6	---	---	---
8	---	9	---

Bound: 34+1=35

---	---	---	20
---	---	9	10
6	---	---	---
---	8	---	---

Bound: 43+0

---	10	---	20
---	---	9	---
6	---	---	---
---	---	9	---

Bound: 34+10=44

---	10	---	---
---	---	---	10
6	---	---	---
---	---	9	---

Bound: 35+0
→vollständige
Lösung
= obere Schranke

(2, 1) **(3, 1)** **(4, 2)** **(2, 4)**

Vollständigkeit der Lösung →

Paradigma: Divide-and-Conquer

Idee:

Große Probleme in kleinere zerteilen, die leichter zu lösen sind und die Gesamtlösung ermöglichen

Drei Schritte:

Divide:	Das Problem wird in mehrere Unterprobleme aufgeteilt
Conquer:	Die (kleineren) Unterprobleme werden rekursiv gelöst. Sind die Unterprobleme klein genug werden sie direkt gelöst.
Merge:	Die Lösungen der Teilprobleme werden zu einer Gesamtlösung kombiniert.

Divide-and-Conquer Algorithmen

Merge-Sort:

Divide: Die zu sortierende Liste der Länge n wird in zwei zu sortierende Listen der Länge $n/2$ aufgeteilt.

Conquer: Die Listen werden weiterhin in kleinere Listen aufgeteilt. Einelementige Listen sind automatisch sortiert.

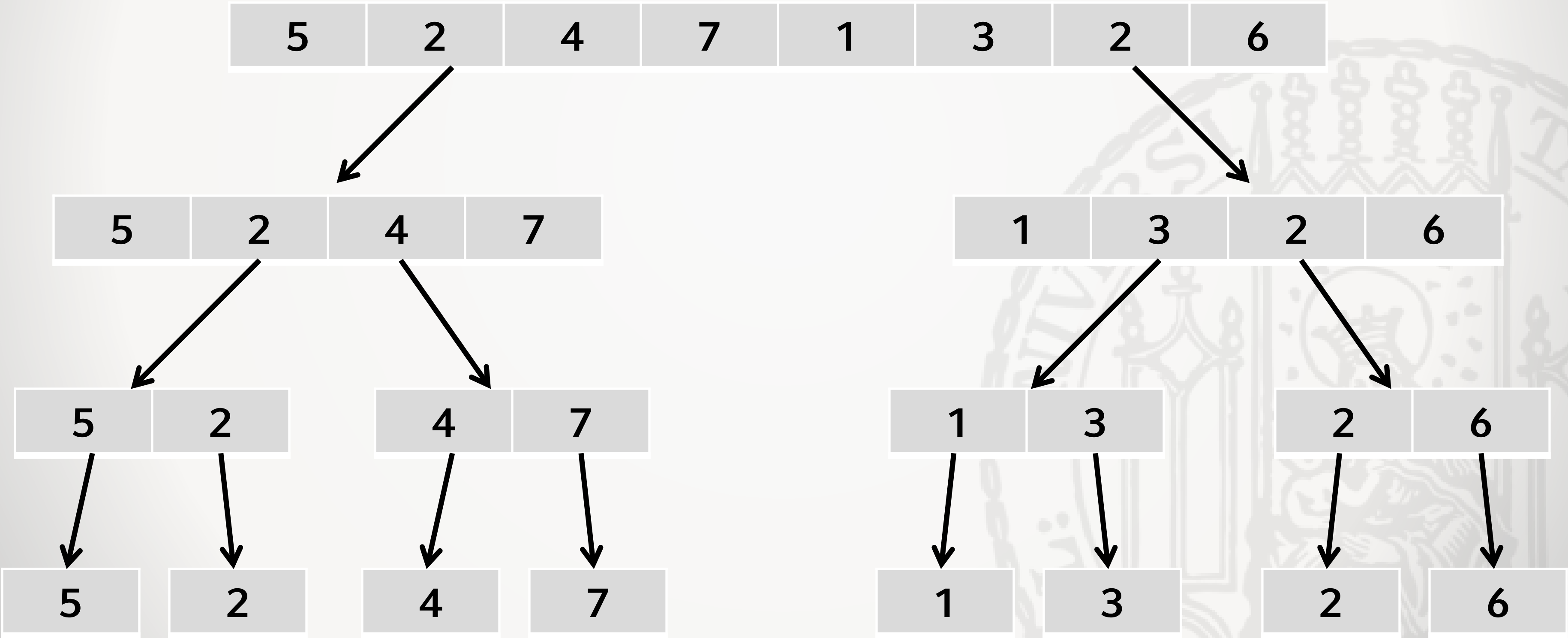
Combine: Die zwei sortierten Teillisten werden zusammengefügt (merge) um eine sortierte Liste zu erhalten.

Weitere Beispiele:

- Binary Search
- Quicksort

Merge-Sort: Divide

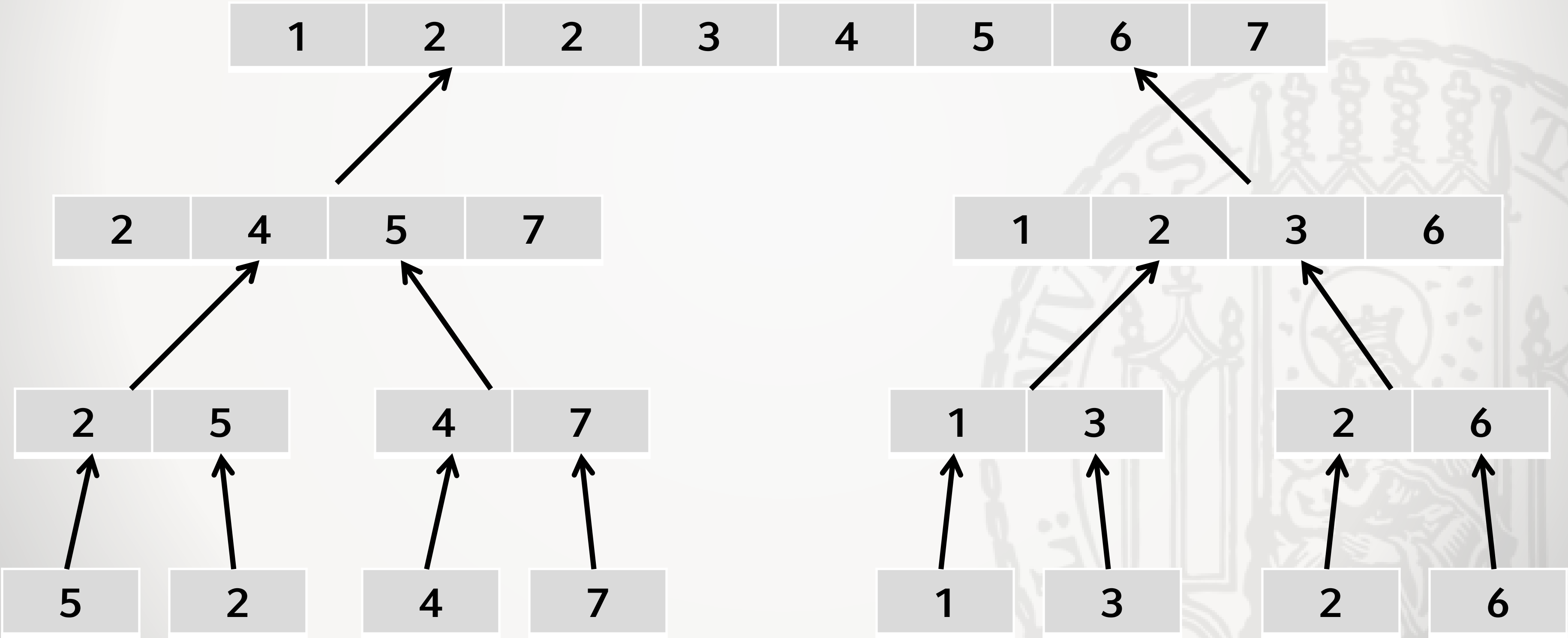
Die zu sortierende Liste der Länge n wird in zwei zu sortierende Listen der Länge $n/2$ aufgeteilt



Merge-Sort: Conquer und Merge

Einelementige Listen sind immer sortiert.

Längere Listen werden durch „mergen“ der sortierten Teillisten sortiert.



1 2 2 3 4 5 6 7

2 4 5 7 1 2 3 6

2 5 4 7 1 3 2 6

5 2 4 7 1 3 2 6

Paradigma: Greedy (gierige) Algorithmen

Idee:

Schrittweise die jeweils für den Moment beste Entscheidung treffen.

- Meist schnelle Lösung
- Liefert oft keine optimalen Lösungen

Beispiel:

Prim-Algorithmus zur Berechnung des MST

Vorgehen:

Die besten Kanten werden sukzessive hinzugenommen.

Rucksackproblem

Gegeben:

- Ein Behälter mit definierter Größe bzw. maximalem Gewicht *G*.
- Verschiedene Objekte definierter Größe/Gewicht und definierten Wertes.
  - Objekte: 1, …, i, …, n
  - Größe/Gewicht: $g_1, \ldots, g_i, \ldots, g_n$
  - Wert: $v_1, \ldots, v_i, \ldots, v_n$

Gesucht:

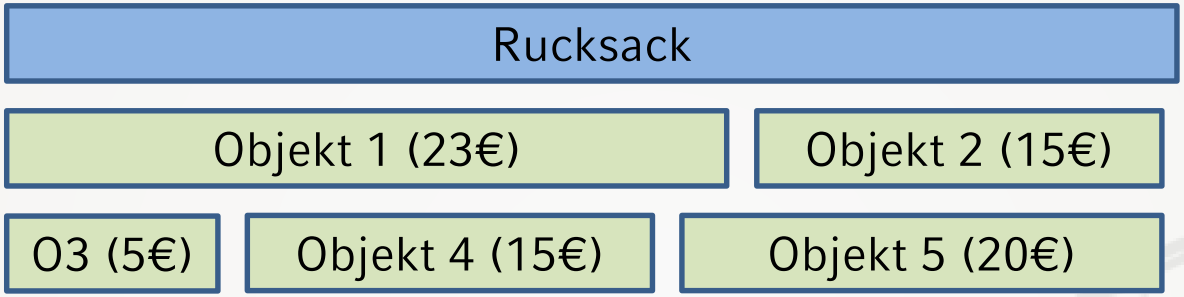
Maximaler Wert des Behälterinhaltes

Anwendung:

- LKW-Kapazität optimal nutzen
- Lagerhaltung
- …

Rucksackproblem - Beispiel

Rucksack und Objekte:



Rucksack

Objekt 1 (23€)

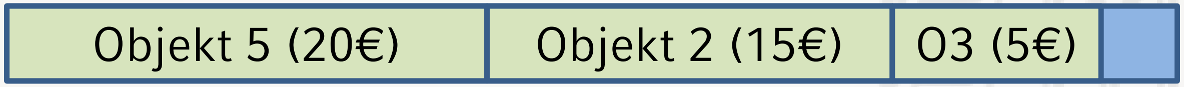
Objekt 2 (15€)

O3 (5€)

Objekt 4 (15€)

Objekt 5 (20€)

Optimale Lösung:



Objekt 5 (20€)

Objekt 2 (15€)

O3 (5€)

Greedy – Rucksackproblem

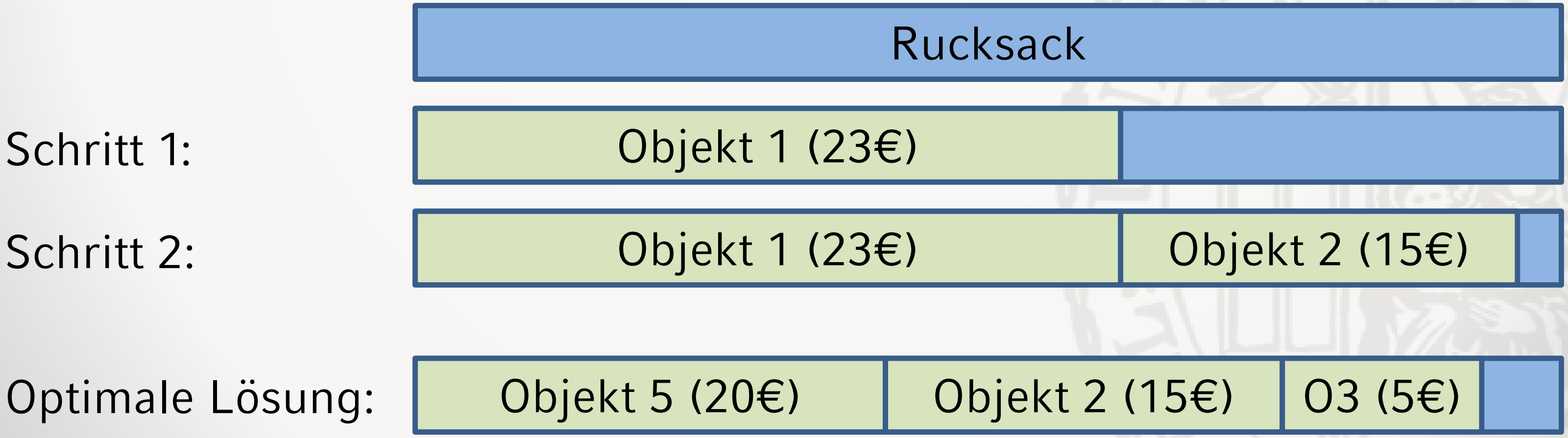
Greedy:

Wähle jeweils aus den übrigen Objekt jenes mit der höchsten Bewertung, das noch in den Rucksack passt und packe dieses hinzu.

Bewertungen:

- Je kleiner desto besser
- **Je wertvoller desto besser**
- Je größer das Verhältnis Wert zu Größe desto besser

Greedy-Ansatz führt nicht zur optimalen Lösung!



Rucksack

Schritt 1: Objekt 1 (23€)

Schritt 2: Objekt 1 (23€) | Objekt 2 (15€)

Optimale Lösung: Objekt 5 (20€) | Objekt 2 (15€) | O3 (5€)

Paradigma: Dynamische Programmierung

Idee:

Erst kleinste Probleme lösen und dann deren Lösungen sukzessive nutzen um größere Problemlösungen zu konstruieren.

Paradigma bereits bekannt:

- Floyd-Algorithmus

Jetzt:

- Edit Distanz
- Dynamischer-Algorithmus für das Rucksackproblem

Vorgehensweise
Dynamische Programmierung

1. Definition des Optimierungskriteriums
2. Definition von Teilproblemen
3. Zerlegung in Teilprobleme => führt zu Rekursionsgleichungen
4. Auswertung der Rekursionsgleichung mittels Tabelle

Rucksackproblem

1. Definition des Optimierungskriteriums:

Für jedes Objekt *i* ist zu entscheiden, ob es eingepackt wird.

g_j: Gewicht von Objekt *j*

v_j: Wert von Objekt *j*

$$a_i \in \{0,1\}$$
$$a^n := a_1, \dots, a_n$$

$$\arg\max_{a^n} \left\{ \sum_{j=1}^{n} a_j v_j \,\middle|\, \sum_{j=1}^{n} a_j v_j \leq G \right\}$$

Gesucht sind $a_1, \dots, a_n$, sodass das zulässige Gewicht nicht überschritten wird und der Gesamtwert maximal ist.

Rucksackproblem

2. Definition von Teilproblemen und einer Hilfsgröße:

Betrachtet wird der Wert *w(i,h)* eines nur bis *h* gefüllten Rucksackes, wobei nur die Objekte *1,…, i* verwendet werden.

Hilfsgröße:

$$w(i,h) = max_{a^i}\left\{\sum_{j=1}^{i} a_j v_j \,\middle|\, \sum_{j=1}^{i} a_j g_j \leq h\right\}$$

Rucksackproblem

3. Zerlegung in Teilprobleme

Ansatz: Konstruiere eine Lösung für i Objekte auf Basis der Lösung von i-1 Objekten

$$w(i,h) = max \begin{cases} 0 + \ max_{a^{i-1}} \left\{ \sum_{j=1}^{i-1} a_j v_j \,\middle|\, \sum_{j=1}^{i-1} a_j g_j \leq h \right\}, & \text{(d.h. ohne } i\text{)} \\ v_i + max_{a^{i-1}} \left\{ \sum_{j=1}^{i-1} a_j v_j \,\middle|\, \sum_{j=1}^{i-1} a_j g_j \leq h - g_i \right\} & \text{(d.h. mit } i\text{)} \end{cases}$$

$$= max\{\, w(i-1,h),\ v_i + w(i-1,h-g_i)\}$$

$w(i-1,h-g_i)$: „höchster Wert wenn Objekt i noch nicht verwendet wurde und noch Platz für das Objekt ist.“

Rucksackproblem

4. Auswertung der Rekursionsgleichung

Randbedingungen:

$$i = 0 : w(i,h) = 0$$

$$h \leq 0 : w(i,h) = 0$$

Implementierung: 2 Schleifen

```
for i = 1, …, n do
    for h = 0, …, G do
        w[i,h] = max{ … }
```

- Lösung für *n* Objekte und Rucksack der Größe *G* steht in `w[n,G]`

Rucksackproblem in Java

```
public static int pack(int capacity, int[] value, int[] weight){

 int[][] matrix = new int[value.length+1][capacity+1];
 int count = value.length;

 for(int i = 1; i <= count; i++){
  for(int h = 1; h <= capacity; h++){

   if(weight[i] <= h)
    matrix[i][h] =
     max(matrix[i-1][h], value[i] + matrix[i-1][h-weight[i]]);
   else
    matrix[i][h] = matrix[i-1][h];

  }
 }

 return matrix[count][capacity];

}
```

Rekursionsgleichung:

$$w(i,h) = \max\{w(i-1,h), v_i + w(i-1,h-g_i)\}$$

Rucksackproblem

Rucksackgröße: 10

Objekte	1	2	3	4	5	6	7
Größe	4	4	3	1	6	2	1
Wert	5	7	3	2	10	1	2

Größe

Objekte

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	5	5	5	5	5	5	5
2	0	0	0	0	7	7	7	7	12	12	12
3	0	0	0	3	7	7	7	10	12	12	12
4	0	2	2	3	7	7	9	10	12	14	14
5	0	2	2	3	7	7	10	12	12	14	17
6	0	2	2	3	7	7	10	12	12	14	17
7	0	2	4	4	7	9	10	12	14	14	17

Beispiel: Edit-Distanz

- Ähnlichkeitssuche in Datenbanken:
  - Z.B. Suchen von ähnlichen DNA-Sequenzen
  - Benötigt die Definition eines Ähnlichkeitsmaßes

- Ein Ansatz ist die Edit-Distanz zweier Sequenzen *s* und *t*:
  - Minimale Anzahl von Operationen, die benötigt werden um s in t zu überführen
  - Operationen sind Umbenennen, Löschen oder Hinzufügen einzelner Zeichen der Sequenz

- Alignments:
  - Hilfsmittel zur Veranschaulichung der Edit-Distanz
  - Ein Alignment zweier Sequenzen ist die zeichenweise Anordnung, so dass jeder Buchstabe einem anderen der anderen Sequenz oder einer Lücke „-“ zugeordnet ist
  - Die relative Ordnung der Zeichen bleibt dabei gewahrt

Edit-Distanz: Rekursionsgleichung

- Die Berechnung der Edit-Distanz zweier Sequenzen entspricht also der Suche nach einem Alignment, so dass der Ähnlichkeitswert maximal bzw. die Distanz minimal ist

- Zu einer Sequenz s seien s_i diejenige Sequenz, die aus den ersten i Zeichen von s besteht (Präfix): $s_i = s[1] \ldots s[i]$ und $q_j = q[1] \ldots q[j]$.
- Sei $|s| = l\,, |q| = m$, also und $s = s_l$, $q = q_m$. Die Edit-Distanz $d(s,q) \coloneqq d(s_l, q_m)$ lässt sich dann rekursiv wie folgt bestimmen:

$$d(s_i, q_j) = \begin{cases} j & falls\ i = 0 \\ i & falls\ j = 0 \\ d(s_{i-1}, q_{j-1}) & falls\ s[i] = q[j] \\ 1 + \ min \begin{cases} d(s_{i-1}, q_{j-1}) \\ d(s_i, q_{j-1}) \\ d(s_{i-1}, q_j) \end{cases} & sonst \end{cases}$$

Edit-Distanz: Beobachtungen

- Naiver Ansatz: Löse das Problem mittels der Rekursionsgleichung
  - Exponentieller Zeitaufwand
  - entsteht durch Mehrfachberechnungen der einzelnen Teillösungen
- Idee:
  - Bottom-Up statt Top-Down Berechnung
- Ansatz Dynamische Programmierung:
  - Teilberechnungen in eine Matrix eintragen
  - Aus den einzelnen Teillösungen für kleinere Teilprobleme Lösungen für größere Teilprobleme (bis zum Gesamtproblem) zusammensetzen
- Platz- bzw. Zeitkomplexität:
  - Jeder Matrixeintrag kann in $O(1)$ bestimmt werden
  - Komplexität beträgt also $O(|s| \cdot |t|)$

Edit-Distanz: Algorithmus

Gegeben: Zwei Sequenzen s und t

- Erstelle $(s + 1) \times (t + 1)$-Matrix D, die die einzelnen Distanzen enthält

1. Basisfälle:
   - $i = j = 0$ ➔ $D[i][j] = D[0][0] = 0$
   - $i = 0, 0 < j \leq |t|$ ➔ $D[i][j] = D[0][j] = j$ (füge j Zeichen in 1. Zeile hinzu)
   - $0 < i \leq |s|, j = 0$ ➔ $D[i][j] = D[i][0] = i$ (füge i Zeichen in 1. Spalte hinzu)

2. Ausfüllen der restlichen Tabelle:
   - Zeilen- oder spaltenweises Ausfüllen
   - $d(s_i, t_j)$ ergibt sich aus $d(s_{i-1}, t_{j-1})$, $d(s_{i-1}, t_j)$ sowie $d(s_i, t_{j-1})$, also: $D[i][j]$ ergibt sich aus $D[i-1][j-1]$, $D[i-1][j]$ sowie $D[i][j-1]$

- Die gesuchte Edit-Distanz ist dann $d(s_{|s|}, t_{|t|})$ bzw. der letzte Matrixeintrag (rechts unten), also $D[|s|][|t|]$

Beispiel zur Edit-Distanz

Berechne die Edit-Distanz der beiden Sequenzen HALLO und AUTO:

		A	U	T	O
	0	1	2	3	4
H	1				
A	2				
L	3				
L	4				
O	5				

Beispiel zur Edit-Distanz

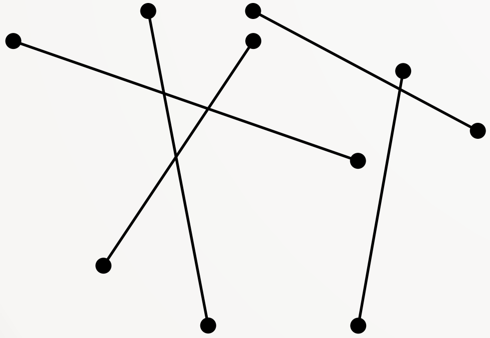
Berechne die Edit-Distanz der beiden Sequenzen HALLO und AUTO:

		A	U	T	O
	0	1	2	3	4
H	1	1	2	3	4
A	2	1	2	3	4
L	3	2	2	3	4
L	4	3	3	3	4
O	5	4	4	4	**3**

Problemstellung: Schnitt von Liniensegmenten

- Gegeben:
  Eine Menge von abgeschlossenen Liniensegmenten in der Ebene.
  (abgeschlossene Liniensegmente enthalten ihre Endpunkte)

- Gesucht:
  Schnittpunkte der Segmente

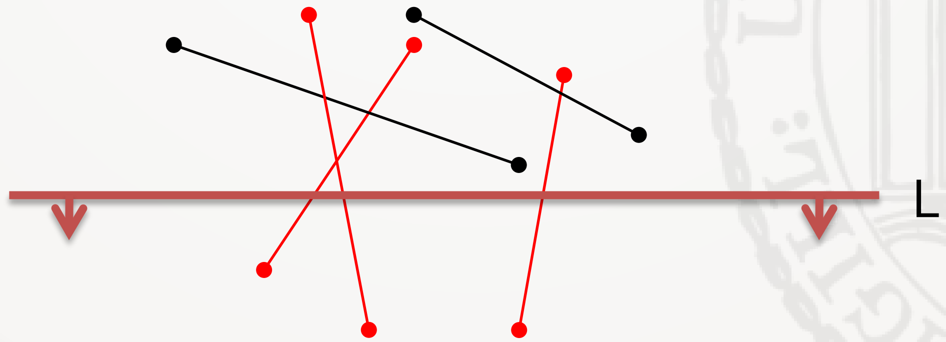


- Naive Brute-Force-Lösung:
  Testen aller Segmentpaare in $O(n^2)$
  -> Nur im Worst Case optimal.

- Gewünscht:
  Algorithmus, dessen Laufzeit geringer ist, insbesondere abhängig von der Größe des Outputs, d.h. „output-sensitiv".

Einfacher Sweep-Line-Algorithmus

- Idee: Vermeide den Test von Segmenten, die weit voneinander entfernt liegen.
- Durchführung: Teste nur die Segmentpaare, die gleichzeitig von einer horizontalen Sweep Line L geschnitten werden.
- Die Sweep Line wandert von oben nach unten und stoppt an bestimmten Events. Dieser Algorithmus wird über zwei Datenstrukturen durchgeführt:
  - Events = Endpunkte aller Segmente; verwaltet über eine Event-Liste, welche dem „Fahrplan“ des Algorithmus entspricht.
  - Status von L = Menge der von L aktuell geschnittenen Segmente. Der Status wird nach jedem Event aktualisiert und ist zu Beginn und am Ende leer.

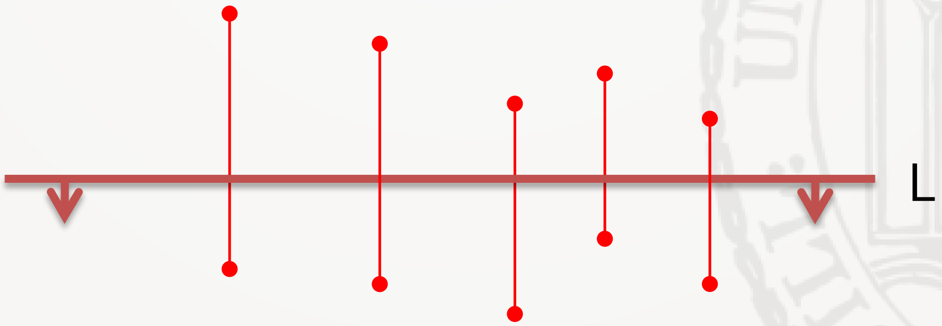


Einfacher Sweep-Line-Algorithmus

- Ist das Event der obere Endpunkt eines Segments, so wird das Segment gegen alle Segmente im Status auf Schnitt getestet und dann dem Status hinzugefügt.
- Ist das Event der untere Endpunkt eines Segments, so wird das Segment aus dem Status entfernt.

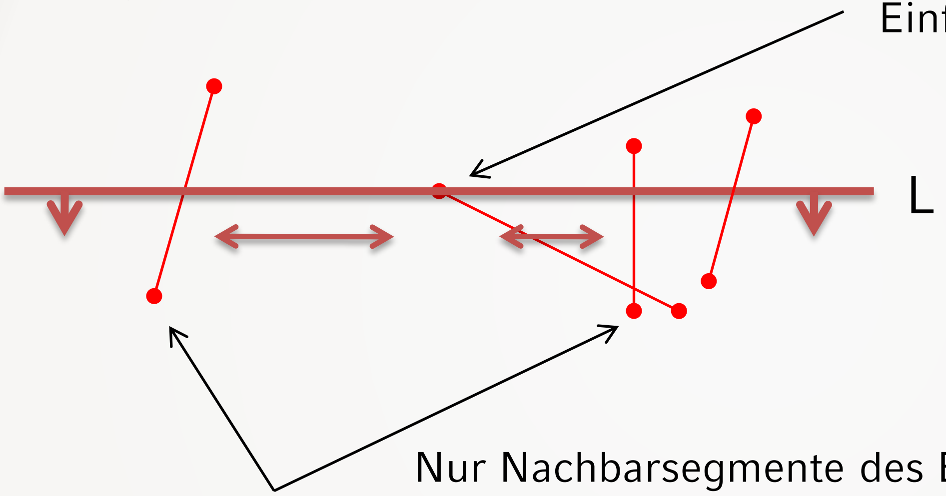
⇒ Es werden nur Paare getestet, die gleichzeitig von einer horizontalen Linie geschnitten werden.

Problem: Laufzeit immer noch unabhängig von der Größe des Outputs.



Verbesserter Sweep-Line-Algorithmus

- Ordnen der Segmente im Status von links nach rechts.
- Vorteil: Bei Events werden nur noch Tests gegen Nachbarsegmente durchgeführt.



Einfüge-Event

Nur Nachbarsegmente des Events werden betrachtet

- Zu Beachten: Bei Schnittpunkten vertauscht sich die Ordnung der beteiligten Segmente im Status.

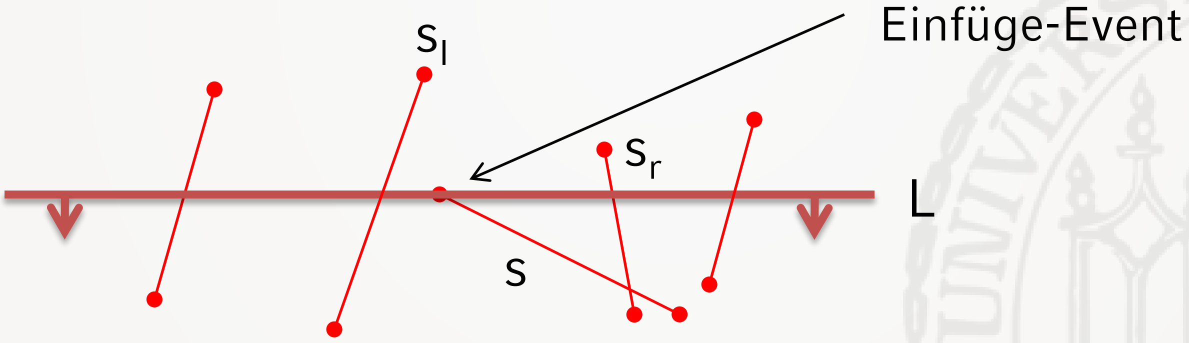
$\Rightarrow$ Schnittpunkte sind ebenfalls Events (Vertauschung), d.h. nach Erkennen eines Schnittpunktes wird dieser in die Eventliste eingefügt.

Event-Verarbeitung

- Event = Oberer Endpunkt eines Segments s

⇒ Das Segment s wird in den Status eingefügt und muss gegen seine zwei Nachbarn s_l und s_r getestet werden

⇒ Nur Schnittpunkte unterhalb von L sind von Interesse; diese werden als Events in die Queue eingefügt.

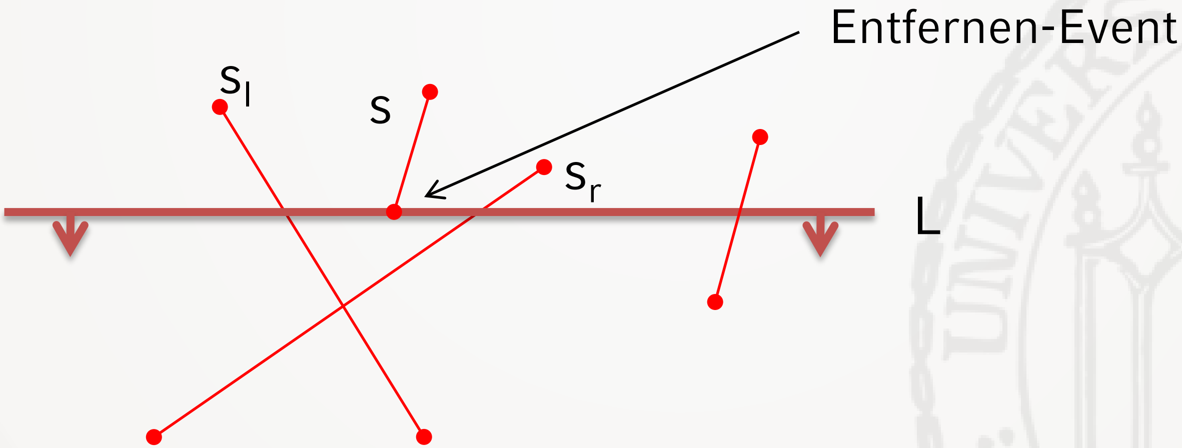


Event-Verarbeitung

- Event = Unterer Endpunkt eines Segments s

⇒ Das Segment wird aus dem Status entfernt und die Nachbarn von s werden Nachbarn und müssen auf Schnitt getestet werden

⇒ Schnittpunkte werden ggf. als Events in die Queue eingefügt.

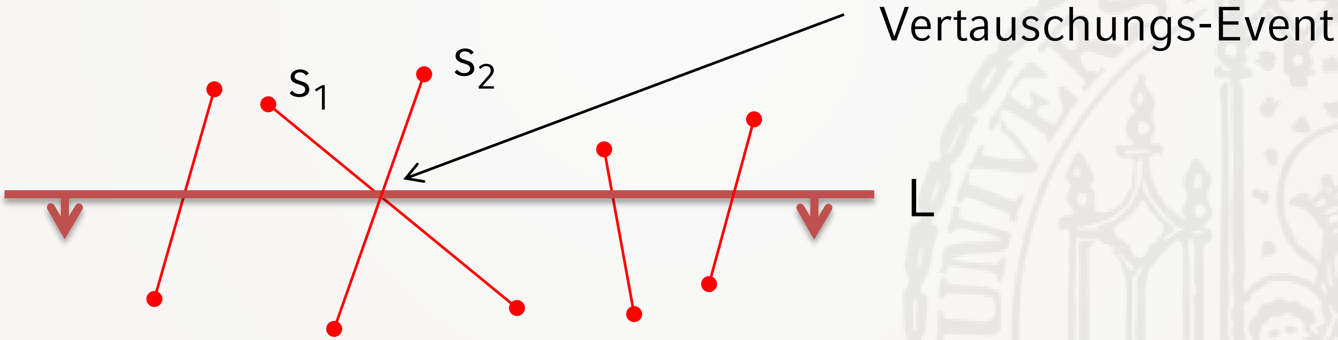


Event-Verarbeitung

- Event = Schnittpunkt zwischen Segmenten s_1 und s_2

⇒ Die Reihenfolge von s_1 und s_2 im Status wird getauscht und die Segmente jeweils gegen (höchstens) einen neuen Nachbarn getestet.

⇒ Schnittpunkte werden ggf. als Events in die Queue eingefügt.



Algorithmus: Datenstrukturen

- Event-Queue Q als balancierter binärer Suchbaum oder Heap
  - Lexikographische Ordnung:
    $p < q \Leftrightarrow (p_y < q_y$ oder $p_y = q_y$ und $p_x < q_x)$
    d.h. der linke Endpunkt eines horizontalen Segments wird zuerst abgearbeitet.
  - Operationen: Liefere nächstes Event, Prüfe ob Event bereits vorhanden

- Status T als balancierter binärer Suchbaum
  - Segmente sind entlang der Sweep Line L geordnet.
  - Operationen: Finde Segment links/auf/rechts von einem Punkt q

- Bezeichungen für disjunkte Segmentmengen:
  - U(p) = Segmente, die p als oberen Endpunkt haben.
  - L(p) = Segmente in T, die p als unteren Endpunkt haben.
  - C(p) = Segmente in T, die p im Inneren enthalten.

Algorithmus

```
FindIntersections {
    Q ← empty              // Events
    insert segment endpoints into Q
    T ← empty              // Status
    while (Q is not empty) {
            p ← GetnextEventPoint(Q)
            removeEventPoint(Q, p)
            HandleEventPoint(p)
    }
}
```

Algorithmus

```
HandleEventPoint(p) {
    determine U(p), L(p), C(p)
    if (|U(p)| + |L(p)| + |C(p)| > 1) report p as intersection
    remove L(p), C(p) from T        // Löschen & Neueinfügen von C(p)
    insert U(p), C(p) into T        //ändert die Ordnung der Segmente
    if (U(p) $\cup$ C(p) is empty)
        $s_l$, $s_r$ ← left, right neighbors of p in T
        FindNewEvents($s_l$, $s_r$, p)
    else
        s‘ ← leftmost segment of U(p) $\cup$ C(p)
        $s_l$ ← left neighbor of s‘ in T
        FindNewEvents($s_l$, s‘, p)

        s“ ← rightmost segment of U(p) $\cup$ C(p)
        $s_r$ ← left neighbor of s“ in T
         FindNewEvent(s“, $s_r$, p)
}
```

Algorithmus

```
FindNewEvents($s_l$, $s_r$, p) {
    if ($s_l$, $s_r$ intersect below L || $s_l$, $s_r$
    intersect on L and on the right of p) {
            r ← Intersection($s_l$, $s_r$)
            if (r not in Q) insert(r, Q)
    }
}
```

Komplexität: Verbesserter Sweep-Line-Alg.

- Speicherkomplexität:

  $O(n+r)$, wobei n die Anzahl der Segmente ist, und r die Anzahl ihrer Schnittpunkte. Die Anzahl der Events ist $2n+r$ ($2n$ Endpunkte und r Schnittpunkte).

- Laufzeitkomplexität:

  $O\big((n+r)\cdot\log n\big)$, wobei sich $\log n$ auf die Suchkomplexität im binären Statusbaum bezieht. Dieser enthält maximal n Segmente gleichzeitig und erfährt $n+r$ Einfügungen und $n+r$ Entfernungen.

Recap: Was haben wir gemacht?

- Was ist ein Algorithmus? Eigenschaften, Komplexitätsklassen
- Sortieren von Listen
- Suchen in Listen
- Hashing
- Baumstrukturen (Binärbäume, balancierte Bäume, Mehrwegbäume)
- Graphdarstellungen und –traversierungen
- Kürzeste Wege und minimale Spannbäume bestimmen
- Backtracking
- Divide and Conquer
- Geometrische Verfahren
- Greedy-Algorithmen
- Dynamische Programmierung
- Branch and Bound
- Online-Algorithmen