Algorithmen und Datenstrukturen

Vorlesung #04 – Backtracking und Greedy Algorithmen

Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

09 · Mai · 2023

Themen der heutigen Vorlesung

- Konzepte in der Algorithmenentwicklung
- ***Backtracking***
  - Beispiel: Permutation
  - Allgemeines Konzept, Elemente des Backtracking
  - Beispiel: Sudoku
  - Ansätze zur Verkleinerung des Suchraums
- **Greedy-Algorithmen** (*greedy algorithms*)
- Entwicklung von Greedy-Algorithmen und Beweis der Optimalität:
  - Ablaufplanung – Intervallauswahl (*Interval Scheduling*)
  - Weitere Beispiele im Anhang
- Das Rucksack Problem

Lösungsparadigmen in der Algorithmenentwicklung

Operieren auf Baum der Teillösungen:

- *Backtracking*
- *Branch-and-Bound*
- Dynamisches Programmieren
- *Greedy*-Algorithmen
- *Divide-and-Conquer* (IntroProg)

Techniken als Zusatz zu anderen Algorithmenarten:

- Heuristische Algorithmen
- Approximative Algorithmen
- Randomisierte Algorithmen

Konzepte von Algorithmen auf Baum der Teillösungen

- **Backtracking** generiert schrittweise Lösungskandidaten und versucht sie zu einer (optimalen) Lösung zu erweitern (bzw. alle Lösungen zu finden). Bei Misserfolg wird zu einem früheren Kandidaten zurückgekehrt und weitergesucht.
- **Branch-and-Bound** erkundet den Lösungsraum in einem Entscheidungsbaum, wobei nicht aussichtsreiche Zweige abgeschnitten werden.
- **Dynamische Programmierung:** Aufteilung in überlappende Teilprobleme; Speicherung von Zwischenlösungen, um größere Probleme unter Rückgriff auf gespeicherte Zwischenlösungen effizient zu lösen
- **Greedy-Algorithmen:** Schrittweise Auswahl, die auf direktem Weg zu einer optimalen Lösung führt
- **Divide-and-Conquer:** Aufteilung in unabhängige, disjunkte Teilprobleme

Lösungsparadigmen um vorhandene Algorithmen zu verbessern

- **Randomisierte Algorithmen** verwenden Zufallsentscheidungen, um bessere Laufzeiten zu erzielen.
- **Heuristische Algorithmen** verwenden eine Heuristik, um den Lösungsraum zunächst in aussichtsreichere Richtungen zu explorieren.
- **Approximative Algorithmen** nehmen vorgegebene Abweichungen von der optimalen Lösungen in Kauf, um bessere Laufzeiten zu erzielen.

Backtracking

- Bei manchen Anwendungen ist eine vollständige Durchsuchung des Lösungsraums unausweichlich.
- Systematischen Ansatz: **Backtrack-Prinzip** (*Backtracking*).
- Lösung wird schrittweise durch Probieren aller Möglichkeiten gefunden
- *Backtracking* lässt sich auch bei komplexeren Strukturen relativ gut implementieren (Speichereffizienz).
- In einigen Fällen kann die Backtracking-Suche stark abgekürzt werden (Erkennung aussichtsloser Teillösungen).

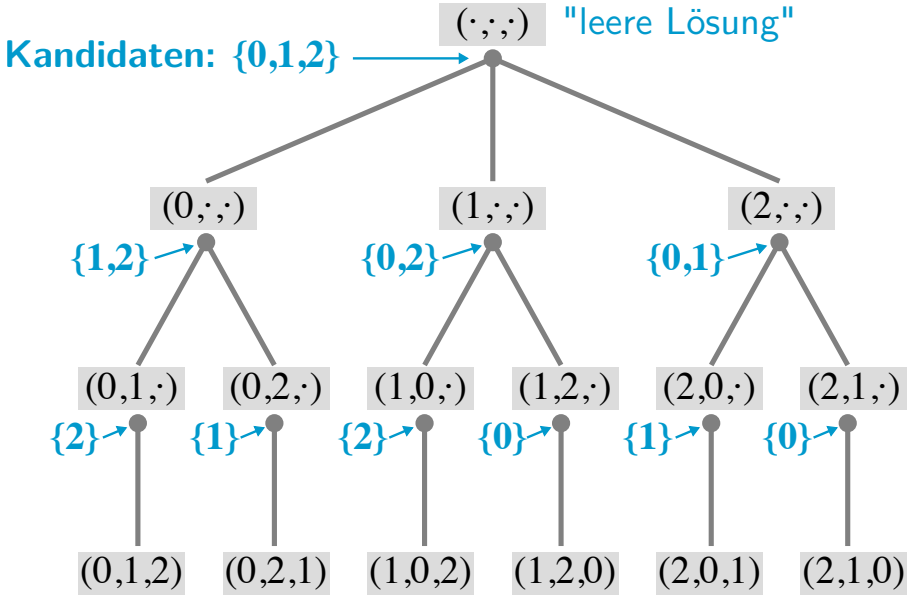
Backtracking

- Viele Algorithmen zielen auf Optimierung ab (*bester* Zug, *kürzester* Weg).
- Das Backtrack-Prinzip wird häufig bei Problemen angewendet, bei denen alle Lösungen gefunden werden sollen, z.B. Gewinnstellungen bei einem Strategiespiel,
- ... oder eine Lösung unabhängig von einem Optimierungskriterium, z.B. Lösung für ein Knobelspiel.
- Beispiel 1: Auflisten aller Permutationen der Zahlen $0, 1, \ldots, N-1$.
- Beispiel 2: Lösung von Sudoku
- Generelles Prinzip des *Backtracking*

Beispiel: Aufzählung aller Permutationen

- Aufgabe: Liste alle Permutationen der Zahlen $0, 1, \ldots, N-1$ auf.
- Vorgehen:
  - Wir fangen mit einer leeren Liste an, d.h. es wurde noch keine Zahl gewählt.
    Die Liste enthält die aktuelle Teillösung. Eine Liste der Länge N ist eine Volllösung.
  - Erste Stelle: alle Zahlen $\{0, \ldots, N-1\}$ möglich
  - Wählen eine aus und gehe zur zweiten Stelle.
  - Kandidaten: alle Zahlen, die bisher nicht gewählt wurden
  - Iteriere, bis alle Plätze besetzt sind $\rightarrow$ Lösung
  - Sobald es für eine Stelle keine neuen Kandidaten mehr gibt, geht man eine Stelle zurück (*backtracking!*) und setzt dort mit dem nächsten Kandidaten fort.

Beispiel: Aufzählung aller Permutationen



Java Implementation: Permutationen mit *Backtracking*

```
public class Permutations {
  private int N;
  private int[] a;

  public Permutation(int N) {
    this.N = N;
    a = new int[N];
    backtracking(0);
  }

  public void backtracking(int k) {
    if (k == N) printSolution();
    else {
      Iterable<Integer> candidates = candidateList(k);
      for (Integer c : candidates) {
        a[k] = c;
        backtracking(k+1);
      }
    }
  }
```

Java Implementation: Permutationen mit *Backtracking* (2)

```
  public Iterable<Integer> candidateList(int k) {
    LinkedList<Integer> c = new LinkedList<>();
    boolean[] used = new boolean[N];

    for (int i = 0; i < k; i++) {
      used[a[i]] = true;
    }
    for (int i = 0; i < N; i++) {
      if (!used[i])
        c.add(i);
    }
    return c;
  }

  public void printSolution() {
    for (int i : a)
      System.out.print(i + " ");
    System.out.println();
  }
}
```

Zwischenresümee für *Backtracking* aus Permutationsbeispiel

Positiv zu vermerken:

- Der Zustand (aktuelle Teillösung `a`) braucht nur einmal gespeichert zu werden.

Zur Erzeugung der Permutationen haben wir folgende Elemente benutzt:

- Test, ob Lösung gefunden wurde
- Menge der Kandidaten für Auswahlschritt erzeugen
- Schleife über alle diese Kandidaten
- Auswahl durchführen
- rekursiver Aufruf

Generelles Backtrack-Prinzip

zum Generieren aller Lösungen für ein Problem:

```
1 Starte mit leerer Teillösung
2 Rekursion:
3   Falls vollständige Lösung vorliegt: ausgeben bzw. speichern; return // [1]
4   Generiere Kandidaten für nächsten Schritt in aktueller Teillösung // [2]
5   Für jeden Kandidaten:
6     Führe Schritt aus // [3]
7     Gehe in Rekursion für den nächsten Schritt
8     Mache Schritt rückgängig // [4]
```

Elemente des *Backtracking*:

1. Erkennen, ob eine Teillösung eine vollständige Lösung ist
2. Menge an möglichen Lösungskandidaten für eine Entscheidung generieren
3. Lösungsschritt ausführen
4. Lösungsschritt rückgängig machen

Bemerkungen zum allgemeinen Backtrack-Prinzip

- *Schritt rückgängig machen* war im Permutationen Beispiel überflüssig
- *Prüfen auf Lösung* wird manchmal nach *Schritt ausführen* durchgeführt.
- Falls nur eine Lösung gesucht wird:
  - Falls eine Lösung gefunden wurde (Zeile 3), setze eine Variable oder Rückgabewert
  - Nach Rekursionsaufruf (Zeile 7), prüfe ob eine Lösung gefunden wurde: falls ja, beende Schleife/ Prozedur

Beispiel: Sudoku

				5			8	
		6				1		
	8				3			
2	3							
			8		1	2		
8		1			4			
3		5		7				2
							4	
9			3		5			8

- In die freien Kästchen werden die Ziffern 1–9 eingetragen.
- Dabei muss jede Ziffer genau einmal auftreten
  - in jeder Zeile,
  - in jeder Spalte und
  - in jeder umrandeten 3×3 Submatrix.

Methoden des *Backtracking* zum Sudoku Lösen

Wir legen die *Backtracking* Elemente fest:

1. Teillösung vollständig? Wenn alle Felder gefüllt sind
2. Lösungskandidaten: Alle Ziffern, die nach den Regeln passen
3. Schritt ausführen: Ziffer in ein Kästchen speichern
4. Schritt rückgängig machen: Ziffer aus dem Kästchen löschen

- Mit diesen Methoden des generellen Backtrack-Prinzip lässt sich leicht leicht ein Programm zum Lösen von Sudoku Rätsel implementieren.
- Um gute Effizienz, insbesondere bei schwierigen Sudokus zu erzielen, werden Zusatzüberlegungen benötigt.

Java Snippet: Sudoku mit *Backtracking*

```
public class Board {
    private int[][] field;
    protected Queue<CandidateList> candidates = new LinkedList<>();

    public void fillCell(Position cell, int value) {
        field[cell.y][cell.x] = value;
        populateCandidates();
    }

    public void freeCell(Position cell) {
        field[cell.y][cell.x] = FREE;
        populateCandidates();
    }

    /**
     * Populate the list of {@code candidates} for all empty cells with
     * all possible values that would fit into that cell.
     */
    public void populateCandidates() {    // ...
    }
}
```

Java Snippet: Sudoku mit *Backtracking* (2)

```
public boolean backtracking() {
    if (board.candidates.isEmpty()) {                // [1] Lösung?
        return true;
    }
    CandidateList candy = board.candidates.poll();   // [2] Kandidaten
    for (int value : candy.possibleValues()) {
        board.fillCell(candy.cell(), value);         // [3] Zug ausführen
        if (backtracking())
            return true;
        board.freeCell(candy.cell());                // [4] Zugrücknahme
    }
    return false;
}
```

Verbesserung der Laufzeit

- *Basic*: Backtracking füllt die Kästchen der Reihe nach mit jeweils passenden Zahlen (`candidates` als Queue).
- Intelligente Kästchenwahl: Füllt zunächst Kästchen mit den wenigsten Kandidaten (`candidates` als PriorityQueue).
- Insbesondere: Abbruch falls es für ein Kästchen keine passende Zahl gibt.
- Dynamische Kandidatenlisten: Aktualisiert Kandidatenlisten nach jedem Zug / jeder Zugrücknahme (möglich mit `candidates` als IndexPQ).

Laufzeit Vergleich der Sudoku Methoden

- Laufzeiten zum Lösen von drei Sudoku Rätseln verschiedener Schwierigkeitsstufen
  - leicht: erzeugt auf Stufe “mäßig” mit `https://www.surfpoeten.de/apps/sudoku/generator/`
  - mittel: erzeugt auf Stufe “extrem schwer” wie oben
  - schwer: aus [Skiena 2008].

							1	2
				3	5			
			6				7	
7						3		
			4			8		
1								
			1	2				
	8						4	
	5					6		

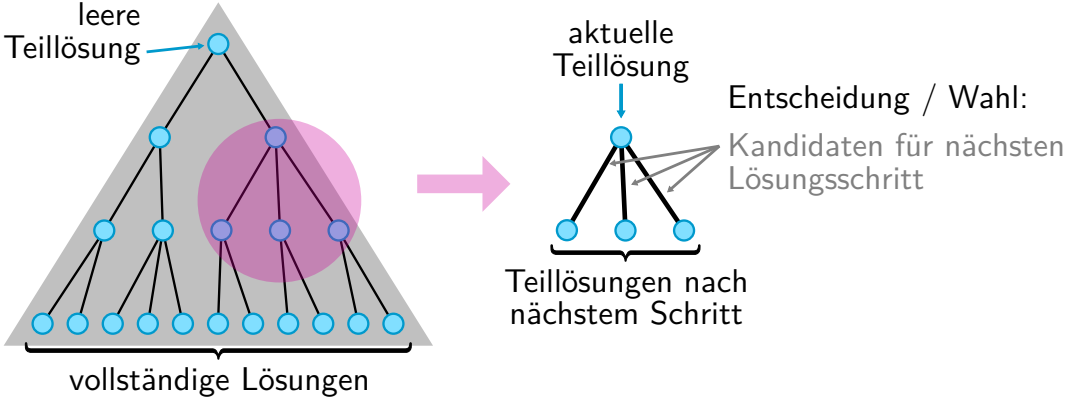
Backtracking Variante	Sudoku Schwierigkeitsgrad		
	leicht	mittel	schwer
Basic	100 ms	800 ms	104.100 ms
Intelligente Kästchenwahl	12 ms	100 ms	300 ms
Dynamische Kandidatenlisten	0,2 ms	1,5 ms	1,5 ms

Backtracking als Suche im Baum der Teillösungen

Baum der Teillösungen

- Voraussetzung: Lösungen können schrittweise entwickelt werden.
- Mögliche Lösungswege als Baum:
- An der Wurzel steht die leere Teillösung.
- Knoten: Auswahl für einen nächsten Schritt
- Jede Auswahlmöglichkeit entspricht einer Kante.
- Dies ergibt einen Baum, wenn es zu keinen Zyklen kommen kann.

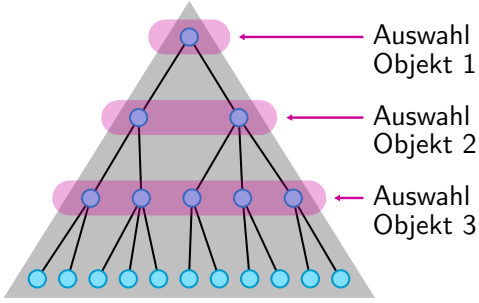
Baum der Teillösungen



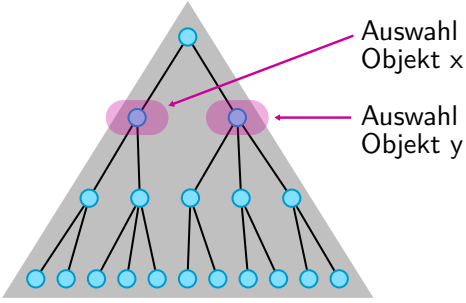
Backtracking+: Statische vs. dynamische Baumstruktur

- statisch: Es steht vorher fest, welcher Knoten welche Entscheidung repräsentiert. (Beispiele Permutationen und Rösselsprung)
- dynamisch: Während der Suche wird für jeden Knoten bestimmt, welche Entscheidung dort getroffen wird. ('Intelligente Kästchenwahl' im Sudoku Beispiel)

Statischer Suchbaum

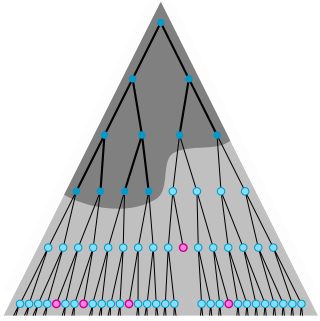


Dynamischer Suchbaum



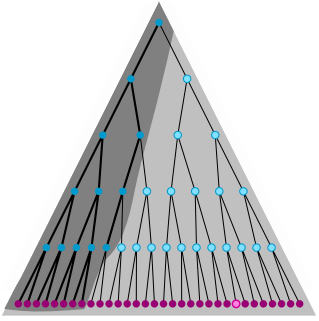
Backtracking and beyond: Reihenfolge des Expandierens

Breitensuche



Finde Schachmatt mit geringster Zuganzahl

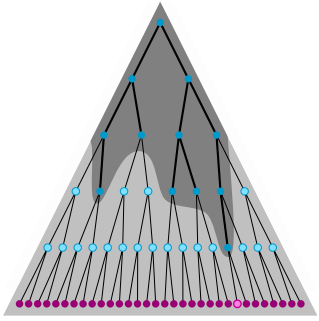
Tiefensuche



Sudoko: wähle nächstes freie Kästchen

Dynamische Suche

(informierte Reihenfolge)



Sudoko: wähle nächstes Kästchen intelligent aus

Ausblick für Verbesserungen

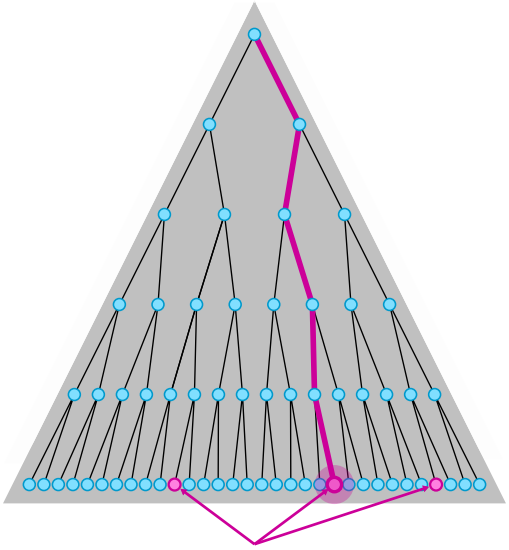
- Lösungsraum durch Vorüberlegungen beschränken, z. B. durch Ausnutzen von ‘Symmetrien’.
- Aussichtsreiche Kandidaten bevorzugt weiterverfolgen, damit früher eine Lösung gefunden wird: *Heuristische Algorithmen*
- Äquivalente Situationen (Teillösungen) erkennen und Doppelberechnungen vermeiden
  - strukturell: führt zu *Dynamischer Programmierung*
  - on-the-fly: mit *Hashtabellen*

Ausblick für Verbesserungen bei Optimierungsproblemen

- Kandidaten weglassen, die zu keiner *optimalen* Lösung führen können
- Wie kann man wissen, dass eine Teillösung nicht zu einer optimalen Lösung fortgesetzt werden kann?
- Schranken für bestenfalls erreichbare Lösungswerte bestimmen!
- Diese Grundidee führt zum *Branch-and-Bound* (nächste Vorlesung).

Motivation von Greedy-Algorithmen

- Nehmen wir an, für unser Problem gibt es nur wenige Lösungen.
- Bei Optimierungsproblemen betrachtet man oft nur Lösungen, die ein Optimalitätskriterium erfüllen, z.B. kürzeste Wege von s nach t.
- In dieser Sitation (siehe Abb.) wäre es ideal, an jedem Knoten zu wissen, bei welcher Kante man sicher zu einer Lösung kommen kann.
- Wenn man diese Auswahl an jeden Knoten trifft, gelangt man auf direktem Weg von der Wurzel zu einer Lösung.



Generelles Prinzip von Greedy-Algorithmen

- Das Paradigma der **Greedy-Algorithmen** geht folgendermaßen vor:
  - Das Problem wird durch eine schrittweise Auswahl gelöst.
  - In jedem Schritt wird eine Auswahl getroffen, die nach einem lokalen Kriterium aussichtsreich ist, und einer optimalen Lösung nicht im Wege steht.
- immense Einschränkung des Suchraums gegenüber *Backtracking*
  (Teillösungen jenseits des eingeschlagenen Pfades werden ignoriert.)
- Es gibt nicht für jedes Problem einen Greedy-Algorithmus.
- Nicht jeder Greedy-Algorithmus führt zur optimalen Lösung (die wenigsten tun es).
- Wenn er es tut, ist er oft effizienter als andere Ansätze.
- Greedy-Algorithmen sind meist einfach zu implementieren.
- Der Beweis der Optimalität ist manchmal schwierig.

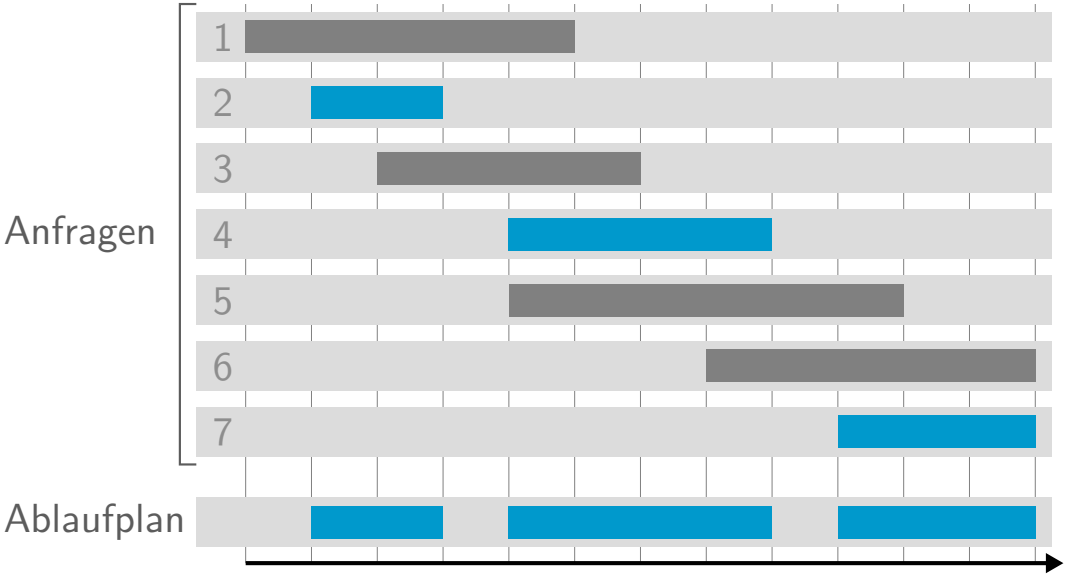
Ausblick: Greedy-Algorithmen sind wichtig

- In späteren Vorlesungen über Graphen werden wir folgenden gierigen Algorithmen begegnen:
  - Dijkstra Algorithmus für kürzeste Pfade
  - Prim Algorithmus für minimale Spannbäume
  - Kruskal Algorithmus für minimale Spannbäume
- Greedy-Algorithmen werden oft innerhalb von anderen Algorithmen verwendet:
  - in *Branch-and-Bound*, um Schranken zu bestimmen
  - in Randomisierten Algorithmen
  - in Approximationsalgorithmen als Initialisierung

Greedy Beispiel: Intervallauswahl

- Als Beispiel betrachten wir eine Variante der Ablaufplanung, die **Intervallauswahl** (*interval scheduling*)
- Es gibt Anfragen $1, \ldots, n$ zur Nutzung einer Resource (z.B. Raum, Prozessor, Messgerät) mit Anfangszeiten s_k und Endzeiten f_k entsprechend dem Zeitintervall $[s_k,\ f_k)$.
- Ziel: Belegung, die möglichst viele Anfragen erfüllt.
- Wir nennen Intervalle **kompatibel**, wenn es keine Konflikte (Überlappungen) zwischen ihnen gibt.
- Ziel: möglichst große Menge kompatibler Intervalle
- Greedy: Durchlaufe Anfragen in einer gewissen Reihenfolge und entscheide, ob Anfrage akzeptiert wird.
- Wir betrachten unterschiedliche Sortierungen und diskutieren ihre Tauglichkeit.

Intervallauswahl – Beispiel



Mögliche Kriterien für eine Greedy Auswahl bei der Intervallplanung

Wir betrachten die Anfragen in einer gewissen Sortierung (siehe Liste) und akzeptieren eine Anfrage, falls sie zu den bisher gewählten kompatibel ist.

- **[Frühester-Start]** Anfragen aufsteigend nach s_k sortiert
- **[Frühestes-Ende]** Anfragen aufsteigend nach f_k sortiert
- **[Kürzestes-Intervall]** Anfragen aufsteigend nach $f_k - s_k$ sortiert
- **[Wenigste-Konflikte]** Anfragen aufsteigend sortiert nach der Anzahl von anderen Anfragen, mit denen sie nicht kompatibel sind.

Welche Sortierungen resultieren in einem geeigneten Greedy-Algorithmus?

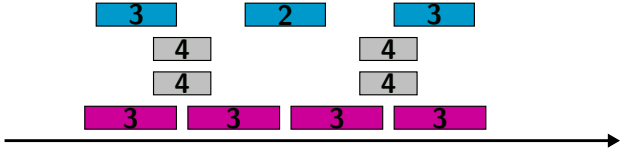
Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich nicht optimal:

- **[Frühester-Start]**

- **[Kürzestes-Intervall]**

- **[Wenigste-Konflikte]**



Notationen und Konventionen

- Wir identifizieren ein Intervall mit seinem Index k.
- Für ein Intervall (bzw Index) a schreiben wir $s(a)$ für seine Start- und $f(a)$ für seine Endzeit.
- Diese Notation erleichtert es, unterschiedliche Intervallauswahlen und -reihenfolgen gleichzeitig zu betrachten.

Greedy-Algorithmus "Frühestes-Ende" – Optimalität

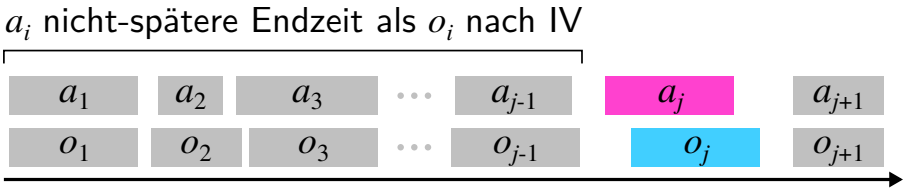
Optimalität des Greedy-Algorithmus Frühestes-Ende

Durchlaufe die n Intervalle nach der aufsteigenden Reihenfolge ihrer Endzeitpunkte und wähle jedes aus, das kompatibel mit den bisher gewählten ist.
Dieser Algorithmus liefert eine optimale Intervallplanung in einer Laufzeit in $\boldsymbol{O}(n \log n)$.

- Seien $a_1, \ldots, a_k$ die ausgewählten Intervalle in der Reihenfolge unserer Auswahl und $o_1, \ldots, o_m$ die Intervalle einer optimalen Intervallplanung, ebenfalls nach Endzeiten sortiert. Es gilt also $f(a_i) < f(a_j)$ und $f(o_i) < f(o_j)$ für $i < j$.
- Wir zeigen durch Induktion nach j, dass $f(a_j) \leq f(o_j)$ gilt (siehe nächste Seite).
- Daraus folgt $k = m$: Für $k < m$ wäre Intervall o_{k+1} kompatibel mit $a_1, \ldots, a_k$: $s(o_{k+1}) \geq f(o_k) \geq f(a_k)$. Also hätte der Algorithmus nach a_k das Intervall o_{k+1} ausgewählen können und hätte nicht nach k Intervallen gestoppt.
- Die Planung gemäß Algorithmus Frühestes-Ende beinhaltet also genauso viele Intervalle, wie die optimale Planung, ist somit auch optimal. □

Beweis der Korrektheit des Algorithmus Frühestes-Ende

- Wir zeigen durch Induktion nach j, dass $f(a_j) \leq f(o_j)$ gilt.
- Für $j = 1$ gilt dies nach unserem Auswahlkriterium 'frühste Endzeit'.
- Nach IV können wir voraussetzen, dass $f(a_i) \leq f(o_i)$ für alle $i < j$ gilt.
- Anfrage o_j ist kompatibel mit $a_1, \ldots, a_{j-1}$, denn es gilt $s(o_j) \geq f(o_{j-1}) \geq^{\text{IV}} f(a_{j-1})$. Daher wählt der Algorithmus eine Anfrage a_j, deren Endzeit nicht später als $f(o_j)$ ist.
- Es gilt also $f(a_j) \leq f(o_j)$, was zu zeigen war.



Pseudocode des Greedy Algorithmus für *Interval Scheduling*

```
1 // Gegeben: n Intervalle
2 sortiere Intervalle nach Endzeiten → I_1, ..., I_n
3 R ←∅
4 for j = 1 to n
5   if Intervall I_j ist kompatibel mit R
6     füge I_j zu R hinizu
7   end
8 end
```

- Für die Kompatibilitätsprüfung kann einfach der Endzeitpunkt des letzten Intervals von R gespeichert werden.
- Das Durchlaufen der Intervalle nach Endzeitpunkten kann über eine Vorrangwarteschlange PQ implementiert werden. Dadurch ergibt sich die Laufzeit $\boldsymbol{O}(n \log n)$.
- Für das sortierte Durchlaufen der Intervalle definieren wir eine Klasse `Interval` und eine Vergleichsoperation.

Implementation des Greedy Algorithmus für *Interval Scheduling*

```java
public class Interval {
  private double s;
  private double f;

  public Interval(double s, double f) {
    this.s = s;
    this.f = f;
  }

  public double start()  { return s; }
  public double finish() { return f; }
}
```

- Zunächst Implementieren wir eine Klasse `Interval`, die Anfangs- und Endzeit speichern und zurückgeben kann.
- Dann definieren wir eine Methode, die zwei Intervalle gemäß ihrer Endzeitpunkte vergleicht, als `Comparator` Schnittstelle.

```java
import java.util.Comparator;

public class IntervalCompareFinishTime implements Comparator<Interval> {
  @Override public int compare(Interval i1, Interval i2)
  {
    return Double.compare(i1.finish(), i2.finish());
  }
}
```

Implementation des Greedy Algorithmus für *Interval Scheduling*

```
public abstract class IntervalScheduling {

  public static Iterable<Interval> intervalScheduling(Iterable<Interval> intervals) {
    PriorityQueue<Interval> pq = new PriorityQueue<>(new IntervalCompareFinishTime());
    for (Interval i : intervals)
      pq.add(i);

    Queue<Interval> queue = new LinkedList<>();
    double finishTime = 0;     // Endzeit des letzten der ausgewählten Intervalle
    while (!pq.isEmpty()) {
      Interval iv = pq.poll();           // Intervall mit Frühestem Ende
      if (iv.start() >= finishTime) {    // Falls kompatibel
        queue.add(iv);                   // wähle es aus und
        finishTime = iv.finish();        // aktualisiere Endzeit
      }
    }
    return queue;
  }
}
```

Strategien für Optimalitätsbeweise von gierigen Algorithmen

- Der vorige Optimalitätsbeweis folgt der Strategie "der gierige Algorithmus fällt nie zurück" gegenüber anderen, insbesondere optimalen Verfahren. Dies ist hier in dem folgenden Sinn erfüllt:
- Der Endzeitpunkt des von Frühestes-Ende in Schritt k ausgewählten Intervalls ist nie später als das Ende des Intervalls, das von irgendeinem anderen Algorithmus in Schritt k gewählt wurde.
- Bei anderen Optimierungsproblemen sucht man eine andere Interpretation für das *nicht Zurückfallen*.
- Es gibt allerdings auch Fälle, in denen diese Strategie nicht greift. Dann helfen manchmal die folgenden Ansätze (Beispiele siehe Anhang)
  - "Strukturargument": Man leitet eine Grenze für den Optimalwert her und zeigt, dass der Greedy-Algorithmus diese erreicht.
  - "Austauschargument": Man kann eine optimale Lösung durch äquivalentes Ändern der Auswahl in die Lösung des Greedy-Algorithmus umwandeln.

Rucksackproblem

Es sind n Objekte mit Gewicht w_i und Wert v_i (für $1 \leq i \leq n$) sowie ein Rucksack (*knapsack*) mit einer maximalen Kapazität W gegeben. Wähle Objekte, so dass ihr Gesamtwert maximal ist und ihr Gesamtgewicht die Kapazität nicht überschreitet.

- Formal ist das Ziel $S \subseteq \{1, \ldots, n\}$ gemäß folgender Optimierung zu wählen:

$$S \text{ maximiert } \sum_{i \in S} v_i \quad \text{unter der Bedingung } \sum_{i \in S} w_i \leq W$$

- Dieses Aufgabenstellung wird auch 0/1-Rucksackproblem genannt.
- Die Auswahl könnte auch über einen Vektor $\mathbf{a} \in \{0, 1\}^n$ erfolgen.
- Dabei zeigt $a_i = 1$ an, dass Objekt i eingepackt wird.
- Diese Formulierung generalisiert zu einer anderen Variante, bei der von jedem Objekt i beliebige Anteile $a_i \in [0, 1]$ ausgewählt werden können.

Teilbares Rucksackproblem

Im Gegensatz zu dem vorigen Problem, können bei dem **teilbaren Rucksackproblem** beliebige Anteile jedes Objektes ausgewählt werden.

Teilbares Rucksackproblem

In derselben Situation wie oben sollen Anteile $a_i \in [0,1]$ (für $1 \leq i \leq n$) so gewählt werden, dass das anteilige Gesamtgewicht die Kapazität nicht überschreitet und der anteilige Gesamtwert maximal ist.

- Dies entspricht der Wahl der Koeffizienten a_i nach folgender Optimierung:

$$\mathbf{a} = (a_1, \ldots, a_n) \in [0,1]^n \text{ maximiert } \sum_{i=1}^{n} a_i v_i \quad \text{unter der Bedingung } \sum_{i=1}^{n} a_i w_i \leq W$$

Greedy-Algorithmus für das teilbare Rucksackproblem

- Möglich wären Greedy-Algorithmen Größter-Wert oder Kleinstes-Gewicht, sie sind aber nicht optimal.
- Für die Ausnutzung des Rucksacks sind Objekte geeignet mit möglichst großem relativem Wert (Wert-pro-Gewicht): $d_i = \frac{v_i}{w_i}$.
- Daher geht der Greedy-Algorithmus folgendermaßen vor:

Listing 1: **Greedy-Algorithmus für das teilbare Rucksackproblem**

```
1 sortiere Objekte absteigend nach v_i/w_i
2 r ← W          // Restkapazität
3 for i = 1 to n
4   a_i ←min(1, r / w_i)
5   r ← r − a_i w_i
6 end
```

Beweis der Optimalität

Optimalität des Greedy-Knapsack Algorithmus

Der Greedy-Knapsack Algorithmus von Listing 1 bestimmt die optimale Lösung des teilbaren Rucksackproblems.

Beweis.

- Der Lösungsvektor hat die Form $\mathbf{a} = (1, \ldots, 1, a_{k+1}, 0, \ldots, 0)$ für ein $k < n$.
- Wir zeigen, dass das Verschieben von Anteilen nach hinten die Lösung nicht verbessern kann.
- Sei $i \leq k$ und $j > k$. Wir reduzieren a_i von 1 auf $\alpha < 1$ und erhöhen a_j entsprechend. Durch die Reduktion von Objekt i sinkt das Gewicht um $(1 - \alpha)w_i$.
- Also kann von Objekt j dieses Gewicht zusätzlich genommen werden, entsprechend einer Erhöhung des Koeffizienten um $\beta = (1 - \alpha)w_i/w_j$.

(Fortsetzung nächste Seite)

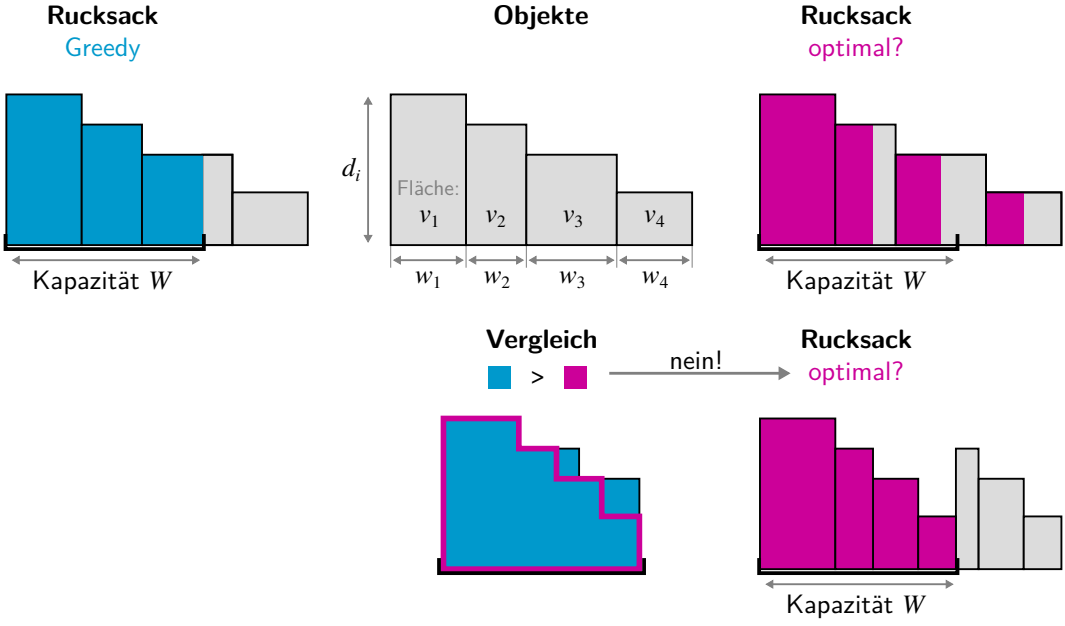
Fortsetzung des Beweises.

- Der Gesamtwert ändert sich durch die Verschiebung um

$$\begin{aligned}\beta v_j - (1-\alpha)v_i &= (1-\alpha)\frac{w_i}{w_j}v_j - (1-\alpha)v_i \\ &= (1-\alpha)\left(\frac{w_i v_j}{w_j} - v_i\right) \\ &= \frac{1-\alpha}{w_j}\left(w_i v_j - w_j v_i\right)\end{aligned}$$

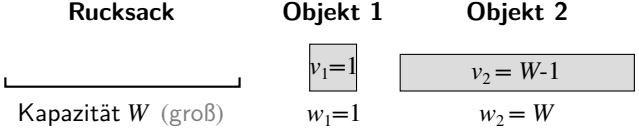
- Diese Änderung ist wegen $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$ (Sortierung nach relativem Wert) nicht positiv.
- Die Lösung kann durch Verschieben von Anteilen nach hinten also nicht verbessert werden. Für $i = k+1$ geht die Beweisführung analog.
- Also ist der Greedy-Algorithmus optimal. □

Illustration des Optimalitätsbeweises



Einschränkung des Greedy-Algorithmus

- Der vorgeschlagene Algorithmus findet die optimale Lösung für das teilbare Rucksackproblem, aber nicht unbedingt für das 0/1-Rucksackproblem.
- Für das 0/1-Problem kann die Lösung sogar beliebig stark vom Optimum abweichen:
- Sei K eine große Zahl. Wir betrachten einen Rucksack mit Kapazität $W = K + 1$ sowie zwei Objekte mit $v_1 = w_1 = 1$ und $v_2 = W - 1$, $w_2 = W$.
- Dann sind die relativen Werte $d_1 = 1$ und $d_2 = \frac{W-1}{W} < 1$.
- Also wählt der Greedy-Algorithmus Objekt 1 mit einem Wert von 1, während die optimale Wahl Objekt 2 mit Wert $W - 1 = K$ ist.
- Durch Wahl von K kann ein beliebig großer Verlust (Quotient von optimaler zu Greedy-Lösung) erzeugt werden.



Weitere Anwendungen von gierigen Algorithmen

- Minimale, aufspannende Wurzelbäume (*minimum-cost arborescence*), siehe z.B. [Kleinberg & Tardos, S. 219ff]
- Clustering, siehe z.B. [Kleinberg & Tardos, S. 199ff]
- Optimales Caching, siehe z.B. [Kleinberg & Tardos, S. 173ff]
- Huffman Code und Komprimierung, siehe z.B. [Kleinberg & Tardos, S. 203ff]

Executive Summary

- ***Backtracking*** expandiert den Baum der Teillösungen nach Art der Tiefensuche. So wird der Lösungsraum strukturiert durchsucht.

- Manchmal kann der Suchraum eingegrenzt werden, z.B. durch Ausnutzung von Symmetrien und durch Identifizieren von Teillösungen, die zu keiner Lösung führen können, also nicht weiter verfolgt werden müssen.

- In besonders glücklichen Fällen gibt es eine lokale Regel, die stets zu einer optimalen Lösung führt: **Greedy Algorithmus**

- Für das teilbare Rucksack-Problem gibt es einen optimalen Greedy-Algorithmus. Für den 0/1 Rucksack kann seine Lösung beliebig schlecht sein.

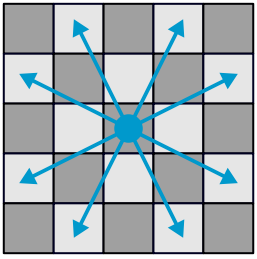
Anhang

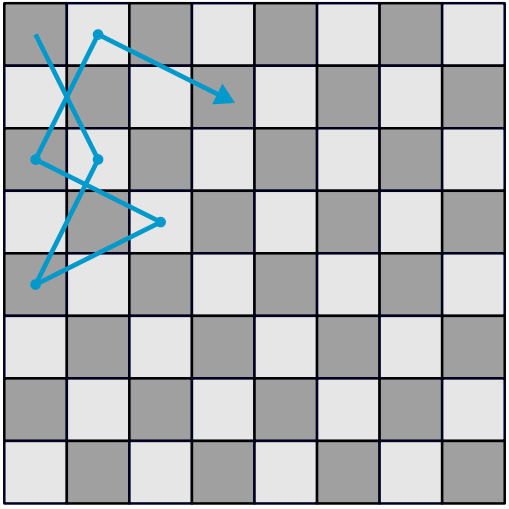
Inhalt des Anhangs: (ergänzendes Material, nicht prüfungsrelevant)

- Ein weiteres Beispiel für Backtracking:
  - Rösselsprung
- Weitere Greedy-Algorithmen und Optimalitätsbeweise
  - Ablaufplanung – Intervallverteilung (*Interval Partitioning*): S. **??**
  - Ablaufplanung mit minimaler Verspätung (*Schedule to Minimize Lateness*): S. **??**

Backtracking Beispiel: Rösselsprung

- Gegeben: Schachbrett mit einem Springer
- Finde eine geschlossene Zugfolge, bei der jedes Feld genau einmal besucht wird.
- Die möglichen Züge eines Springers sind:





Backtracking Ansatz für den Rösselsprung

- Wir brauchen zunächst Strukturen, um Brett und Spielzüge zu realisieren.
- Diese implementieren wir in den Klassen `Postition` (Position auf Spielbrett), `Move` (Spielzug) und `Board` (Spielbrett).
- `Board` muss insbesondere Methoden für die Elemente des *Backtracking* bereitstellen:
  - `validMoves()` mögliche Züge generieren
  - `isSolved()` ist Lösung erreicht?
  - `doMove(move)` Zug ausführen
  - `undoMove(move)` Zug zurücknehmen
- Damit kann die rekursive *Backtracking* Methode leicht implementiert werden.

Java Implementation: Rösselsprung mit *Backtracking*

```
public class Position {
  protected int x, y;

  public Position(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Position(Position that) {
    this(that.x, that.y);
  }

  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass()
        != o.getClass()) return false;
    Position that = (Position) o;
    return this.x == that.x
        && this.y == that.y;
  }
}
```

```
public class Move {
  protected Position pos;
  private int dir;
  public static final int nDirs = 8;
  private static final int[] xd =
      {1, 2, 2, 1, -1, -2, -2, -1};
  private static final int[] yd =
      {-2, -1, 1, 2, 2, 1, -1, -2};

  public Move(Position pos, int dir) {
    this.pos = new Position(pos);
    this.dir = dir;
  }

  public Position endPosition() {
    int x = pos.x + xd[dir];
    int y = pos.y + yd[dir];
    return new Position(x, y);
  }

}
```

Java Implementation: Rösselsprung mit *Backtracking*

```
public class Board {
  public static final int OFF = -1;
  private Position startPos;
  private int nx;
  private int ny;
  private boolean[][] field;
  private Position knightPos;
  private int nFree;

  public Board(int nx, int ny) {
    this.startPos = new Position(0, 0);
    this.nx = nx;
    this.ny = ny;
    this.field = new boolean[nx][ny];
    this.knightPos = startPos;
    this.nFree = nx * ny - 1;
  }

  // to be continued
```

```
  public int getField(Position pos) {
    if (pos.x < 0 || pos.x >= nx ||
        pos.y < 0 || pos.y >= ny) {
      return OFF;
    } else {
      return field[pos.x][pos.y] ? 1: 0;
    }
  }

  // to be continued
```

Rösselsprung: *Backtracking* Elemente

```
  public Stack<Move> validMoves() {
    Stack<Move> moves = new Stack<>();
    for (int dir = 0; dir < Move.nDirs; dir++) {
      Move move = new Move(knightPos, dir);
      if (getField(move.endPosition()) == 0) {
        moves.push(move);
      }
    }
    return moves;
  }
  public boolean isSolved() {
    return nFree == 0 && knightPos.equals(startPos);
  }
  public void doMove(Move move) {
    knightPos = move.endPosition();
    field[knightPos.x][knightPos.y] = true;
    nFree--;
  }
  public void undoMove(Move move) {
    field[knightPos.x][knightPos.y] = false;
    knightPos = move.pos;
    nFree++;
  }
}
```

Rösselsprung: *Backtracking*

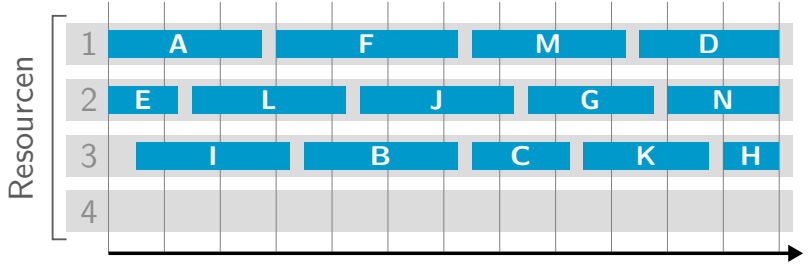
```
public class ClosedKnightsTour {

  public static boolean backtracking(Board board) {
    Stack<Move> moves = board.validMoves();
    for (Move move : moves) {
      board.doMove(move);
      if (board.isSolved()) {
        return true;
      }
      boolean solutionFound = backtracking(board);
      if (solutionFound) return true;
      board.undoMove(move);
    }
    return false;
  }

  public static void main(String[] args) {
    Board board = new Board(5, 5);
    boolean solvable = backtracking(board);
    System.out.println("Lösbar? " + solvable);
  }
}
```

Bemerkungen zur *Backtracking* Lösung für den Rösselsprung

- Der gezeigte Code entscheidet nur, ob es eine Lösung gibt. Um diese auch auszugeben, bedarf es einer kleinen Ergänzung.
- Es gibt zur Laufzeit immer nur ein `Board` Objekt. Bei dem rekursiven Aufruf wird eine Referenz übergeben.
- *Backtracking* ist eine systematische Durchsuchung des Lösungsraums, die meist nicht effizient zu einzelnen Lösungen führt.
- Für den Rösselsprung können mit *Backtracking* bis Brettgrößen 5×5 Lösungen in akzeptabler Zeit gefunden werden.
- Bei größeren Spielfeldern braucht man andere Methoden, z.B.
  - Heuristiken (Regel von Warnsdorff) oder
  - *Divide-and-Conquor*.

Greedy Beispiel: Intervallverteilung

- Nun sind die Anfragen als Anforderungen zu interpretieren, die alle erfüllt werden müssen.
- Dabei können die angeforderten Intervalle auf parallele Ressourcen verteilt werden, mit dem Ziel möglichst wenige Ressourcen zu benutzen (*Interval Partitioning Problem* oder *Interval Coloring Problem*).
- Genauer: Es gibt Intervalle $1, \ldots, n$ mit Anfangszeiten s_k und Endzeiten f_k.
- Ziel: Verteile alle Intervalle auf möglichst wenige Ressourcen, wobei die Intervalle innerhalb jeder Ressource kompatibel sein müssen.



Gierige Auswahl bei der Intervallverteilung

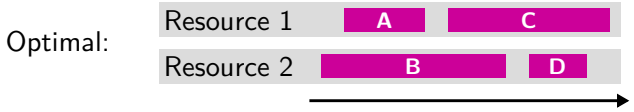
Wir betrachten die Intervalle in einer gewissen Ordnung (siehe Liste) und verteilen die jeweilige Anforderung auf eine Ressource (welche?), zu deren bisherigen Intervalle sie kompatibel ist.

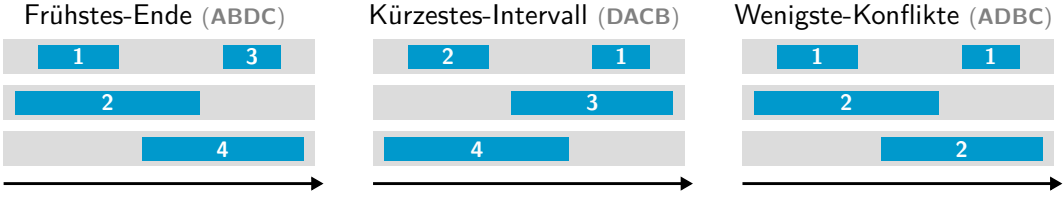
- **[Frühester-Start]** Intervalle aufsteigend nach s_k sortiert
- **[Frühestes-Ende]** Intervalle aufsteigend nach f_k sortiert
- **[Kürzestes-Intervall]** Intervalle aufsteigend nach $f_k - s_k$ sortiert
- **[Wenigste-Konflikte]** Intervalle aufsteigend sortiert nach der Anzahl von anderen Intervalle, mit denen sie in Konflikt stehen.

Welche Sortierungen resultieren in einem geeigneten gierigen Algorithmus?

Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich nicht optimal:





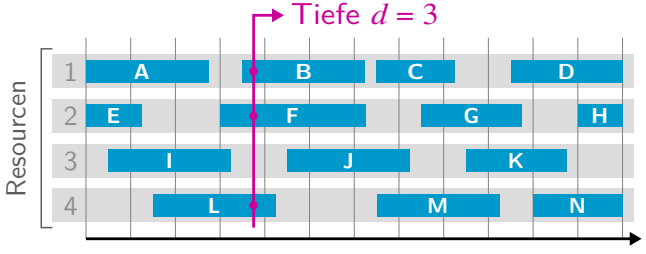
Die drei Greedy-Algorithmen benötigen 3 an Stelle der optimalen 2 Ressourcen.

Gierige Strategie “Frühester-Start”

Intervallverteilung durch Frühesten Startzeitpunkt

Durchlaufe die Intervalle $I_1, \ldots, I_n$ nach der aufsteigenden Reihenfolge ihrer Startzeitpunkte und weise jedes einer Ressource zu, in der das Intervall keinen Konflikt verursacht. Nehme eine neue Ressource, falls das Intervall in keine der existierenden Ressourcen passt.

- Zu einer Menge von Intervallen $\boldsymbol{S} = \{I_1, \ldots, I_n\}$ definieren wir die **Tiefe** (*depth*) $d(\boldsymbol{S})$ als die größte Anzahl von Intervallen, die einen gemeinsamen Punkt umfassen.



Strukturelle Grenze bei der Intervallverteilung

Offensichtlich gilt die folgende Aussage:

Tiefe als untere Grenze bei der Intervallverteilung

Die Anzahl der Ressourcen, die bei einer Intervallverteilung benötigt werden, ist mindestens die Tiefe der Menge der zu verteilenden Intervalle.

- **Beweis:** Seien $I_1, \ldots, I_d$ die Intervalle, die einen gemeinsamen Punkte umfassen. Diese Intervalle müssen auf d unterschiedliche Ressourcen verteilt werden, da es andernfalls zu einem Konflikt in dem gemeinsamen Punkt kommt. □
- Erstaunlich ist hingegen, dass dies auch die obere Grenze ist.
- Es kann also immer eine Intervallverteilung gefunden werden, deren Anzahl an Ressourcen der Tiefe der vorgegebenen Intervallmenge entspricht.
- Und unser einfacher Greedy-Algorithmus Frühester-Start findet sie!

Pseudocode des Greedy-Algorithmus Frühester-Start

```
1  // Gegeben: n Intervalle mit Start- und Endzeiten s_i und f_i
2  sortiere Intervalle nach Anfangszeiten → I_1, ..., I_n
3  K = 0
4  for j = 1 to n
5     if Intervall j ist kompatibel zu einer Ressource R(k) für ein k < K
6        füge j zu R(k) hinzu
7     else
8        initialisiere neue Ressource R(K)
9        füge j zu R(K) hinzu
10       K ←K + 1
11    end
12 end
```

- Das Durchlaufen der Intervalle nach Anfangszeitpunkten wird wieder über eine Vorrangwarteschlange PQ implementiert.
- Diesmal wird die Startzeit in der Vergleichsoperation für die PQ benutzt.

Beweis der Optimalität von Frühester-Start

Optimalität des Greedy Algorithmus für *Interval Partitioning*

Der Algorithmus Frühester-Start auf Seite 63 liefert eine optimale Intervallverteilung und kann in einer Laufzeit in $\boldsymbol{O}(n \log n)$ implementiert werden.

- Die Abfrage in Zeile 5 stellt sicher, dass alle Ressourcen kompatibel sind.
- Nun zeigen wir noch, dass Frühester-Start nicht mehr als d Ressourcen benötigt, wobei d die Tiefe der gegebenen Menge von Intervallen ist.
- Wenn eine neue Ressource in Zeile 8 für j eröffnet wird, müssen die letzten Intervalle in allen vorhandenen K Ressourcen den Anfangspunkt von I_j umfassen. Wegen der Sortierung nach Anfangszeitpunkten müssen jene Intervalle früher starten, und wegen der Inkompatibilität enden sie später als s_j.
- Es gibt also $K + 1$ Intervalle, die einen gemeinsamen Punkt umfassen. Somit ist die neu eröffnete Ressource innerhalb der Optimalitätsgrenze: $K + 1 \leq d$. □
- Die Laufzeit wird an Hand einer Java Implementation untersucht.

Implementation des Greedy Algorithmus für *Interval Partitioning*

```
public class Ressource implements
                     Comparable<Ressource>
{
  private double f;
  public Queue<Interval> intervals;

  public Ressource(Interval iv)
  {
    f = iv.finish();
    intervals = new LinkedList<>();
    intervals.add(iv);
  }

  public double getFinish()        { return f;  }
  public void setFinish(double f)  { this.f= f; }

  @Override
  public int compareTo(Ressource that)
  {
    return Double.compare(this.f, that.f);
  }
}
```

- Wir implementieren eine Klasse `Ressource`, die die Intervalle einer Ressource speichert. Außerdem wird der späteste Endzeitpunkte der enthaltenen Intervalle gespeichert.
- Und diesmal definieren wir einen `Comparator`, der zwei Intervalle gemäß ihrer Startzeitpunkte vergleicht analog zu `IntervalCompareFinishTime` durch Austauschen von `finish()` durch `start()`.

Implementation des Greedy Algorithmus für *Interval Partitioning*

```
1  public static Iterable<Ressource> eariestStart(Iterable<Interval> intervals)
2  {
3    PriorityQueue<Interval> pq = new PriorityQueue<>(new IntervalCompareStartTime());
4    for (Interval iv : intervals)
5      pq.add(iv);
6
7    PriorityQueue<Ressource> ressources = new PriorityQueue<>();
8    while (!pq.isEmpty()) {
9      Interval iv = pq.poll();
10     Ressource res = ressources.peek();   // null at first iteration
11     if (res != null && iv.start() >= res.getFinish()) {
12       res.intervals.enqueue(iv);
13       ressources.poll();                 // remove res
14       res.setFinish(iv.finish());       // update finish time
15       ressources.add(res);               // and re-add
16     } else
17       ressources.add(new Ressource(iv));
18   }
19
20   return ressources;
21 }
```

Untersuchung der Laufzeit von Frühester-Start

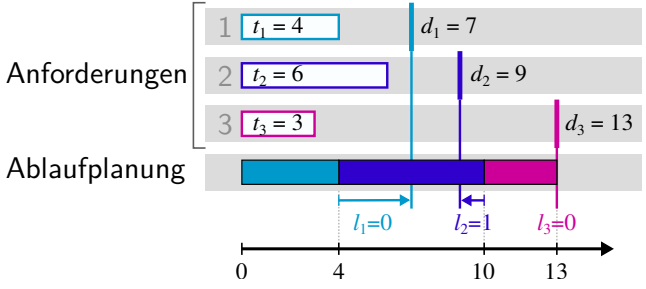
Laufzeit des Greedy Algorithmus für *Interval Partitioning*

Der Implementation von Frühester-Start auf Seite 66 zur optimalen Intervallverteilung hat eine Laufzeit in $\boldsymbol{O}(n \log n)$ für n Intervalle.

- Zur Sortierung wird jedes Interval per `add()` in eine PQ eingefügt. Für n Intervalle ergibt dies eine Laufzeit in $\boldsymbol{O}(n \log n)$.
- Die Schleife (Zeilen 8–18) wird für jedes der n Intervalle einmal durchlaufen.
- Die verwendeten PQ Methoden `poll()` und `add()` haben eine Laufzeit in $\boldsymbol{O}(\log n)$, da die PQs höchstens n Elemente beinhalten. Die Methode `peek()` hat konstante Laufzeit.
- Die Methode `add()` von `Queue` hat konstante Laufzeit.
- Damit hat diese Schleife und der gesamte Algorithmus eine Laufzeit in $\boldsymbol{O}(n \log n)$. □

Greedy Beispiel: Planung mit minimalen Verspätungen

- Bei der Planung mit minimalen Verspätungen (*Schedule to Minimize Lateness*) geht es wieder um eine Ablaufplanung mit einer einzigen Zeitschiene.
- In diesem Fall sind als Anforderungen Intervalllängen t_j und ein spätester Endzeitpunkt d_j (*deadline*) vorgegeben.
- Durch Festsetzen des Anfangszeitpunktes s_j der Anforderungen j entsteht ggf. eine Verspätung (*lateness*) $l_j = \max\{0, s_j + t_j - d_j\}$.
- **Ziel:** Finde Ablaufplanung, die die Maximalverspätung $L = \max_j l_j$ minimiert.



Gierige Auswahl bei Planung mit minimalen Verspätungen

Die Anforderungen werden in einer gewissen Ordnung (siehe Liste) ohne Leerlaufzeiten arrangiert. Diesmal sind keine Start- und Endzeiten vorgegeben, sondern Dauer t_i und Deadline d_i. Daher gibt es andere Vorschläge für die Greedy-Auswahl.

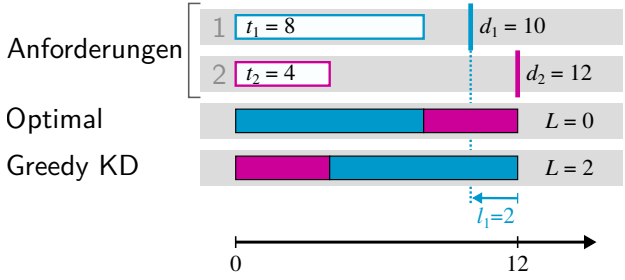
- **[Kürzeste-Dauer]** Anforderungen aufsteigend nach t_j
- **[Kleinste-Pufferzeit]** Anforderungen aufsteigend nach $d_j - t_j$
- **[Früheste-Deadline]** Anforderungen aufsteigend nach d_j

Welche Sortierungen resultieren in einem geeigneten gierigen Algorithmus?

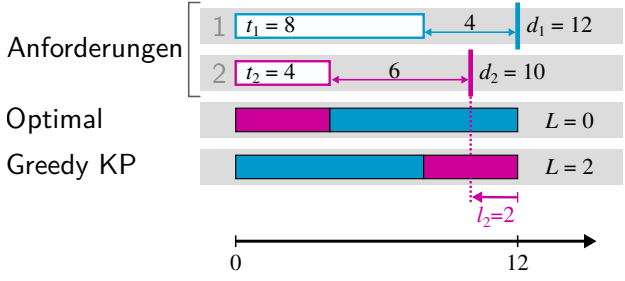
Gegenbeispiele

Zwei der vorgeschlagenen Strategien sind offensichtlich nicht optimal:

- **Kürzeste-Dauer:**



- **Kleinste-Pufferzeit:**



Optimale Strategie zur Planung mit minimaler Verspätung

Planung mit minimaler Verspätung durch Früheste-Deadline

Plane alle Anfragen in der aufsteigenden Sortierung ihrer *Deadlines*.

- Der Beweis der Optimalität dieses Greedy Algorithmus erfolgt durch ein sogenanntes **Austauschargument**:
- Wir nehmen einen beliebigen optimalen Ablauf als gegeben an. Dieser wird schrittweise in das Ergebnis des Greedy Algorithmus umgeformt, ohne dass dabei die Optimalität verloren geht.
- Bei dem Beweis spielen Vertauschungen eine wichtige Rolle.
- Wir sprechen von einer **Inversion**, wenn in einem Ablauf eine Aufgabe i vor einer Aufgabe j eingeplant ist, obwohl die Deadline von j vor der Deadline von i liegt, also $d_j < d_i$.
- Der von Früheste-Deadline erzeugte Ablauf hat offensichtlich keine Inversionen.

Optimalität der gierigen Strategie zur Planung mit minimaler Verspätung

Optimalität von Früheste-Deadline

Die gierige Strategie Früheste-Deadline ergibt einen optimalen Ablauf, d.h. das Maximum der Verspätungen ist so klein wie möglich.

- Auf den folgenden Seiten werden folgende Aussagen bewiesen:
- Es gibt einen optimalen Ablauf ohne Inversionen und Leerlauf.
- Alle Abläufe ohne Inversionen und Leerlauf haben dieselbe maximale Verspätung, also sind sie im Sinne der Optimalität gleichwertig.
- Da der Ablauf, der durch Früheste-Deadline erzeugt wird, keine Inversionen und keinen Leerlauf hat, folgt aus den beiden Aussagen die Optimalität. □

Existenzbeweis

Es gibt einen optimalen Ablauf ohne Inversionen und Leerlauf.

1. Es gibt einen optimalen Ablaufplan ohne Leerlauf.

- **Beweis.** Wir nehmen einen optimalen Ablaufplan mit Leerauf als gegeben an. Dann können alle auf den Leerlauf folgenden Intervalle entsprechend vorgezogen werden. Dadurch kann sich keine Verspätung vergrößern. Also ist der resultierende Ablaufplan ohne Leerlauf auch optimal.
- Wenn es also überhaupt einen optimalen Ablaufplan gibt, dann gibt es auch einen ohne Leerlauf. Da es nur endlich viele Ablaufpläne ohne Leerlauf gibt (Permutationen der Aufgaben), wird das Minimum der Verspätungen angenommen.
- Sei nun ein optimaler Ablauf O ohne Leerlauf gegeben. Wir zeigen in mehreren Schritten, dass Inversionen von O, sofern überhaupt vorhanden, durch Vertauschung eliminiert werden können.
- Es folgen drei weitere Aussagen und ihre Beweise, die zeigen, dass eventuell vorhandene Inversion schrittweise beseitigt werden können, ohne die Optimalität zu verlieren.

Existenzbeweis – Aussagen 2 & 3

- Sei $\boldsymbol{O}$ ein optimaler Ablauf ohne Leerlauf (aus 1) mit Inversion.

2 Es gibt Aufgaben $i < j$ mit $d_j < d_i$, die im Ablauf $\boldsymbol{O}$ direkt aufeinander folgen.

- **Beweis.** Wir betrachten eine Inversion $h < k$ mit $d_k < d_h$. Wenn man von Aufgabe h im Ablauf voranschreitet, muss es irgendwann den ersten Punkt geben, an dem die Deadline kleiner wird.
- Dieser Übergang ist das aufeinander folgende Paar $i < j$ mit $d_j < d_i$.

3 Wenn in der Situation von **2** Aufgaben i und j vertauscht werden, erhalten wir einen Ablauf mit einer Inversionen weniger.

- **Beweis.** Das Paar (i, j) war eine Inversionen im ursprünglichen Ablauf, die im neuen Ablauf nicht mehr existiert.
- Da i und j direkt aufeinander folgen, werden keine neuen Inversionen erzeugt.

Existenzbeweis – Aussage 4

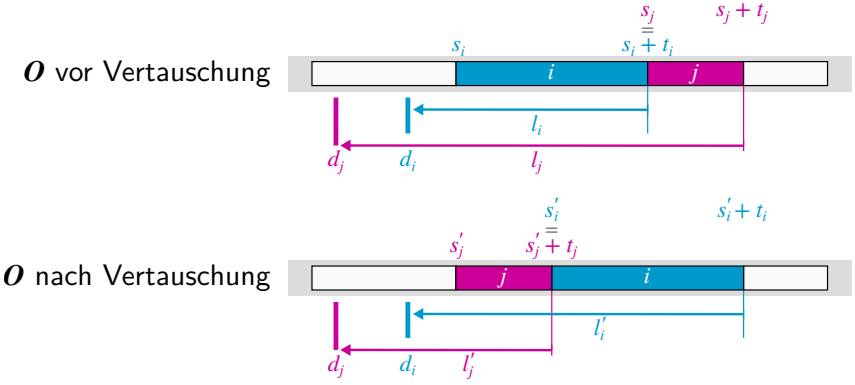
4 Die maximale Verspätung wird durch die Vertauschung nicht vergrößert.

- **Beweis.** Bei der lokalen Vertauschung der direkt aufeinander folgenden Aufgaben i und j bleiben die Verspätungen von allen anderen Aufgaben gleich.
- Wir bezeichnen Anfangszeit und Verspätung des urspünglichen Ablaufs mit s_i, s_j und l_i, l_j und die Werte nach Vertauschung mit s'_i, s'_j und l'_i und l'_j.

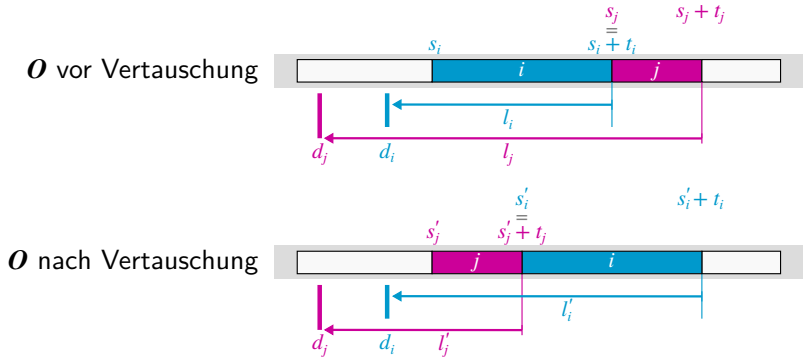
Existenzbeweis – Aussage 4

4 Die maximale Verspätung wird durch die Vertauschung nicht vergrößert.

- **Beweis.** Bei der lokalen Vertauschung der direkt aufeinander folgenden Aufgaben i und j bleiben die Verspätungen von allen anderen Aufgaben gleich.
- Wir bezeichnen Anfangszeit und Verspätung des urspünglichen Ablaufs mit s_i, s_j und l_i, l_j und die Werte nach Vertauschung mit s'_i, s'_j und l'_i und l'_j.



Existenzbeweis – Gleichungskette zu Aussage 4



$$
\begin{aligned}
l'_i &= s'_i + t_i - d_i && \text{Definition von Verspätung} \\
&= s'_j + t_j + t_i - d_i && \text{wegen } s'_i = s'_j + t_j \\
&= s_i + t_j + t_i - d_i && \text{wegen } s'_j = s_i \\
&= s_j + t_j - d_i && \text{wegen } s_j = s_i + t_i \\
&\leq s_j + t_j - d_j = l_j && \text{wegen } d_j < d_i
\end{aligned}
$$

Abschluss des Existenzbeweises

- Aus den Aussagen **1**–**4** folgt die behauptete Existenz:
- Laut **1** gibt es einen optimalen Ablauf ohne Leerlauf. Dieser kann nur endlich viele Inversionen (maximal $\binom{n}{2}$) enthalten.
- Alle direkt aufeinander folgenden Inversionen können gemäß **3** durch Vertauschung beseitigt werden.
- Dabei bleibt die optimale Verspätung nach **4** erhalten.
- Wenn es keine direkt aufeinander folgenden Inversionen mehr gibt, kann es laut **2** überhaupt keine Inversionen mehr geben.
- Wir haben also einen optimalen Ablauf ohne Inversionen und ohne Leerlauf. □

Beweis der Äquivalenz von Abläufen ohne Inversion und Leerlauf

Äquivalenzbeweis

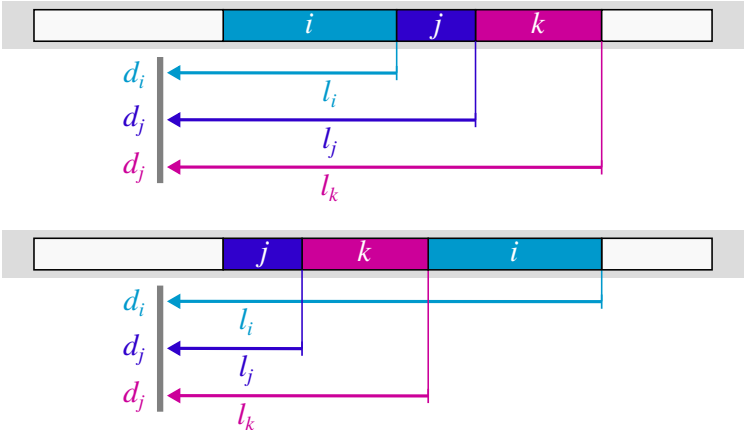
Alle Abläufe ohne Inversionen und Leerlauf haben dieselbe maximale Verspätung.

Beweis.

- Zwei Abläufe ohne Inversionen und Leerlauf können sich nur in der Reihenfolge von Intervallen unterscheiden, die dieselbe Deadline haben.
- Diese Intervalle müssen aufgrund der Inversionsfreiheit in beiden Abläufen in jeweils einem Block hintereinander liegen.
- Da die beiden Blöcke dieselben Intervalle umfassen und es keine Lücken gibt, ist die Länge der Blöcke dieselbe.
- Somit stimmen die Endzeiten der jeweils letzten Intervalle überein.
- Da die maximale Verspätung dieser Intervalle gerade durch diesen Endzeitpunkt bestimmt wird, ist sie in beiden Abläufen gleich. □

Abbildung zum Äquivalenzbeweis

- Intervalle i, j, und k bilden einen Block von Intervallen, die alle dieselbe Deadline $d_i = d_j = d_k$ haben. Die maximale Verspätung erzeugt das letzte Intervall in dem Block.
- Da das Ende des letzten Intervalles bei allen Vertauschung gleich ist, ändert sich die maximale Verspätung durch eine solche Vertauschung nicht.



Literatur I

Generell:

- Kleinberg J, Tardos E. *Algorithm Design*. Pearson Education Limited; Auflage: Pearson New International Edition (30. Juli 2013). ISBN: 978-1292023946
- Schöning U. *Algorithmik (Spektrum Lehrbuch).* Spektrum Akademischer Verlag; 2001. ISBN: 978-3827410924
- Blum N. *Algorithmen und Datenstrukturen.* Oldenbourg Wissenschaftsverlag, 1. Auflage; 2004. ISBN: 3-486-27394-9
- Skiena S. *The Algorithm Design Manual.* Springer; Auflage: 2nd ed. 2008. ISBN: 978-1848000698
- Ottmann T & Widmayer P. *Algorithmen und Datenstrukturen.* Springer Verlag, 5. Auflage; 2011. ISBN: 978-3827428042

Material zu speziellen Themen dieser Vorlesung:

- Conrad A, Hindrichs T, Morsy H, Wegener I. *Solution of the knight's Hamiltonian path problem on chessboards*. Discrete Applied Mathematics 50(2):125-34, 1994.

Literatur II

- von Warnsdorf, HC. *Des Rösselsprunges einfachste und allgemeinste Lösung*. Verhagen, 1823.

Anderes Vorlesungsmaterial:

- Wayne K. Vorlesung *Theory of Algorithms* (COS 423), Princeton University 2013. `https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures.php`
- Dietzfelbinger. *Effiziente Algorithmen*, TU Ilmenau 2012. `https://www.tu-ilmenau.de/en/institute-of-theoretical-computer-science/lehre/lehre-ss-2017/aud`
- Röglin H. *Skript zur Vorlesung Algorithmen und Berechnungskomplexität I*, Universität Bonn, `http://www.roeglin.org/teaching/Skripte/AlgoI.pdf`

Danksagung I

Bei der Darstellung der Gierigen Algorithmen habe ich viele Ideen von den großartigen Folien von Kevin Wayne zu seiner Vorlesung *Theory of Algorithms* (COS 423, Princeton University 2013) aufgenommen. (Seine Vorlesung orientiert sich seinerseits an dem Buch von Kleinberg & Tardos.)

Index