

Algorithmen und Datenstrukturen

Vorlesung #04 – Backtracking und Greedy Algorithmen



Benjamin Blankertz

Lehrstuhl für Neurotechnologie, TU Berlin

benjamin.blankertz@tu-berlin.de

08 · Mai · 2019



- ▶ Konzepte in der Algorithmenentwicklung
- ▶ *Backtracking*
 - ▶ Beispiel: Permutation
 - ▶ Beispiel: Rösselsprung
 - ▶ Allgemeines Konzept, Elemente des Backtracking
 - ▶ Ansätze zur Verkleinerung des Suchraums
 - ▶ Beispiel: Sudoku

Themen der heutigen Vorlesung

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

Operieren auf Baum der Teillösungen:

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

Techniken als Zusatz zu anderen Algorithmenarten:

- ▶ Heuristische Algorithmen
- ▶ Approximative Algorithmen
- ▶ Randomisierte Algorithmen

- ▶ **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.
- ▶ **Greedy-Algorithmen:** Schrittweise Auswahl, die auf direktem Weg zu einer optimalen Lösung führt
- ▶ **Dynamische Programmierung:** Aufteilung in überlappende Teilprobleme; Speicherung von Zwischenlösungen, um größere Probleme unter Rückgriff auf gespeicherte Zwischenlösungen effizient zu lösen
- ▶ **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.

- ▶ Bei manchen Anwendungen ist eine **vollständige Durchsuchung des Lösungsraums** unausweichlich.
- ▶ Einen systematischen Ansatz hierzu liefert das **Backtrack-Prinzip** (*Backtracking*).
- ▶ *Backtracking* lässt sich auch bei komplexeren Strukturen relativ gut implementieren (Speichereffizienz).
- ▶ In manchen Anwendungen kann bei gewissen Teillösungen ausgeschlossen werden, dass sie zu einer Lösung fortgesetzt werden können. In diesem Fall kann der Ast des Suchbaums beim Backtracking unberücksichtigt bleiben.

- ▶ Viele Algorithmen, die wir im Laufe der Vorlesung besprechen werden, 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.

- ▶ Viele Algorithmen, die wir im Laufe der Vorlesung besprechen werden, 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 $0, 1, \dots, N - 1$.
- ▶ Beispiel 2: Rösselsprung
- ▶ Im Anschluss wird das generelle Prinzip des *Backtracking* erläutert.
- ▶ Schließlich wird das Lösen von Sudoku Rätseln mit *Backtracking* skizziert.

Beispiel: Aufzählung aller Permutationen

- Aufgabe: Liste alle Permutationen der Zahlen $0, 1, \dots, N - 1$ 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.
 - ▶ Für die erste Stelle kommen alle Zahlen aus $\{0, \dots, N - 1\}$ als Kandidaten in Frage.
 - ▶ Wir wählen einen Kandidaten für die erste Stelle und widmen uns der zweiten.

Beispiel: Aufzählung aller Permutationen

- Aufgabe: Liste alle Permutationen der Zahlen $0, 1, \dots, N - 1$ 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.
 - ▶ Für die erste Stelle kommen alle Zahlen aus $\{0, \dots, N - 1\}$ als Kandidaten in Frage.
 - ▶ Wir wählen einen Kandidaten für die erste Stelle und widmen uns der zweiten.
 - ▶ Hier kommen alle Zahlen, außer denen, die schon in der Teillösung verwendet wurden, in Frage.
 - ▶ Dieser Prozess wird solange durchlaufen, bis alle Plätze besetzt, bzw. alle Zahlen ausgewählt, sind. Dann ist eine Lösung gefunden.
 - ▶ Sobald es für eine Stelle keine neuen Kandidaten mehr gibt, geht man eine Stelle zurück (*backtracking!*) und setzt dort mit dem nächsten Kandidaten fort.

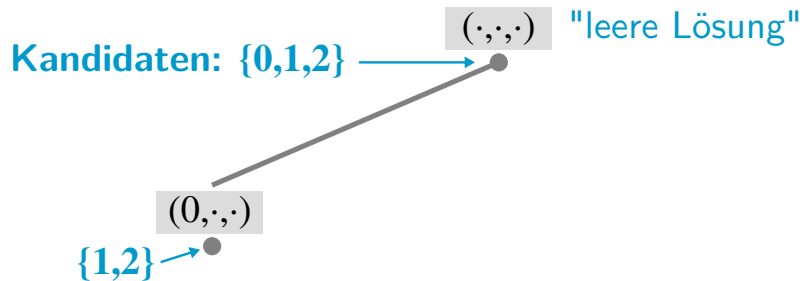
Beispiel: Aufzählung aller Permutationen

(\cdot, \cdot, \cdot) "leere Lösung"

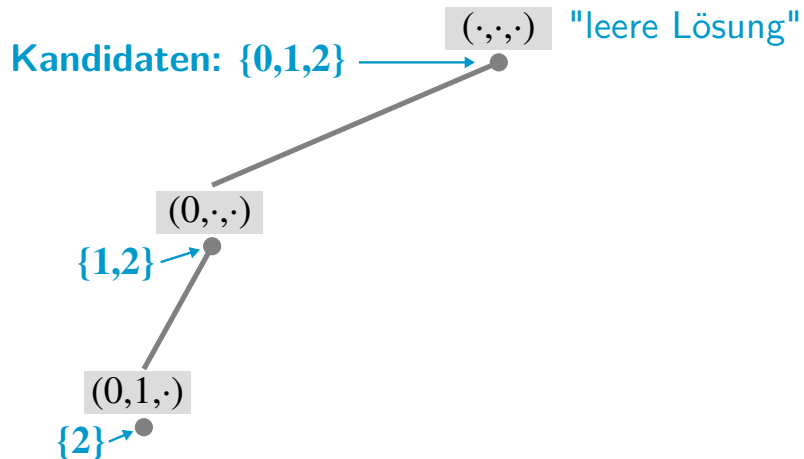
Beispiel: Aufzählung aller Permutationen

Kandidaten: {0,1,2} \longrightarrow (\cdot, \cdot, \cdot) "leere Lösung"

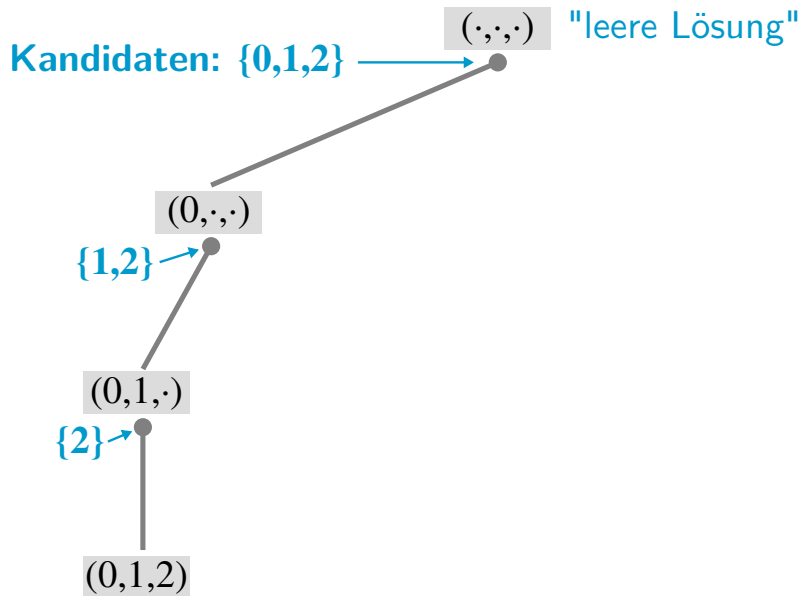
Beispiel: Aufzählung aller Permutationen



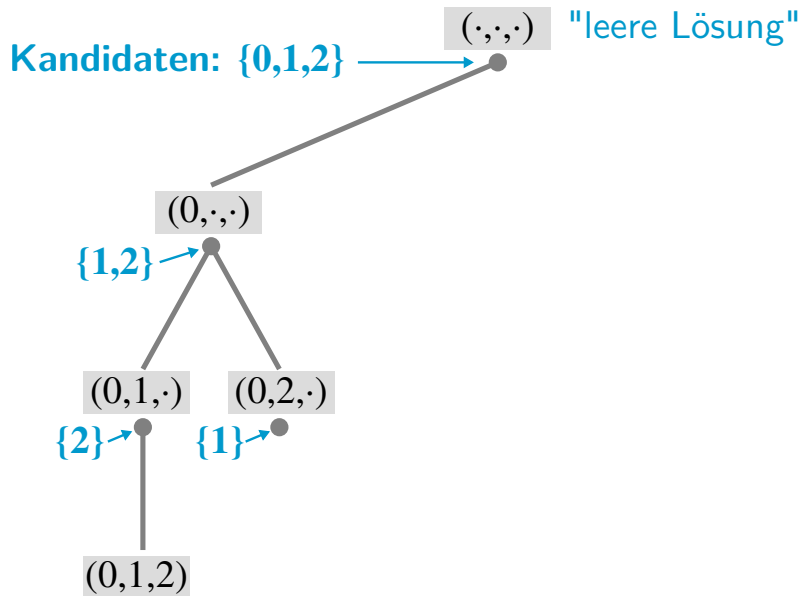
Beispiel: Aufzählung aller Permutationen



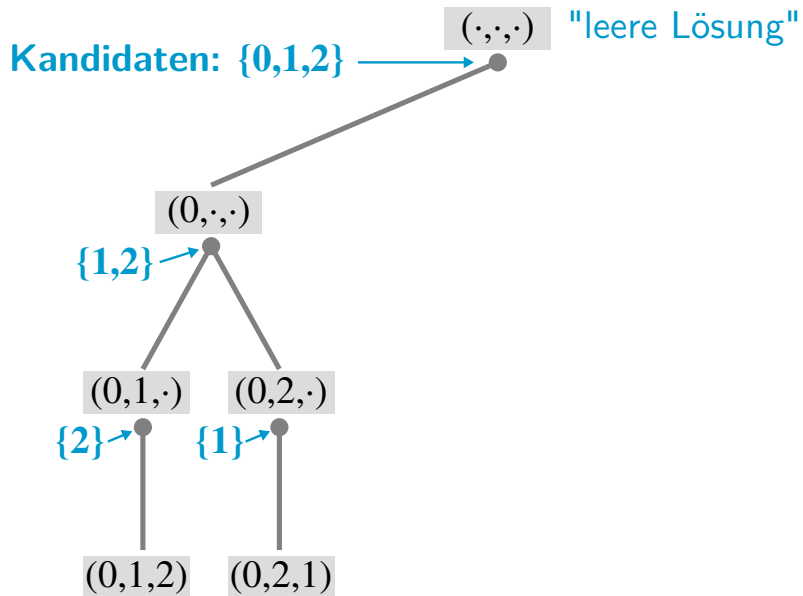
Beispiel: Aufzählung aller Permutationen



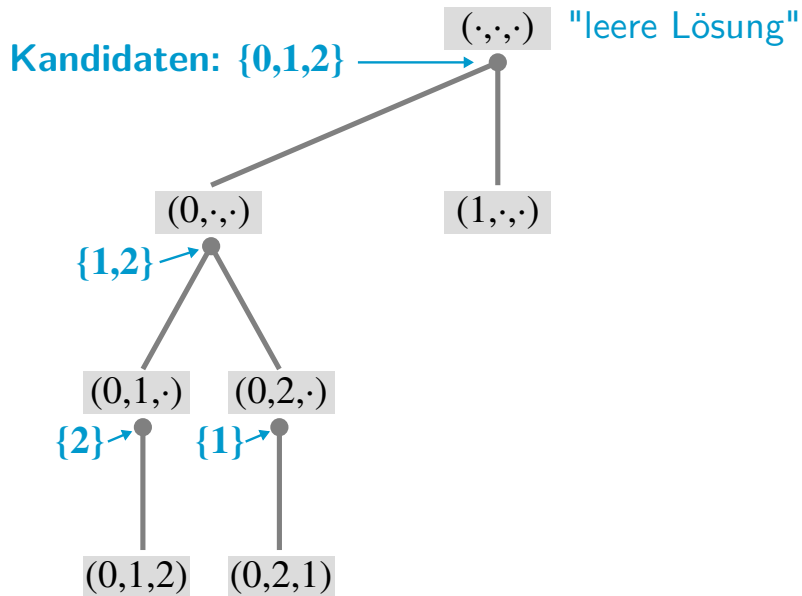
Beispiel: Aufzählung aller Permutationen



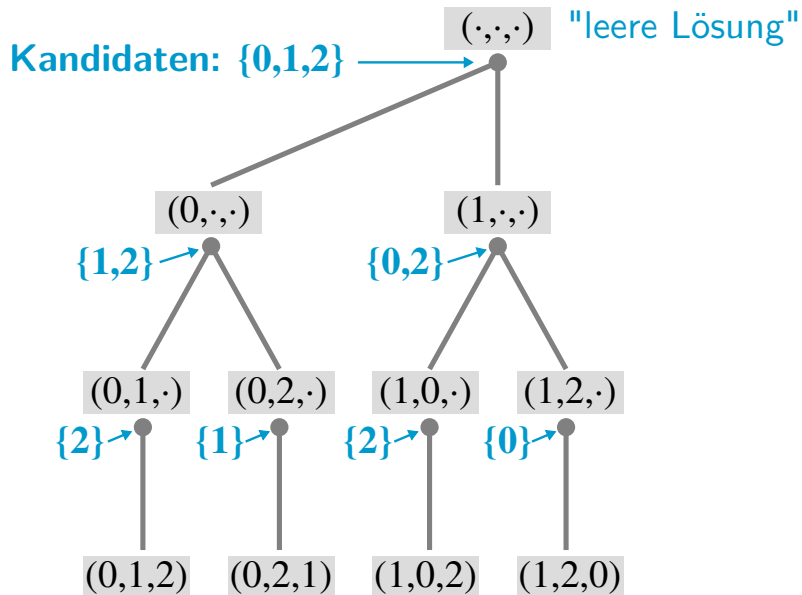
Beispiel: Aufzählung aller Permutationen



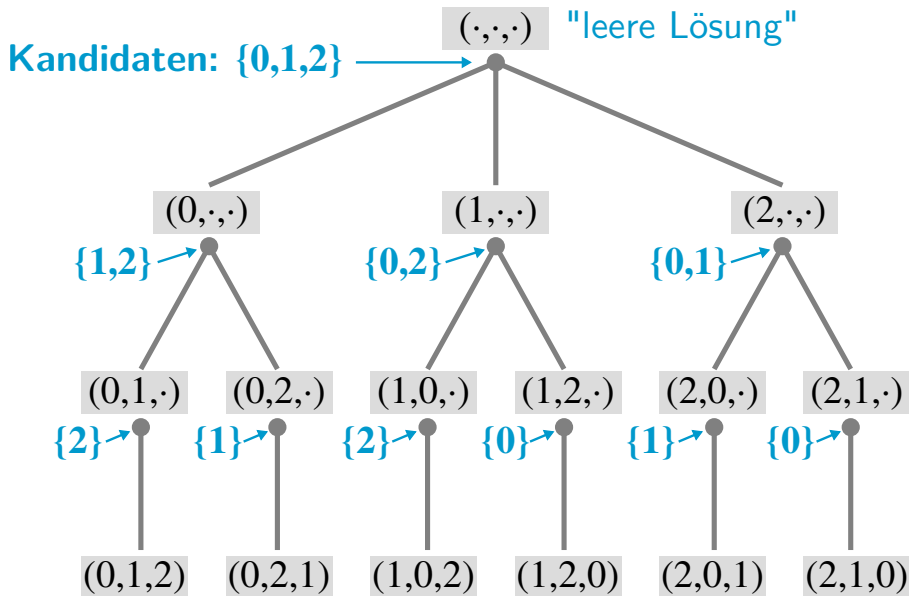
Beispiel: Aufzählung aller Permutationen



Beispiel: Aufzählung aller Permutationen



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*

```
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);  
            }  
        }  
    }  
}
```

```
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();  
    }  
}
```

Sehr positiv zu vermerken:

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

Sehr positiv zu vermerken:

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

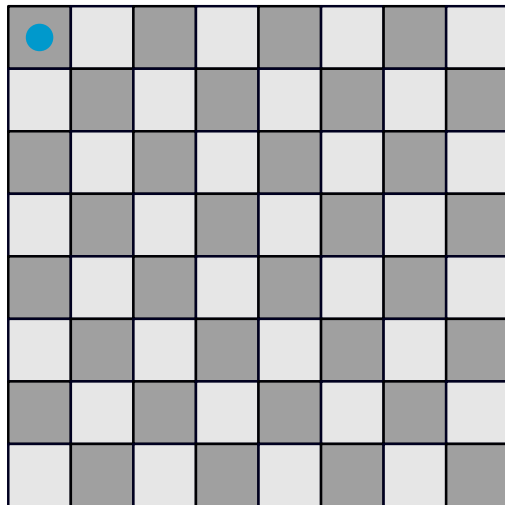
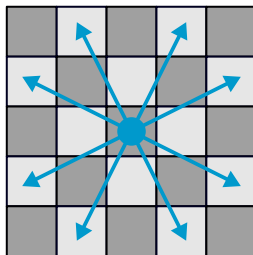
Zur Erzeugung der Permutationen haben wir folgende Elemente benutzt:

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

Bevor wir zu dem generellen Prinzip kommen, besprechen wir noch ein zweites Beispiel.

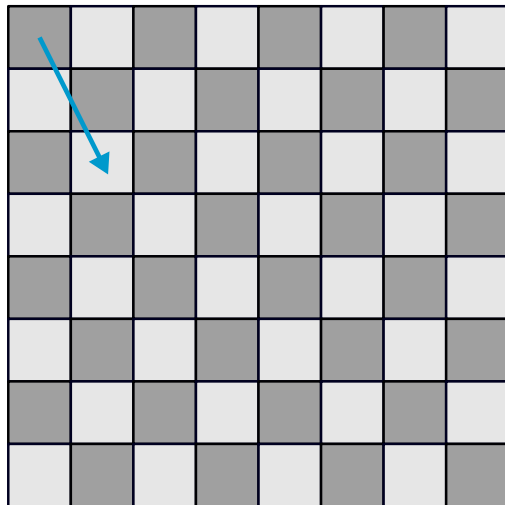
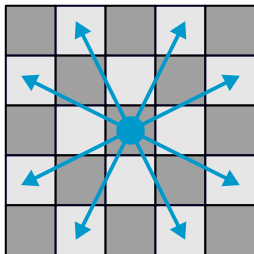
Backtracking Beispiel: Rösselsprung

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



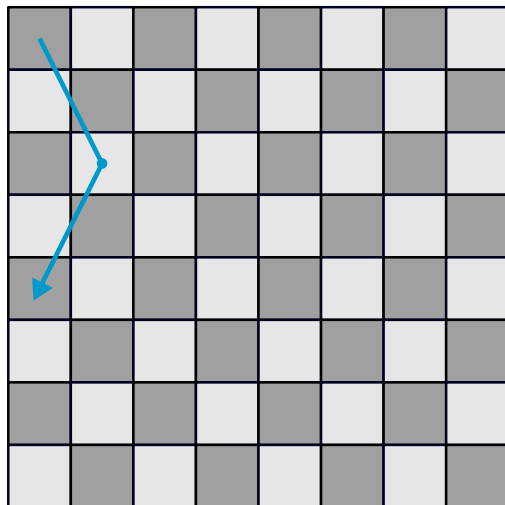
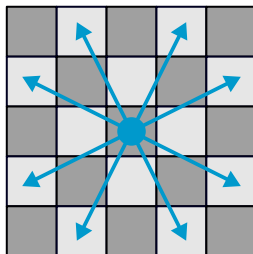
Backtracking Beispiel: Rösselsprung

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



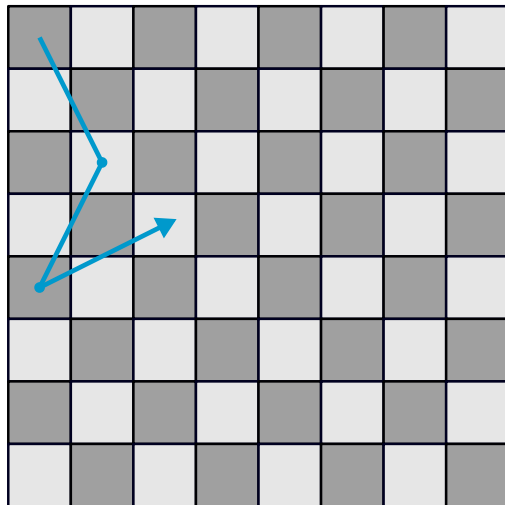
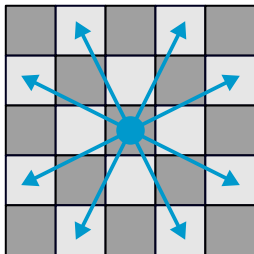
Backtracking Beispiel: Rösselsprung

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



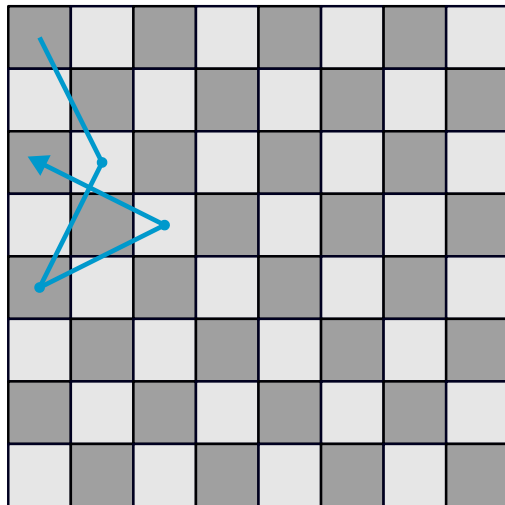
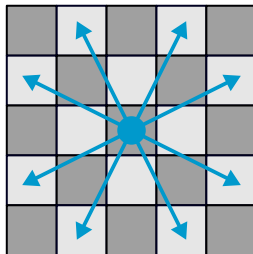
Backtracking Beispiel: Rösselsprung

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



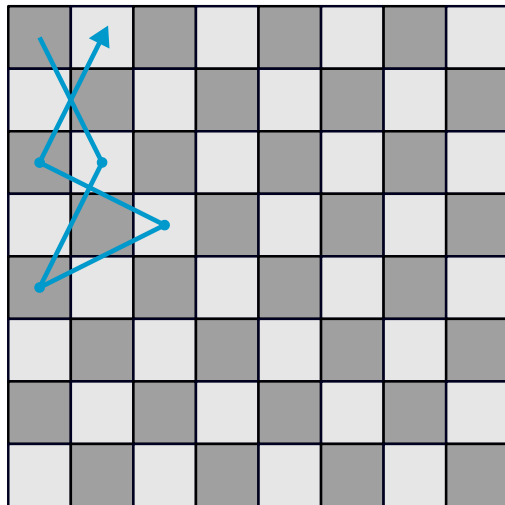
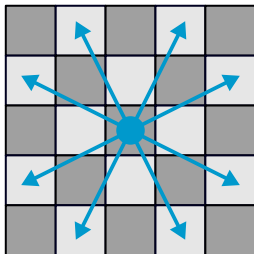
Backtracking Beispiel: Rösselsprung

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



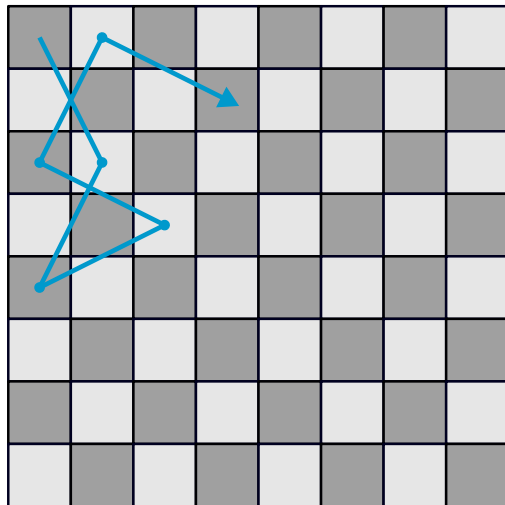
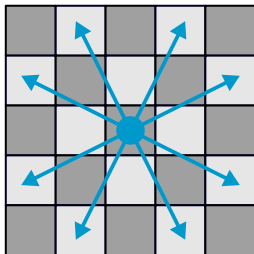
Backtracking Beispiel: Rösselsprung

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



Backtracking 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:



- ▶ Wir brauchen zunächst Strukturen, um Brett und Spielzüge zu realisieren.
- ▶ Diese implementieren wir in den Klassen `Position` (Position auf Spielbrett), `Move` (Spielzug) und `Board` (Spielbrett).

- ▶ Wir brauchen zunächst Strukturen, um Brett und Spielzüge zu realisieren.
- ▶ Diese implementieren wir in den Klassen `Position` (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;
    }
}
```

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.

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.

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-Conquer*.

Generelles Backtrack-Prinzip

Die gezeigten Beispiele motivieren das folgende allgemeine Backtrack-Prinzip zum Generieren **aller Lösungen** für ein Problem:

- 1 *Starte mit leerer Lösung*
- 2 *Rekursion:*
 - 3 *Generiere Kandidaten für nächsten Schritt in aktueller Teillösung // [1]*
 - 4 *Für jeden Kandidaten:*
 - 5 *Führe Schritt aus // [2]*
 - 6 *Falls dies eine Lösung ergibt: ausgeben bzw. speichern // [3]*
 - 7 *Gehe in die Rekursion für den nächsten Schritt*
 - 8 *Mache Schritt rückgängig // [4]*

Generelles Backtrack-Prinzip

Die gezeigten Beispiele motivieren das folgende allgemeine Backtrack-Prinzip zum Generieren **aller Lösungen** für ein Problem:

- 1 *Starte mit leerer Lösung*
- 2 *Rekursion:*
 - 3 *Generiere Kandidaten für nächsten Schritt in aktueller Teillösung // [1]*
 - 4 *Für jeden Kandidaten:*
 - 5 *Führe Schritt aus // [2]*
 - 6 *Falls dies eine Lösung ergibt: ausgeben bzw. speichern // [3]*
 - 7 *Gehe in die Rekursion für den nächsten Schritt*
 - 8 *Mache Schritt rückgängig // [4]*

Elemente des **Backtracking**:

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

Bemerkungen zum allgemeinen Backtrack-Prinzip

- ▶ *Schritt rückgängig machen* ist in manchen Fällen überflüssig (wie im Permutationen Beispiel)
- ▶ Das *Prüfen auf Lösung* wird manchmal am Anfang der *Backtracking* Methode durchgeführt.
- ▶ Mögliche Verbesserung: Wähle aus, für welchen Lösungsschritt als nächstes eine Entscheidung getroffen wird, siehe Sudoku Beispiel.

Bemerkungen zum allgemeinen Backtrack-Prinzip

- ▶ *Schritt rückgängig machen* ist in manchen Fällen überflüssig (wie im Permutationen Beispiel)
- ▶ Das *Prüfen auf Lösung* wird manchmal am Anfang der *Backtracking* Methode durchgeführt.
- ▶ Mögliche Verbesserung: Wähle aus, für welchen Lösungsschritt als nächstes eine Entscheidung getroffen wird, siehe Sudoku Beispiel.
- ▶ Falls **nur eine Lösung** gesucht wird (wie bei unserem Rösselsprung):
 - Breche in Zeile 6 ab, falls eine Lösung gefunden wurde und setze eine Variable oder Rückgabewert
 - Prüfe nach Zeile 7, ob beim Rekursionsaufruf eine Lösung gefunden wurde: falls ja, springe aus der Schleife

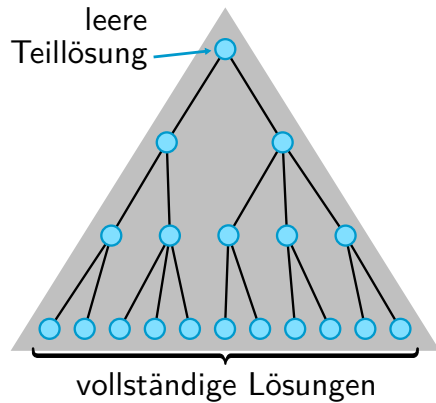
Baum der Teillösungen

- ▶ Lösungen können als Lösungsbaum schrittweise entwickelt werden.
- ▶ Bei jeder Verzweigung ist eine Auswahl aus einer relativ begrenzten Auswahl an Möglichkeiten zu treffen.
- ▶ Die **leere Lösung** ist die Teillösung, bei der noch keine Entscheidung getroffen wurde. Alle Lösungen, lassen sich aus der leeren Lösung durch eine endliche Anzahl von Lösungsschritten entwickeln.

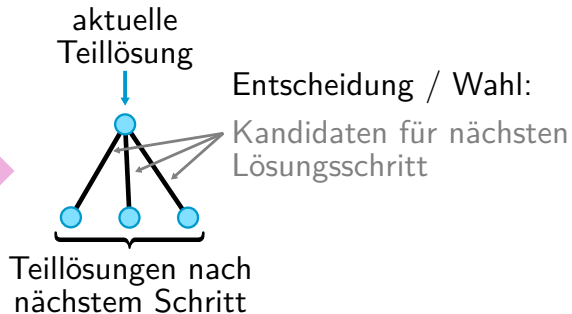
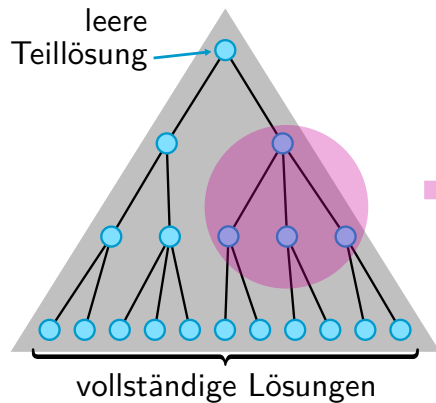
Baum der Teillösungen

- ▶ Lösungen können als Lösungsbaum schrittweise entwickelt werden.
- ▶ Bei jeder Verzweigung ist eine Auswahl aus einer relativ begrenzten Auswahl an Möglichkeiten zu treffen.
- ▶ Die **leere Lösung** ist die Teillösung, bei der noch keine Entscheidung getroffen wurde. Alle Lösungen, lassen sich aus der leeren Lösung durch eine endliche Anzahl von Lösungsschritten entwickeln.
- ▶ An der Wurzel steht die leere Teillösung.
- ▶ Jeder Knoten steht für die Entscheidung in einem bestimmten Lösungsschritt.
- ▶ Jede Auswahlmöglichkeit entspricht einer Kante.
- ▶ Dies ergibt nur dann einen Baum, wenn es zu keinen Zyklen kommen kann.

Baum der Teillösungen

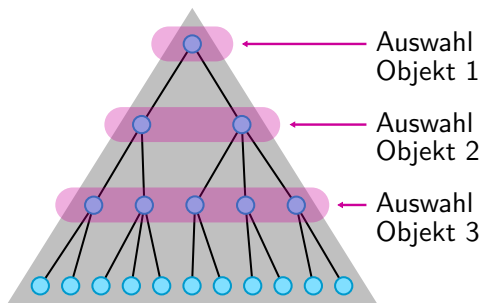


Baum der Teillösungen



- **statisch:** Es steht vorher fest, welcher Knoten welche Entscheidung repräsentiert.
(Beispiele Permutationen und Rösselsprung)

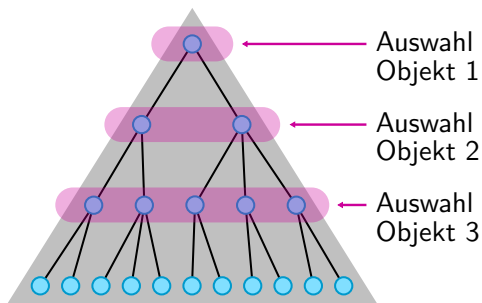
Statischer Suchbaum



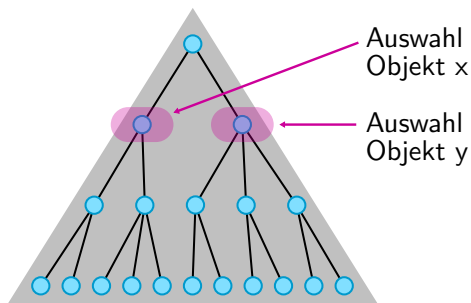
Backtracking+: Statische vs. dynamische Baumstruktur

- ▶ **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. (Erweiterung im Sudoku Beispiel)

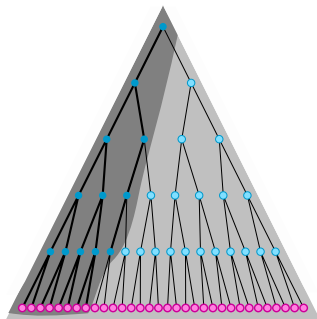
Statischer Suchbaum



Dynamischer Suchbaum



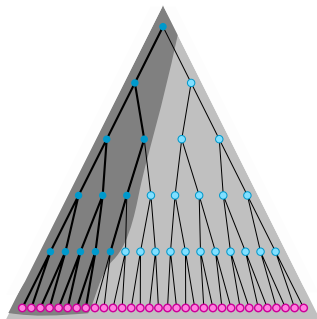
Tiefensuche



Sudoku: wähle nächstes
freie Kästchen

Backtracking and beyond: Reihenfolge des Expandierens

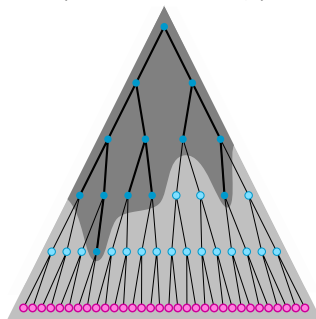
Tiefensuche



Sudoku: wähle nächstes
freie Kästchen

Dynamische Suche

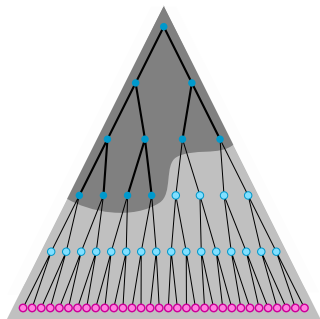
(informierte Reihenfolge)



Sudoku: wähle nächstes
Kästchen intelligent aus

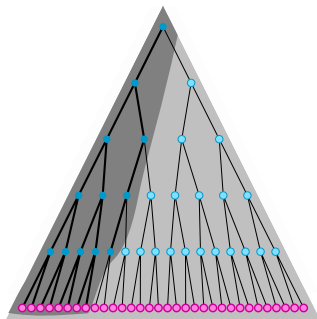
Backtracking and beyond: Reihenfolge des Expandierens

Breitensuche



Finde Schachmatt mit
geringster Zuganzahl

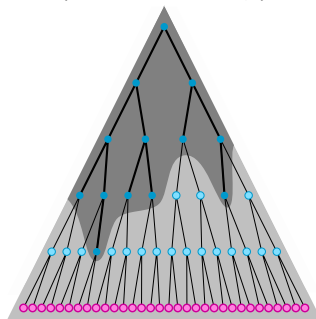
Tiefensuche



Sudoku: wähle nächstes
freie Kästchen

Dynamische Suche

(informierte Reihenfolge)



Sudoku: wähle nächstes
Kästchen intelligent aus

- ▶ Üblicherweise versteht man unter *Backtracking* nur die Variante mit **Tiefensuche** und **statischem Suchbaum**.
- ▶ Breitensuche ist aufwändiger, weil parallele Teillösungen gespeichert werden müssen.
- ▶ Letzteres ist keine Einschränkung, wenn ein Baum oder Graph zur Durchsuchung gegeben ist.

- ▶ Üblicherweise versteht man unter *Backtracking* nur die Variante mit **Tiefensuche** und **statischem Suchbaum**.
- ▶ Breitensuche ist aufwändiger, weil parallele Teillösungen gespeichert werden müssen.
- ▶ Letzteres ist keine Einschränkung, wenn ein Baum oder Graph zur Durchsuchung gegeben ist.
- ▶ Dynamische Suche ist hilfreich, wenn nur bestimmte Lösungen gesucht werden.
- ▶ Dazu benötigt man eine Kriterium bzw. eine Heuristik, die angibt, in welcher Richtung die Lösungen (wahrscheinlich) zu finden sind.
- ▶ Dies wird systematisch in Verfahren wie *Branch-and-Bound* und heuristischen Algorithmen verfolgt.

Beispiel: Sudoku

- ▶ Als Beispiel mit *Backtracking*+ Erweiterungen betrachten wir die Lösung von **Sudoku** Rätseln.

				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:
 - ▶ **Lösungskandidaten:** Alle Ziffern, die nach den Regeln passen
 - ▶ **Teillösung vollständig?** Wenn alle Felder gefüllt sind
 - ▶ **Schritt ausführen:** Ziffer in ein Kästchen speichern
 - ▶ **Schritt rückgängig machen:** Ziffer aus dem Kästchen löschen

Methoden des *Backtracking* zum Sudoku Lösen

- Wir legen die *Backtracking* Elemente fest:
 - ▶ **Lösungskandidaten:** Alle Ziffern, die nach den Regeln passen
 - ▶ **Teillösung vollständig?** Wenn alle Felder gefüllt sind
 - ▶ **Schritt ausführen:** Ziffer in ein Kästchen speichern
 - ▶ **Schritt rückgängig machen:** Ziffer aus dem Kästchen löschen
- ▶ Mit diesen einfachen Methoden im Rahmen des generellen Backtrack-Prinzip lässt sich leicht ein Programm zum Lösen von Sudoku Rätsel implementieren.
- ▶ Die Effizienz ist allerdings nicht so gut, insbesondere bei Rätseln der schwierigen Kategorie.

- Wir betrachten zwei Verbesserungen, die dazu führen, dass aussichtsreiche Lösungsschritte zuerst verfolgt werden und hoffnungslose Teillösungen frühzeitig erkannt und nicht weiter expandiert werden:
- ▶ **Vorausschauende Kandidatenmenge:** Falls für irgendein Kästchen keine Lösungskandidaten mehr existieren, gib für jeden Auswahlschritt eine leere Auswahl zurück. Die Teillösung kann nicht vervollständigt werden, daher sollte dieser Ast des Baumes nicht expandiert werden.
- ▶ **Eingegrenzte Auswahl:** Nimm als nächsten Auswahlschritt dasjenige Kästchen, für das am wenigsten passende Kandidaten existieren (ähnlich der Warnsdorff Regel beim Rösselsprung).

- Wir betrachten zwei Verbesserungen, die dazu führen, dass aussichtsreiche Lösungsschritte zuerst verfolgt werden und hoffnungslose Teillösungen frühzeitig erkannt und nicht weiter expandiert werden:
- ▶ **Vorausschauende Kandidatenmenge:** Falls für irgendein Kästchen keine Lösungskandidaten mehr existieren, gib für jeden Auswahlschritt eine leere Auswahl zurück. Die Teillösung kann nicht vervollständigt werden, daher sollte dieser Ast des Baumes nicht expandiert werden.
- ▶ **Eingegrenzte Auswahl:** Nimm als nächsten Auswahlschritt dasjenige Kästchen, für das am wenigsten passende Kandidaten existieren (ähnlich der Warnsdorff Regel beim Rösselsprung).
- ▶ Mit diesen kleinen Verbesserungen erhält man ein effizientes Programm zum Sudoku lösen, siehe Tabelle auf der nächsten Seite.

Laufzeit Vergleich der Sudoku Methoden

- Die Tabelle zeigt die Anzahl der Lösungsschritte beim Lösen von drei Sudoku Rätseln verschiedener Schwierigkeitsstufen im Vergleich der unterschiedlichen Auswahlmethoden in einer Simulation von [Skiena 2008; S. 243].

Auswahlkriterium	Sudoku Schwierigkeitsgrad		
	leicht	mittel	schwer
einfach	1.904.832	863.305	(terminiert nicht)
vorausschauend	127	142	12.507.212
eingegrenzt & vorausschauend	48	65	10.374

Laufzeit Vergleich der Sudoku Methoden

- Die Tabelle zeigt die Anzahl der Lösungsschritte beim Lösen von drei Sudoku Rätseln verschiedener Schwierigkeitsstufen im Vergleich der unterschiedlichen Auswahlmethoden in einer Simulation von [Skiena 2008; S. 243].
- Rechts das schwierige aus [Skiena 2008].

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

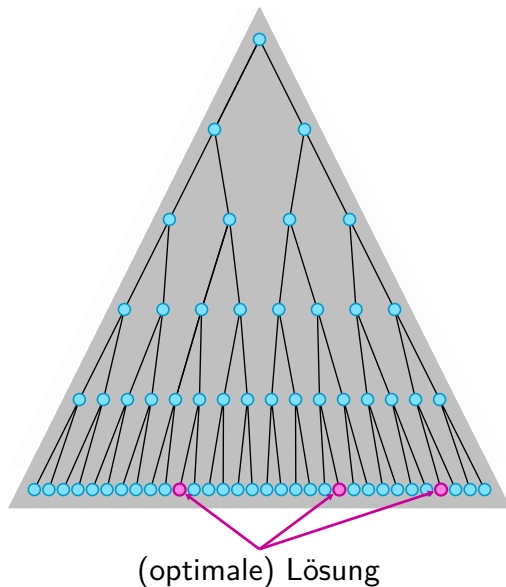
Auswahlkriterium	Sudoku Schwierigkeitsgrad		
	leicht	mittel	schwer
einfach	1.904.832	863.305	(terminiert nicht)
vorausschauend	127	142	12.507.212
eingegrenzt & vorausschauend	48	65	10.374

- ▶ Manchmal lässt sich der Lösungsraum durch Vorüberlegungen beschränken. Ein typisches Beispiel ist das Ausnutzen von 'Symmetrien'.
- ▶ Aussichtsreiche Kandidaten bevorzugt weiterverfolgen, damit früher eine Lösung gefunden wird
- ▶ Kandidaten weglassen, die zu keiner Lösung führen können
- Und bei Optimierungsproblemen:
 - ▶ Kandidaten weglassen, die zu keiner *optimalen* Lösung führen können

- ▶ Manchmal lässt sich der Lösungsraum durch Vorüberlegungen beschränken. Ein typisches Beispiel ist das Ausnutzen von 'Symmetrien'.
- ▶ Aussichtsreiche Kandidaten bevorzugt weiterverfolgen, damit früher eine Lösung gefunden wird
- ▶ Kandidaten weglassen, die zu keiner Lösung führen können
- Und 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).

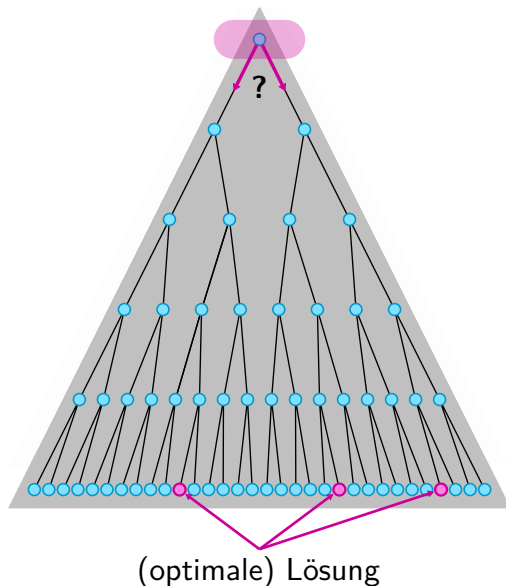
Motivation von Greedy-Algorithmen

- ▶ 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 .



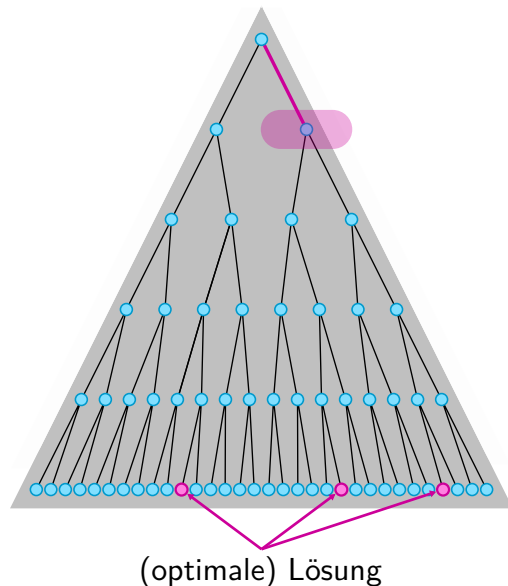
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



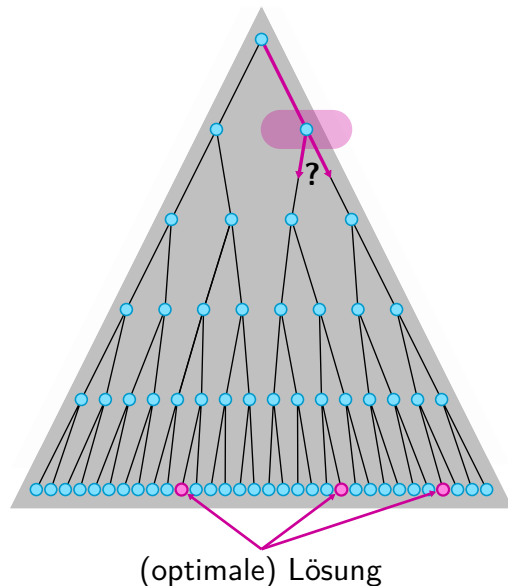
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



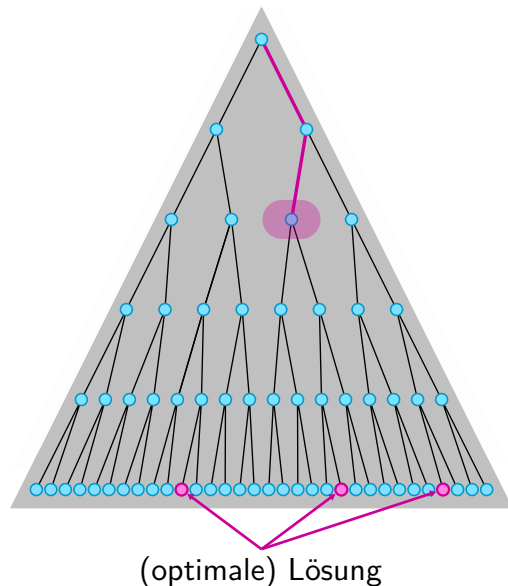
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



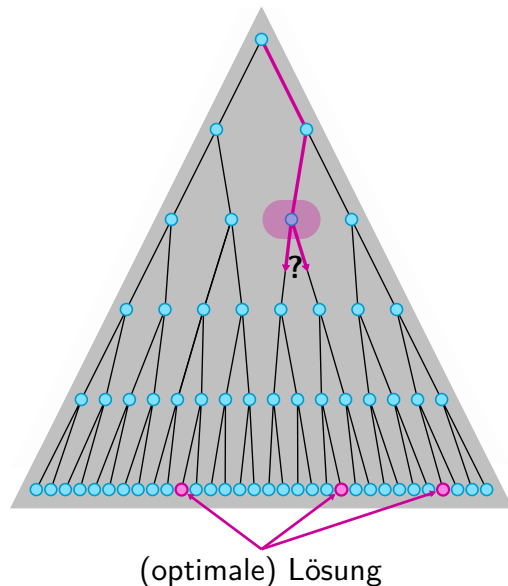
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



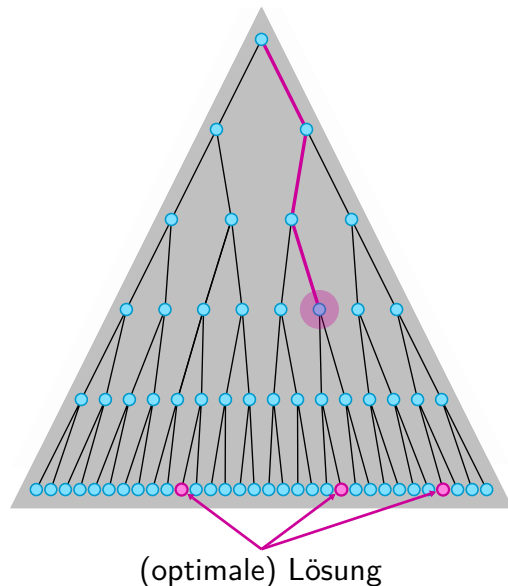
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



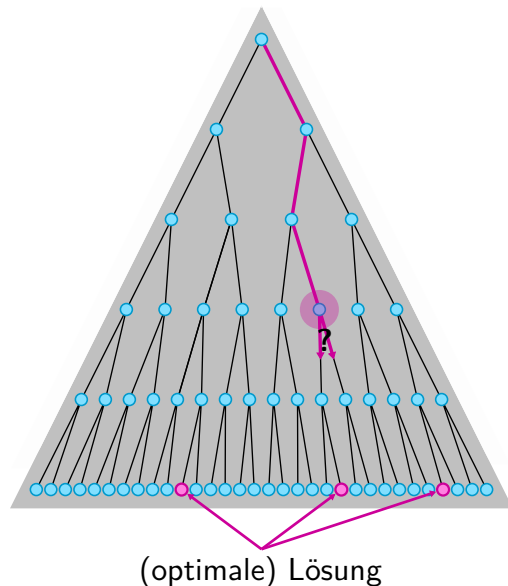
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



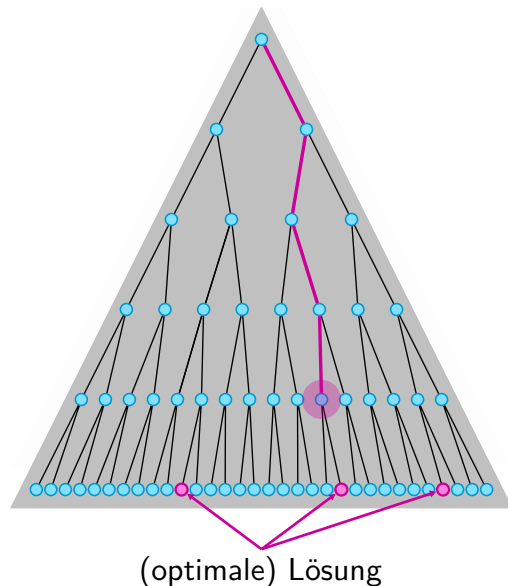
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



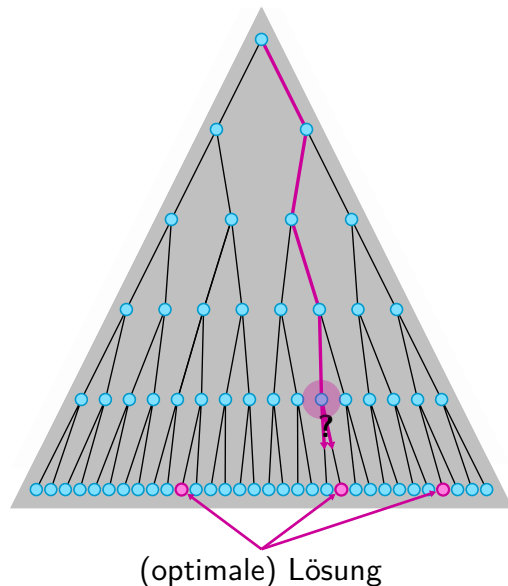
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



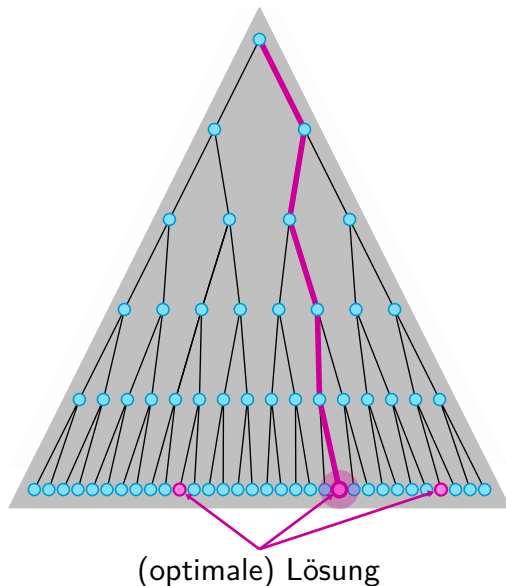
Motivation von Greedy-Algorithmen

- ▶ 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 Situation (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.



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 Situation (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.

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.
- ▶ Die *Greedy* Suche ist eine 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).

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.
- ▶ Die *Greedy* Suche ist eine 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 es zu einem Problem einen Greedy-Algorithmus gibt, der optimal ist, dann ist dies oft eine effiziente Lösung.
- ▶ 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

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 $1, \dots, n$ 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:** Erstelle eine Belegung für die Resource, die **möglichst viele** Anfragen erfüllt.

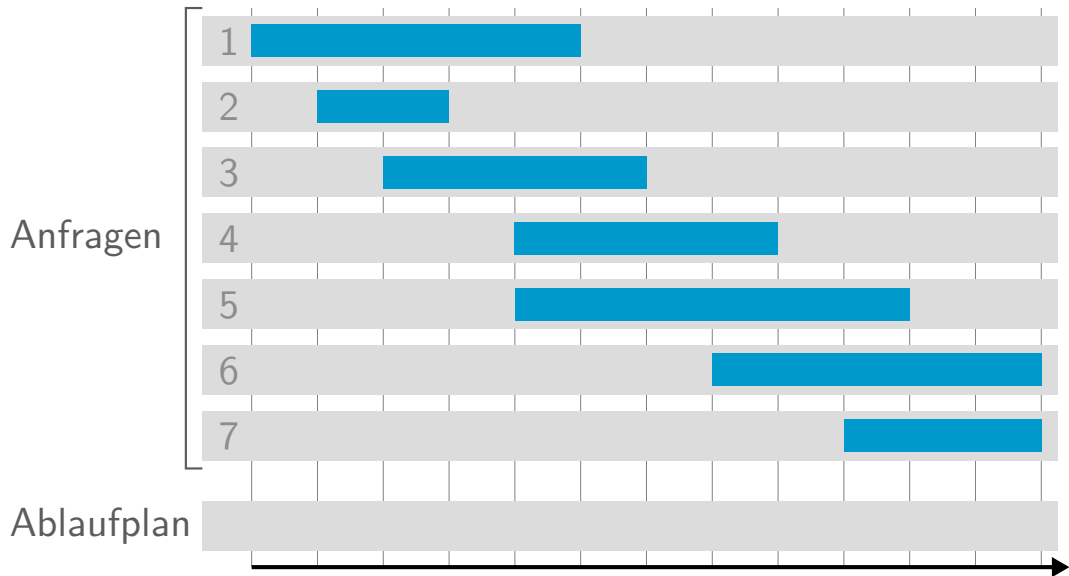
Greedy Beispiel: Intervallauswahl

- ▶ Als Beispiel betrachten wir eine Variante der Ablaufplanung, die **Intervallauswahl** (*interval scheduling*)
- ▶ Es gibt Anfragen $1, \dots, n$ 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:** Erstelle eine Belegung für die Resource, die **möglichst viele** Anfragen erfüllt.
- ▶ Wir nennen eine Menge von Intervallen **kompatibel**, wenn es keine Konflikte (Überlappungen) zwischen den Intervallen gibt.
- ▶ Damit kann das Ziel auch so formuliert werden: Wähle eine möglichst große Menge von kompatiblen Intervallen aus.

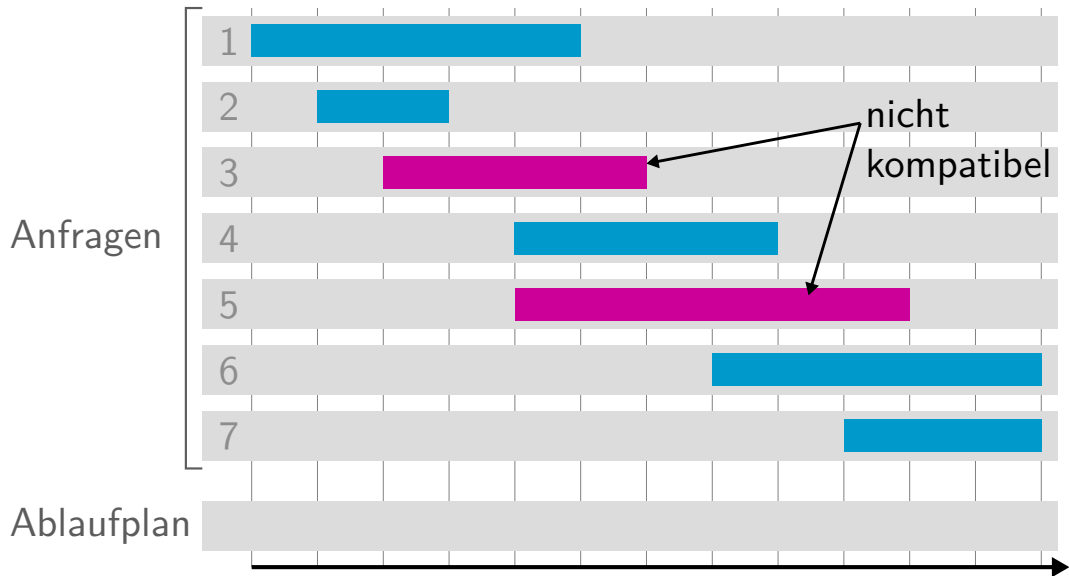
Greedy Beispiel: Intervallauswahl

- ▶ Als Beispiel betrachten wir eine Variante der Ablaufplanung, die **Intervallauswahl** (*interval scheduling*)
- ▶ Es gibt Anfragen $1, \dots, n$ 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:** Erstelle eine Belegung für die Resource, die **möglichst viele** Anfragen erfüllt.
- ▶ Wir nennen eine Menge von Intervallen **kompatibel**, wenn es keine Konflikte (Überlappungen) zwischen den Intervallen gibt.
- ▶ Damit kann das Ziel auch so formuliert werden: Wähle eine möglichst große Menge von kompatiblen Intervallen aus.
- ▶ Ein Greedy-Algorithmus durchläuft die Anfragen in einer gewissen Reihenfolge und entscheidet, ob die Anfrage akzeptiert wird. Wir betrachten unterschiedliche Sortierungen und diskutieren ihre Tauglichkeit.

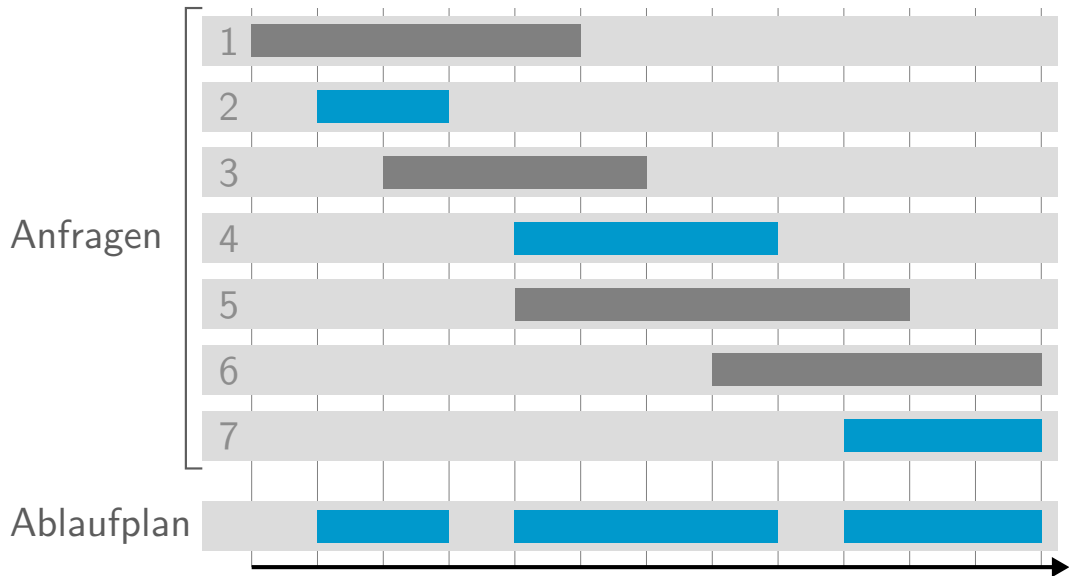
Intervallauswahl – Beispiel



Intervallauswahl – Beispiel



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

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

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

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.

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?

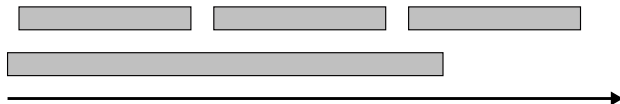
Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

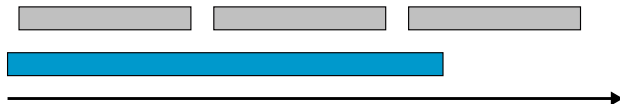
- ▶ **[Frühester-Start]**



Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

- ▶ **[Frühester-Start]**



Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

- ▶ **[Frühester-Start]**



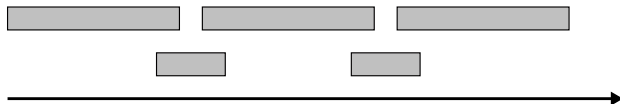
Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



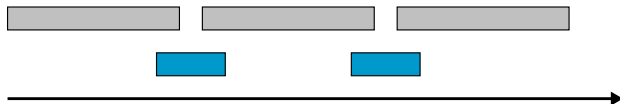
Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



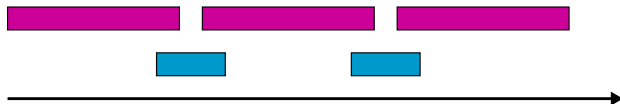
Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



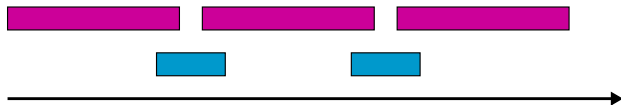
Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



- ▶ **[Wenigste-Konflikte]**

Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

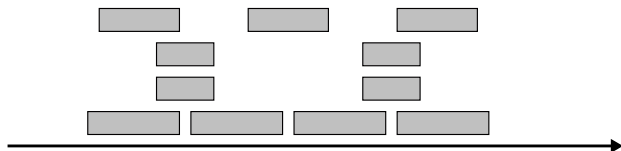
- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



- ▶ **[Wenigste-Konflikte]**



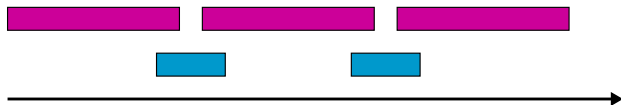
Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

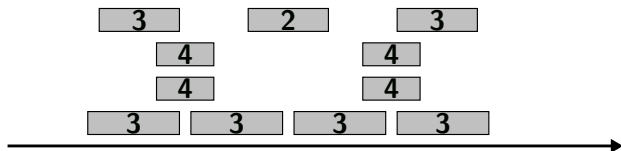
- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



- ▶ **[Wenigste-Konflikte]**



Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

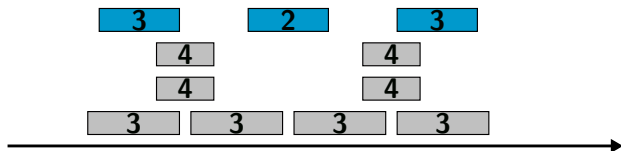
- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



- ▶ **[Wenigste-Konflikte]**



Gegenbeispiele zur Optimalität der vorgeschlagenen Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

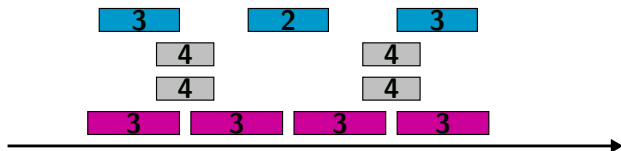
- ▶ **[Frühester-Start]**



- ▶ **[Kürzestes-Intervall]**



- ▶ **[Wenigste-Konflikte]**



- ▶ 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.

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 $\mathcal{O}(n \log n)$.

- ▶ Seien a_1, \dots, a_k die ausgewählten Intervalle in der Reihenfolge der Auswahl und o_1, \dots, 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).

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.

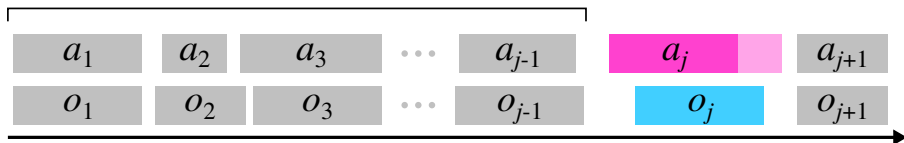
Dieser Algorithmus liefert eine optimale Intervallplanung in einer Laufzeit in $O(n \log n)$.

- ▶ Seien a_1, \dots, a_k die ausgewählten Intervalle in der Reihenfolge der Auswahl und o_1, \dots, 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, \dots, 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} auswählen 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. \square

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 'früheste Endzeit'.
- ▶ Nach IV können wir voraussetzen, dass $f(a_i) \leq f(o_i)$ für alle $i < j$ gilt.

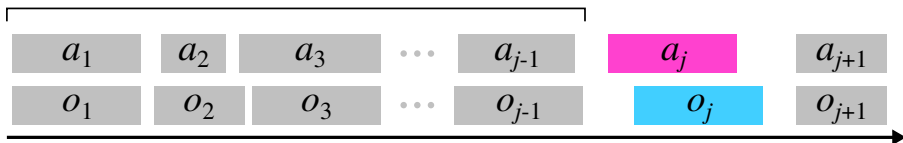
a_i nicht-spätere Endzeit als o_i nach IV



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 'früheste 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, \dots, a_{j-1} , denn es gilt $s(o_j) \geq f(o_{j-1}) \stackrel{\text{IV}}{\geq} f(a_{j-1})$. 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.

a_i nicht-spätere Endzeit als o_i nach IV



Pseudocode des Greedy Algorithmus für *Interval Scheduling*

```
1 // Gegeben:  $n$  Intervalle
2 sortiere Intervalle nach Endzeiten  $\rightarrow I_1, \dots, I_n$ 
3  $R \leftarrow \emptyset$ 
4 for  $j = 1$  to  $n$ 
5   if Intervall  $I_j$  ist kompatibel mit  $R$ 
6     füge  $I_j$  zu  $R$  hinzu
7   end
8 end
```

Pseudocode des Greedy Algorithmus für *Interval Scheduling*

```
1 // Gegeben:  $n$  Intervalle
2 sortiere Intervalle nach Endzeiten  $\rightarrow I_1, \dots, I_n$ 
3  $R \leftarrow \emptyset$ 
4 for  $j = 1$  to  $n$ 
5   if Intervall  $I_j$  ist kompatibel mit  $R$ 
6     füge  $I_j$  zu  $R$  hinzu
7   end
8 end
```

- ▶ Für die Kompatibilitätsprüfung kann einfach der Endzeitpunkt des letzten Intervalls von R gespeichert werden.
- ▶ Das Durchlaufen der Intervalle nach Endzeitpunkten kann über eine Vorrangwarteschlange PQ implementiert werden. Dadurch ergibt sich die Laufzeit $O(n \log n)$.

Pseudocode des Greedy Algorithmus für *Interval Scheduling*

```
1 // Gegeben:  $n$  Intervalle
2 sortiere Intervalle nach Endzeiten  $\rightarrow I_1, \dots, I_n$ 
3  $R \leftarrow \emptyset$ 
4 for  $j = 1$  to  $n$ 
5   if Intervall  $I_j$  ist kompatibel mit  $R$ 
6     füge  $I_j$  zu  $R$  hinzu
7   end
8 end
```

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

Implementation des Greedy Algorithmus für *Interval Scheduling*

```
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.

Implementation des Greedy Algorithmus für *Interval Scheduling*

```
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.

```
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;  
    }  
}
```

- ▶ 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*.

- ▶ 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 $S \subseteq \{1, \dots, n\}$ gemäß folgender Optimierung zu wählen:

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

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 $S \subseteq \{1, \dots, n\}$ gemäß folgender Optimierung zu wählen:

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

- Dieses Aufgabenstellung wird auch 0/1-Rucksackproblem genannt.
- Die Auswahl könnte auch über einen Vektor $a \in \{0, 1\}^n$ 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:

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

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   $r \leftarrow W$            // Restkapazität  
3  for  $i = 1$  to  $n$   
4     $a_i \leftarrow \min(1, r / w_i)$   
5     $r \leftarrow r - a_i w_i$   
6  end
```

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 $\mathbf{a} = (1, \dots, 1, a_{k+1}, 0, \dots, 0)$ für ein $k < n$.
- ▶ Wir zeigen, dass das Verschieben von Anteilen nach hinten die Lösung nicht verbessern kann.
- ▶ Sei $i < k$ und $j \geq k$. Wir reduzieren a_i von 1 auf $\alpha < 1$ und erhöhen a_j entsprechend. Durch die Reduktion von Objekt i sinkt das Gewicht um $(1 - \alpha)w_i$.

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 $\mathbf{a} = (1, \dots, 1, a_{k+1}, 0, \dots, 0)$ für ein $k < n$.
- ▶ Wir zeigen, dass das Verschieben von Anteilen nach hinten die Lösung nicht verbessern kann.
- ▶ Sei $i < k$ und $j \geq k$. 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

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

Fortsetzung des Beweises.

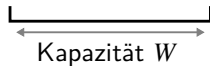
- ▶ Der Gesamtwert ändert sich durch die Verschiebung um

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

- ▶ 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. \square

Illustration des Optimalitätsbeweises

Rucksack



Objekte

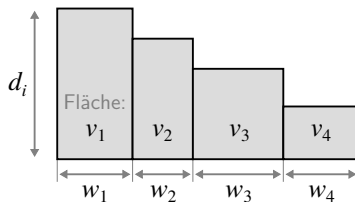


Illustration des Optimalitätsbeweises

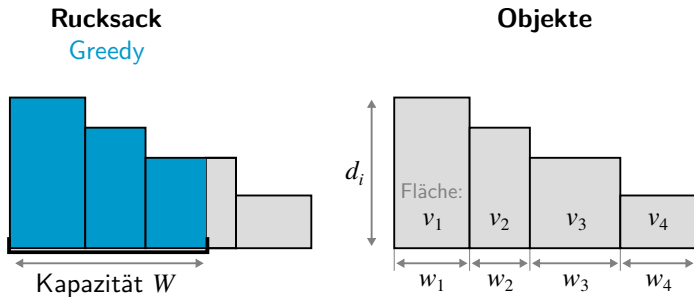


Illustration des Optimalitätsbeweises

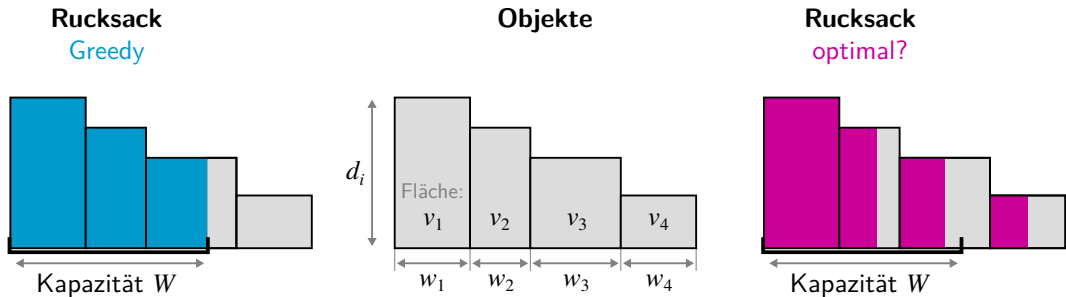


Illustration des Optimalitätsbeweises

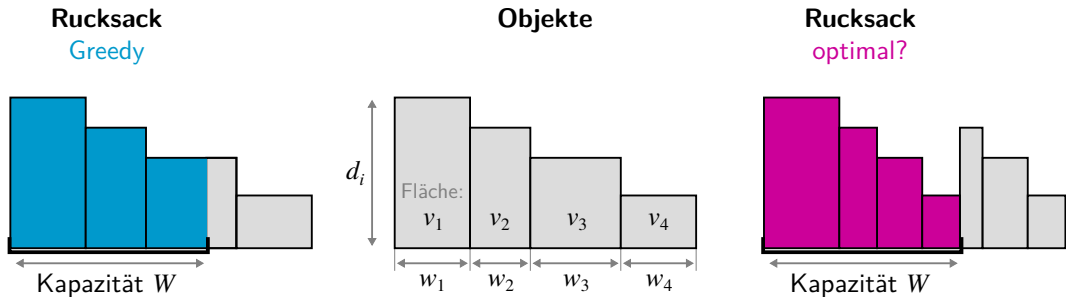
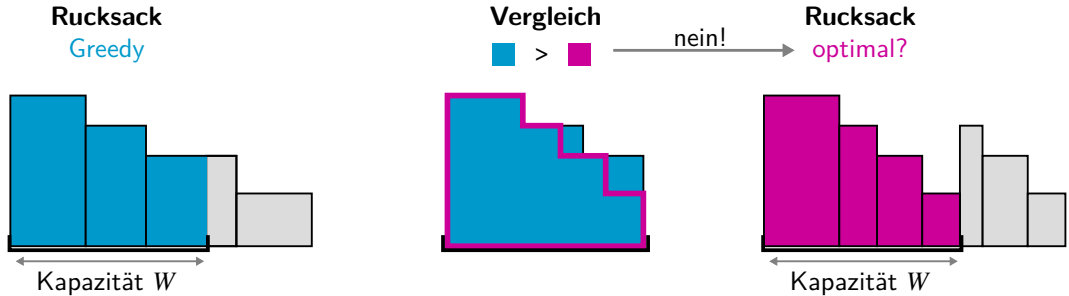


Illustration des Optimalitätsbeweises

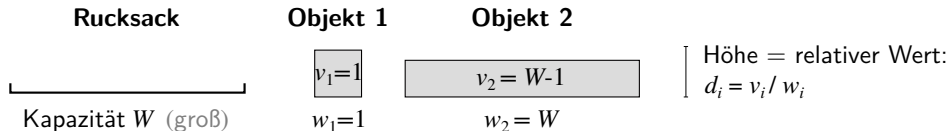


Einschränkung des Greedy-Algorithmus

- ▶ 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:

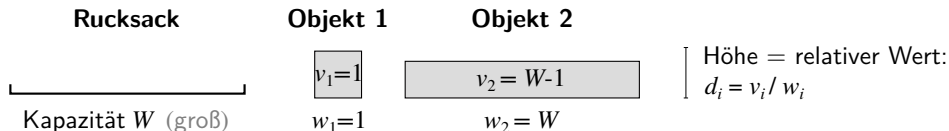
Einschränkung des Greedy-Algorithmus

- ▶ 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 $d_2 = \frac{W-1}{W} < 1$.



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]

- **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.

Inhalt des Anhangs: (ergänzendes Material, nicht prüfungsrelevant)

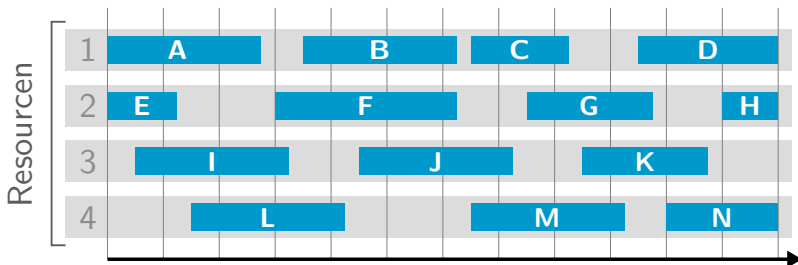
- Weitere Greedy-Algorithmen und Optimalitätsbeweise
 - ▶ Ablaufplanung – Intervallverteilung (*Interval Partitioning*): S. 54
 - ▶ Ablaufplanung mit minimaler Verspätung (*Schedule to Minimize Lateness*): S. 64

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*).

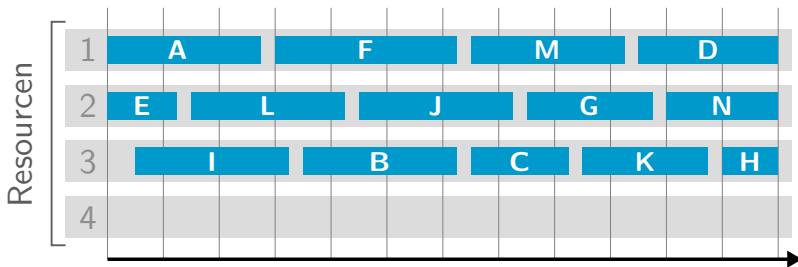
Greedy Beispiel: Intervallverteilung

- ▶ 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 $1, \dots, n$ 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.



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 $1, \dots, n$ 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.

Gierige Auswahl bei der Intervallverteilung

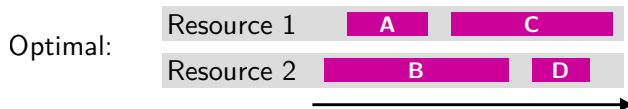
Wir betrachten die Intervalle in einer gewissen Ordnung (siehe Liste) und verteilen die jeweilige Anforderung auf eine Ressource (welche?), zu deren bisherigen Intervalle sie kompatibel ist.

- ▶ **[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?

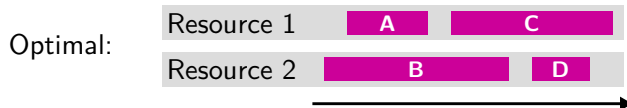
Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

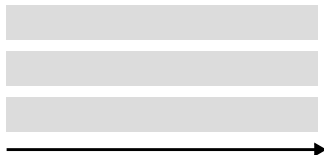


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)

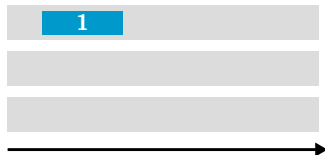


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)

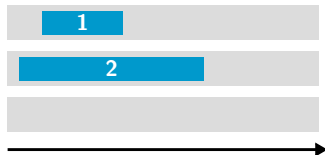


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)

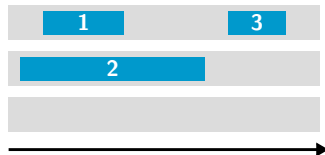


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

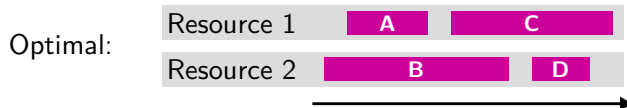


Frühstes-Ende (ABDC)

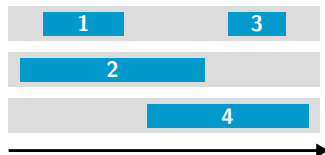


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

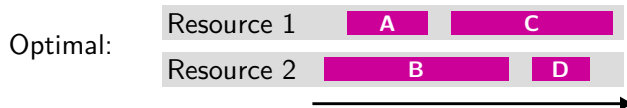


Frühstes-Ende (ABDC)

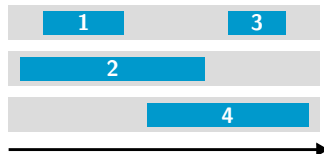


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

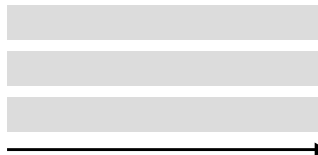
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)

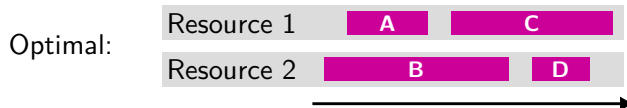


Kürzestes-Intervall (DACB)

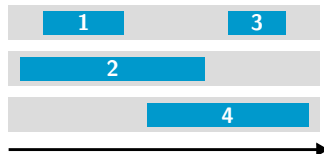


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

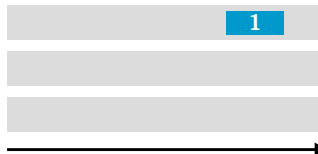
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)

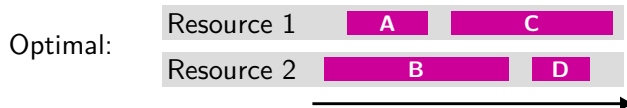


Kürzestes-Intervall (DACB)

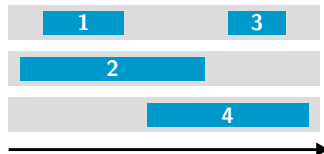


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

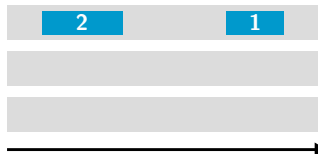
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)

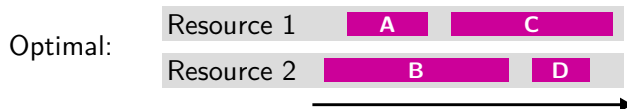


Kürzestes-Intervall (DACB)

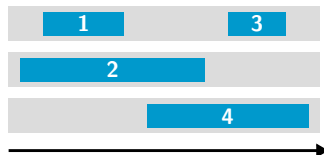


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

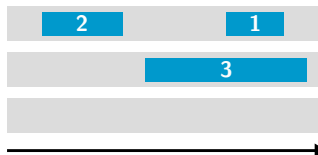
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)



Kürzestes-Intervall (DACB)

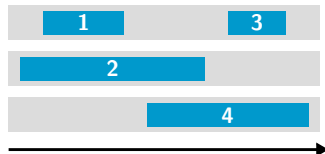


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

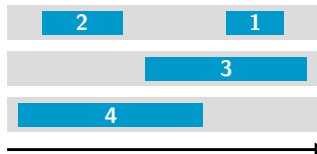
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)

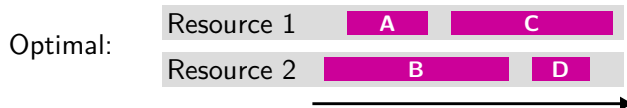


Kürzestes-Intervall (DACB)

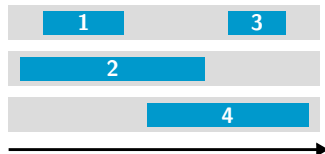


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

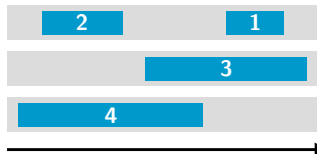
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



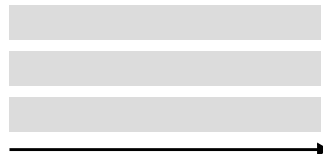
Frühstes-Ende (ABDC)



Kürzestes-Intervall (DACB)

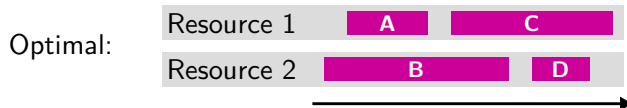


Wenigste-Konflikte (ADBC)

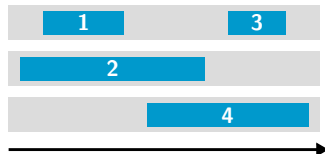


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

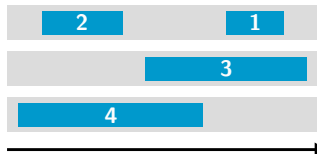
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



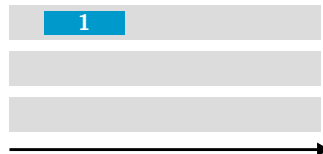
Frühstes-Ende (ABDC)



Kürzestes-Intervall (DACB)

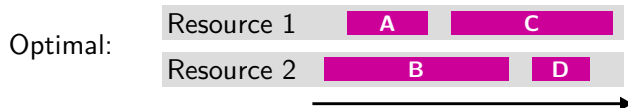


Wenigste-Konflikte (ADBC)

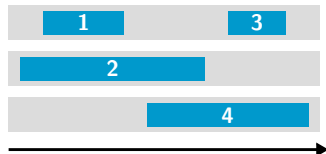


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

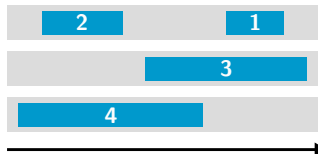
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



Frühstes-Ende (ABDC)



Kürzestes-Intervall (DACB)

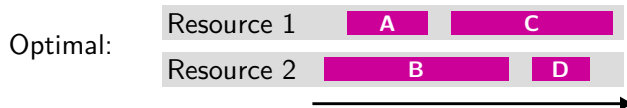


Wenigste-Konflikte (ADBC)

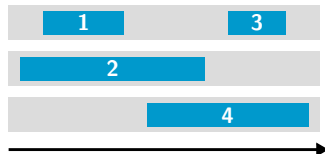


Gegenbeispiele zur Optimalität vorgeschlagener Strategien

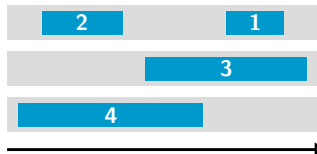
Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



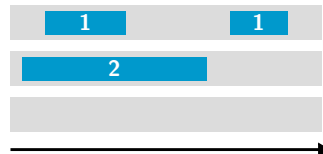
Frühstes-Ende (ABDC)



Kürzestes-Intervall (DACB)

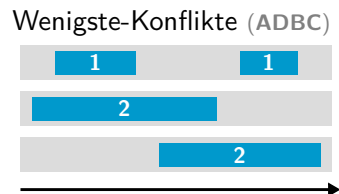
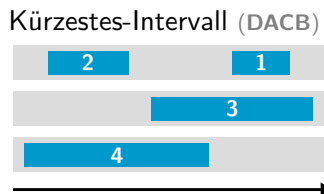
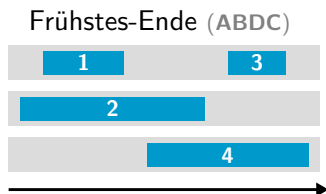
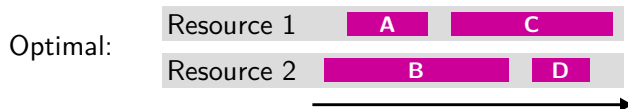


Wenigste-Konflikte (ADBC)



Gegenbeispiele zur Optimalität vorgeschlagener Strategien

Drei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:



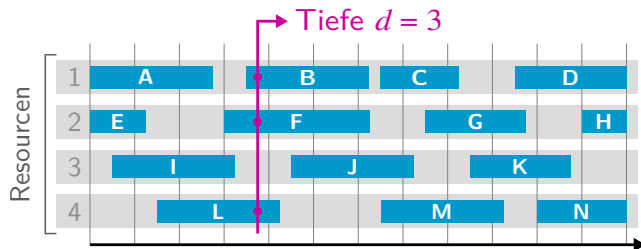
Die drei Greedy-Algorithmen benötigen 3 an Stelle der optimalen 2 Ressourcen.

Gierige Strategie "Frühester-Start"

Intervallverteilung durch Frühesten Startzeitpunkt

Durchlaufe die Intervalle I_1, \dots, 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 $S = \{I_1, \dots, I_n\}$ definieren wir die **Tiefe** (depth) $d(S)$ als die größte Anzahl von Intervallen, die einen gemeinsamen Punkt umfassen.



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, \dots, 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. \square

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, \dots, 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. \square
- ▶ 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 Fröhester-Start findet sie!

Pseudocode des Greedy-Algorithmus Fröhester-Start

```
1 // Gegeben:  $n$  Intervalle mit Start- und Endzeiten  $s_i$  und  $f_i$ 
2 sortiere Intervalle nach Anfangszeiten  $\rightarrow I_1, \dots, I_n$ 
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     $K \leftarrow K + 1$ 
11  end
12 end
```

Pseudocode des Greedy-Algorithmus Frühester-Start

```
1 // Gegeben:  $n$  Intervalle mit Start- und Endzeiten  $s_i$  und  $f_i$ 
2 sortiere Intervalle nach Anfangszeiten  $\rightarrow I_1, \dots, I_n$ 
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     $K \leftarrow K + 1$ 
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.

Optimalität des Greedy Algorithmus für *Interval Partitioning*

Der Algorithmus Fröhester-Start auf Seite 59 liefert eine optimale Intervallverteilung und kann in einer Laufzeit in $O(n \log n)$ implementiert werden.

- ▶ Die Abfrage in Zeile 5 stellt sicher, dass alle Ressourcen kompatibel sind.
- ▶ Nun zeigen wir noch, dass Fröhester-Start nicht mehr als d Ressourcen benötigt, wobei d die Tiefe der gegebenen Menge von Intervallen ist.

Optimalität des Greedy Algorithmus für *Interval Partitioning*

Der Algorithmus Fröhester-Start auf Seite 59 liefert eine optimale Intervallverteilung und kann in einer Laufzeit in $\mathcal{O}(n \log n)$ implementiert werden.

- ▶ Die Abfrage in Zeile 5 stellt sicher, dass alle Ressourcen kompatibel sind.
- ▶ Nun zeigen wir noch, dass Fröhester-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: $K + 1 \leq d$. \square
- ▶ 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.

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> earliestStart(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> ressourcen = new PriorityQueue<>();
8     while (!pq.isEmpty()) {
9         Interval iv = pq.poll();
10        Ressource res = ressourcen.peek(); // null at first iteration
11        if (res != null && iv.start() >= res.getFinish()) {
12            res.intervals.enqueue(iv);
13            ressourcen.poll(); // remove res
14            res.setFinish(iv.finish()); // update finish time
15            ressourcen.add(res); // and re-add
16        } else
17            ressourcen.add(new Ressource(iv));
18    }
19
20    return ressourcen;
21 }
```


Laufzeit des Greedy Algorithmus für *Interval Partitioning*

Der Implementation von Fröhester-Start auf Seite 62 zur optimalen Intervallverteilung hat eine Laufzeit in $\mathcal{O}(n \log n)$ für n Intervalle.

- ▶ Zur Sortierung wird jedes Intervall per `add()` in eine PQ eingefügt. Für n Intervalle ergibt dies eine Laufzeit in $\mathcal{O}(n \log n)$.

Laufzeit des Greedy Algorithmus für *Interval Partitioning*

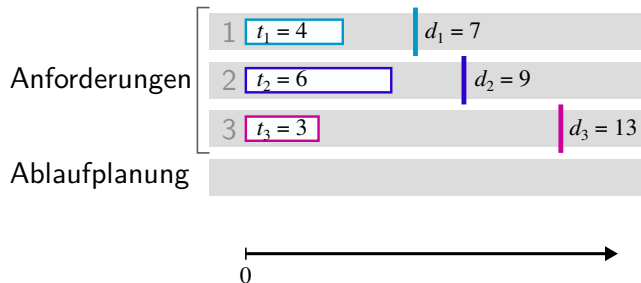
Der Implementation von Fröhester-Start auf Seite 62 zur optimalen Intervallverteilung hat eine Laufzeit in $\mathcal{O}(n \log n)$ für n Intervalle.

- ▶ Zur Sortierung wird jedes Intervall per `add()` in eine PQ eingefügt. Für n Intervalle ergibt dies eine Laufzeit in $\mathcal{O}(n \log n)$.
- ▶ Die Schleife (Zeilen 8–18) wird für jedes der n Intervalle einmal durchlaufen.
- ▶ Die verwendeten PQ Methoden `poll()` und `add()` haben eine Laufzeit in $\mathcal{O}(\log n)$, 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 $\mathcal{O}(n \log n)$.



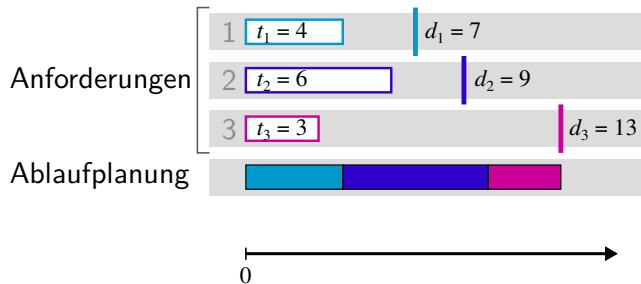
Greedy Beispiel: Planung mit minimalen Verspätungen

- ▶ 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 späterster 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.



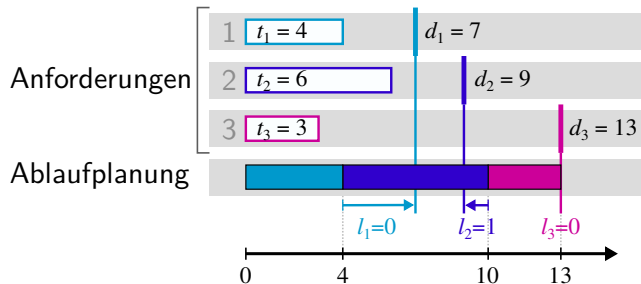
Greedy Beispiel: Planung mit minimalen Verspätungen

- ▶ 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 späterster 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.



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 späterster 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

Gierige Auswahl bei Planung mit minimalen Verspätungen

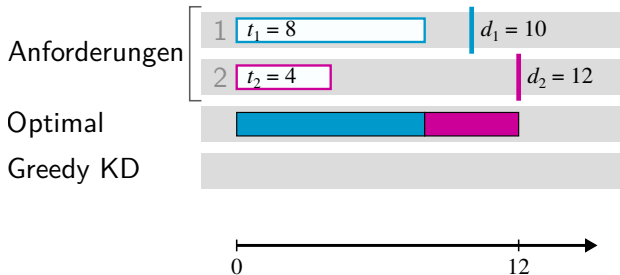
Die Anforderungen werden in einer gewissen Ordnung (siehe Liste) ohne Leerlaufzeiten arrangiert. Diesmal sind keine Start- und Endzeiten vorgegeben, sondern Dauer t_i und Deadline d_i . Daher gibt es andere Vorschläge für die Greedy-Auswahl.

- ▶ **[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?

Zwei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

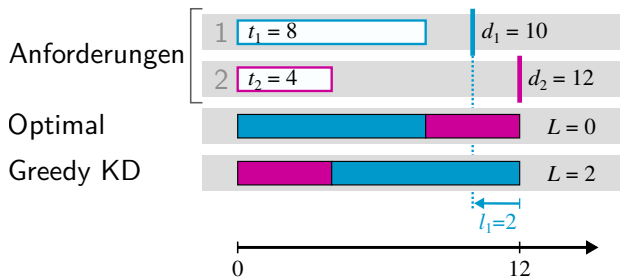
► **Kürzeste-Dauer:**



► **Kleinste-Pufferzeit:**

Zwei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

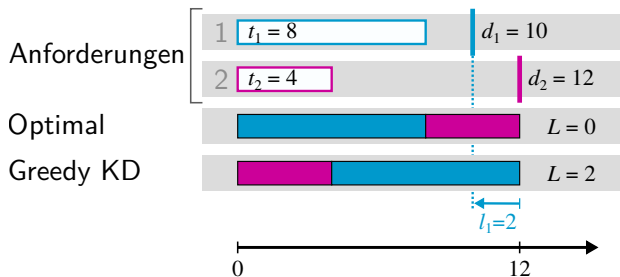
► **Kürzeste-Dauer:**



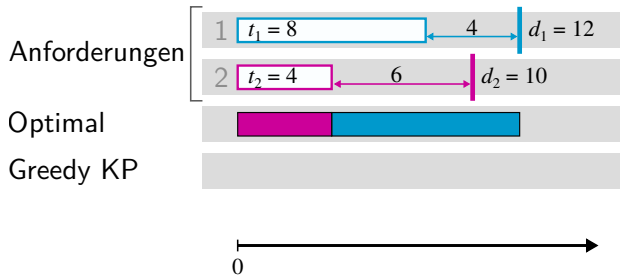
► **Kleinste-Pufferzeit:**

Zwei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

► **Kürzeste-Dauer:**

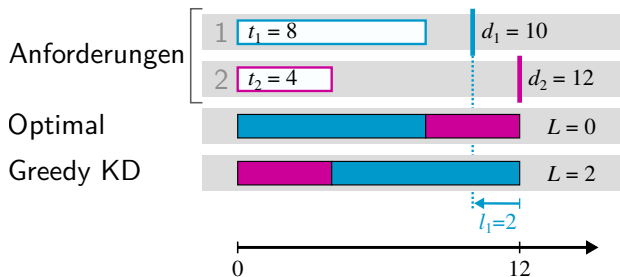


► **Kleinste-Pufferzeit:**

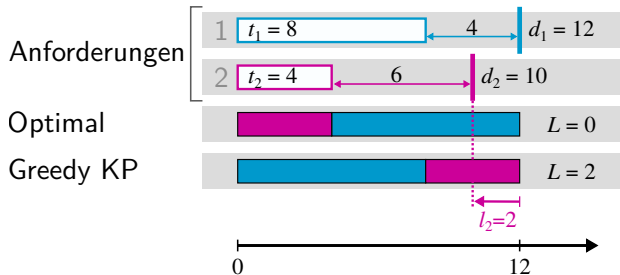


Zwei der vorgeschlagenen Strategien sind offensichtlich **nicht optimal**:

► **Kürzeste-Dauer:**



► **Kleinste-Pufferzeit:**



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.

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 von Früheste-Deadline

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

- ▶ Auf der folgenden Seite 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.

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.

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.

- ▶ Sei O ein optimaler Ablauf ohne Leerlauf (aus 1) mit Inversion.
- 2 Es gibt Aufgaben $i < j$ mit $d_j < d_i$, die im Ablauf 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 $d_j < d_i$.

- ▶ Sei O ein optimaler Ablauf ohne Leerlauf (aus 1) mit Inversion.
- 2 Es gibt Aufgaben $i < j$ mit $d_j < d_i$, die im Ablauf 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.

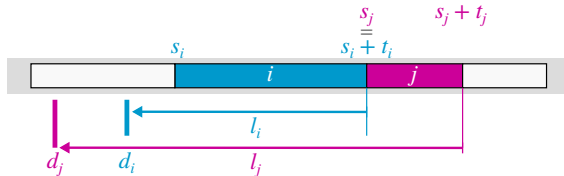
Existenzbeweis – Aussage 4

- 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.

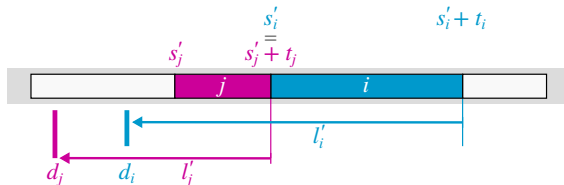
Existenzbeweis – Aussage 4

- 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 ursprünglichen 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 .

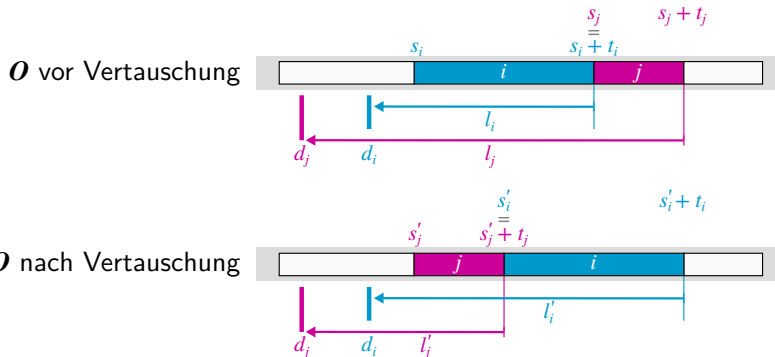
O vor Vertauschung



O nach Vertauschung



Existenzbeweis – Gleichungskette zu Aussage 4



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

Definition von Verspätung

wegen $s'_i = s'_j + t_j$

wegen $s'_j = s_i$

wegen $s_j = s_i + t_i$

wegen $d_j < d_i$

- 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. □

Äquivalenzbeweis

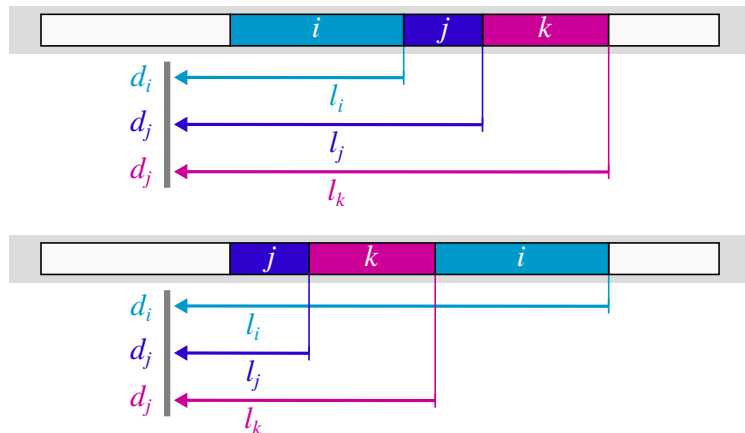
Alle Abläufe ohne Inversionen und Leerlauf haben dieselbe maximale Verspätung.

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. \square

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.



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.

- ▶ 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

Auflistung aller Permutationen
Implementation, 9

Backtracking

allgemeines Prinzip, 18

Pseudocode, 18

Backtracking, 5

Rösselsprung, 11

Baum der Teillösungen, 20

Gieriger Algorithmus, 30

Greedy Algorithm, 30

Greedy-Algorithmus, 30

Implementation

Auflistung aller

Permutationen, 9

Rösselsprung, 13–16

interval scheduling, 33

Intervallauswahl, 33

Inversion, 67

Rösselsprung, 11

Implementation, 13–16

Rucksackproblem, 44

Sudoku, 25

teilbaren Rucksackproblem, 45

Tiefe einer Intervallmenge, 57