Stadtteilschule Walddörfer

Fach: Seminar

Tutor: [Lehrername]

Bibertomograph

43. Bundeswettbewerb Informatik 2024/25

Junioraufgabe 2

Abgabedatum: [Datum]

Name: Jakob

Schule: Stadtteilschule Walddörfer

Profil: [Profilname]

Inhaltsverzeichnis

1 Aufgabe Bibertomograph

Diese Ausarbeitung beschäftigt sich mit der Aufgabe „Bibertomograph“ aus dem 43. Bundeswettbewerb Informatik (2024/25). Die Arbeit entstand im Rahmen des Seminars.

1.1 Fragestellung

Bei der Aufgabe Bibertomograph geht es um die Rekonstruktion eines Bildes aus seinen Randsummen. Das Problem ist angelehnt an medizinische Tomographen, die aus verschiedenen Projektionen ein Bild rekonstruieren.

Gegeben ist ein $n \times n$ Raster, wobei jede Zelle entweder gefüllt (`#`) oder leer (`.`) sein kann. Vom ursprünglichen Bild sind nur die Summen bekannt:

- Die **Spaltensummen**: Wie viele Zellen pro Spalte gefüllt sind
- Die **Zeilensummen**: Wie viele Zellen pro Zeile gefüllt sind
- Die **Hauptdiagonalsummen** (von links-oben nach rechts-unten): Summen entlang der $\searrow$-Diagonalen
- Die **Nebendiagonalsummen** (von links-unten nach rechts-oben): Summen entlang der $\nearrow$-Diagonalen

Die Aufgabe besteht darin, aus diesen Informationen das ursprüngliche Bild zu rekonstruieren. Dabei soll das Programm erkennen, ob es genau eine Lösung, mehrere Lösungen oder keine Lösung gibt.

1.2 Lösungsidee

Mein erster Gedanke war, das Problem als eine Art Sudoku zu betrachten. Wie beim Sudoku hat man Einschränkungen (Constraints), die erfüllt werden müssen. Der Unterschied ist, dass wir hier nur zwei mögliche Werte pro Zelle haben (0 oder 1), dafür aber vier verschiedene Richtungen berücksichtigen müssen.

Ich habe mich für einen Backtracking-Ansatz mit Heuristiken entschieden. Das bedeutet, dass ich systematisch Werte ausprobiere und bei Widersprüchen zurückgehe. Um das Ganze effizienter zu machen, verwende ich zwei Techniken:

1. **Constraint Propagation**: Wenn eine Zeile oder Diagonale schon voll ist oder noch alle verbleibenden Zellen gefüllt werden müssen, setze ich diese automatisch.
2. **Heatmap-Heuristik**: Ich berechne für jede Zelle, wie wahrscheinlich sie gefüllt ist, und wähle die Zellen clever aus.

2 Theoretische Grundlagen

Bevor ich mit der eigentlichen Implementierung beginne, möchte ich einige theoretische Konzepte erklären, die für das Verständnis meiner Lösung wichtig sind.

2.1 Constraint Satisfaction Problems

Das Bibertomograph-Problem gehört zur Klasse der *Constraint Satisfaction Problems* (CSP). Ein CSP besteht aus:

- Einer Menge von **Variablen** (hier: die n^2 Zellen des Rasters)
- Einer **Domäne** für jede Variable (hier: $\{0, 1\}$ für leer oder gefüllt)
- Einer Menge von **Constraints** (hier: die Summen für jede Zeile, Spalte und Diagonale)

Eine Lösung ist eine Belegung aller Variablen, sodass alle Constraints erfüllt sind.

2.2 Backtracking

Backtracking ist ein Algorithmus, der systematisch alle möglichen Belegungen durchprobiert. Er funktioniert rekursiv:

1. Wähle eine nicht zugewiesene Variable
2. Probiere einen Wert aus der Domäne
3. Prüfe, ob alle Constraints noch erfüllbar sind
4. Wenn ja: Gehe rekursiv zum nächsten Schritt
5. Wenn nein: Probiere den nächsten Wert (Backtrack)

Das Problem bei naivem Backtracking ist die Laufzeit. Bei einem 8×8 Raster gäbe es theoretisch 2^{64} mögliche Belegungen. Deshalb sind Optimierungen wichtig.

2.3 Constraint Propagation

Constraint Propagation ist eine Technik, um den Suchraum zu verkleinern. Die Idee ist einfach: Wenn durch die bisherigen Entscheidungen bestimmte Zellen eindeutig festgelegt sind, setze ich sie sofort.

Beispiel: Wenn in einer Zeile mit Zielsumme 3 bereits 3 Zellen gefüllt sind, müssen alle verbleibenden Zellen leer sein. Umgekehrt: Wenn die Zielsumme 3 ist und noch genau 3 Zellen unzugewiesen sind, müssen alle gefüllt werden.

2.4 Verwendete Java-Konzepte

Für die Implementierung habe ich folgende Java-Konstrukte verwendet, die über den Stoff der Stufe 1 hinausgehen:

- `ArrayList<Grid>`: Eine dynamische Liste zum Speichern der gefundenen Lösungen
- `int[][]`: Zweidimensionale Arrays für das Raster
- `System.arraycopy()`: Effizientes Kopieren von Arrays für das Backtracking
- `BufferedReader`: Zum Einlesen der Eingabedateien

3 Programmstruktur

Mein Programm besteht aus fünf Klassen, die jeweils eine bestimmte Aufgabe haben. Diese Aufteilung macht den Code übersichtlicher und leichter zu testen.

3.1 Übersicht der Klassen

Klasse	Beschreibung
`Main.java`	Einstiegspunkt des Programms. Liest die Eingabedatei, startet den Solver und gibt die Ergebnisse aus.
`Grid.java`	Repräsentiert das $n \times n$ Raster. Speichert den Zustand jeder Zelle und berechnet Summen für Zeilen, Spalten und Diagonalen.
`Constraints.java`	Speichert die Zielwerte (Randsummen) und berechnet die Heatmap für die Heuristik.
`InputParser.java`	Liest die Eingabedatei und erzeugt ein Constraints-Objekt.
`HeuristicSolver.java`	Der eigentliche Lösungsalgorithmus mit Backtracking, Constraint Propagation und Heuristik.

3.2 Main.java – Hauptprogramm

Die Main-Klasse ist der Einstiegspunkt. Sie liest die Eingabedatei, startet den Solver und gibt die Ergebnisse formatiert aus.

```
1 public static void main(String[] args) {
2     if (args.length < 1) {
3         System.out.println("Verwendung: java Main <eingabedatei.txt>")
              ;
4         System.exit(1);
5     }
6
7     String filename = args[0];
```

```
8
9       // Eingabe parsen
10      Constraints constraints = InputParser.parse(filename);
11
12      // Heatmap berechnen und anzeigen
13      double[][] heatmap = constraints.computeHeatmap();
14
15      // Solver starten
16      HeuristicSolver solver = new HeuristicSolver(constraints);
17      solver.setMaxSolutions(100);
18
19      List<Grid> solutions = solver.solve();
20
21      // Ergebnisse ausgeben
22      System.out.println("Gefundene Loesungen: " + solutions.size());
23  }
```

Listing 1: Hauptprogramm (Main.java)

3.3 Grid.java – Das Raster

Die Grid-Klasse repräsentiert das Spielfeld. Jede Zelle kann drei Zustände haben:

- `-1`: Noch nicht zugewiesen
- `0`: Leer (wird als . ausgegeben)
- `1`: Gefüllt (wird als # ausgegeben)

```
1  public class Grid {
2      private final int size;
3      private final int[][] cells;
4
5      public Grid(int size) {
6          this.size = size;
7          this.cells = new int[size][size];
8          // Initialisiere alle Zellen als nicht zugewiesen
9          for (int i = 0; i < size; i++) {
10             for (int j = 0; j < size; j++) {
11                 cells[i][j] = -1;
12             }
13         }
14     }
15
16     // Kopiert das Grid (wichtig fuer Backtracking)
17     public Grid copy() {
18         Grid copy = new Grid(size);
19         for (int i = 0; i < size; i++) {
```

```
            System.arraycopy(cells[i], 0, copy.cells[i], 0, size);
        }
        return copy;
    }
}
```

Listing 2: Grid-Konstruktor und Kopieren

Wichtig sind die Methoden zum Berechnen der Summen. Für die Diagonalen muss man etwas aufpassen mit der Indexierung:

```
// Hauptdiagonale: k = row + col, Bereich: 0 bis 2*(size-1)
public int getDiagDownSum(int k) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        int j = k - i;
        if (j >= 0 && j < size && cells[i][j] == 1) {
            sum++;
        }
    }
    return sum;
}

// Nebendiagonale: k = col - row + (size-1)
public int getDiagUpSum(int k) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        int j = k - (size - 1) + i;
        if (j >= 0 && j < size && cells[i][j] == 1) {
            sum++;
        }
    }
    return sum;
}
```

Listing 3: Berechnung der Diagonalsummen

Die Indexierung der Diagonalen war einer der schwierigsten Teile. Für die Hauptdiagonale gilt $k = \text{row} + \text{col}$, für die Nebendiagonale $k = \text{col} - \text{row} + (n-1)$.

4 Der Solver-Algorithmus

Der HeuristicSolver ist das Herzstück meines Programms. Er kombiniert Backtracking mit Constraint Propagation und einer Heatmap-Heuristik.

4.1 Die Heatmap-Heuristik

Die Heatmap berechnet für jede Zelle einen Wert zwischen 0 und 1, der angibt, wie wahrscheinlich die Zelle gefüllt ist. Dafür betrachte ich alle vier Linien, die durch die Zelle gehen, und berechne deren „Dichte“.

```
public double[][] computeHeatmap() {
    double[][] heatmap = new double[size][size];

    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            // Dichte fuer jede Linie durch diese Zelle
            double rowDensity = (double) rowSums[i] / size;
            double colDensity = (double) colSums[j] / size;

            // Hauptdiagonale: k = i + j
            int diagDownK = i + j;
            int diagDownLen = size - Math.abs(diagDownK - (size - 1));
            double diagDownDensity = (double) getDiagDownSum(diagDownK
                ) / diagDownLen;

            // Nebendiagonale: k = j - i + (size - 1)
            int diagUpK = j - i + (size - 1);
            int diagUpLen = size - Math.abs(diagUpK - (size - 1));
            double diagUpDensity = (double) getDiagUpSum(diagUpK) /
                diagUpLen;

            // Durchschnitt aller Dichten
            heatmap[i][j] = (rowDensity + colDensity + diagDownDensity
                + diagUpDensity) / 4.0;
        }
    }

    return heatmap;
}
```

Listing 4: Berechnung der Heatmap

4.2 Backtracking mit Heuristik

Der Backtracking-Algorithmus wählt immer die Zelle aus, bei der wir am „sichersten“ sind. Das bedeutet: Zellen, deren Heatmap-Wert weit von 0.5 entfernt ist (also klar gefüllt oder klar leer), werden zuerst behandelt.

```
private void backtrack(Grid grid) {
    if (solutions.size() >= maxSolutions) {
        return;
    }
```

```
5
6       nodeCount++;
7
8       // Waehle naechste Zelle basierend auf Heuristik
9       int[] nextCell = selectNextCell(grid);
10
11      // Wenn keine unzugewiesene Zelle mehr, pruefe Loesung
12      if (nextCell == null) {
13          if (isValidSolution(grid)) {
14              solutions.add(grid.copy());
15          }
16          return;
17      }
18
19      int row = nextCell[0];
20      int col = nextCell[1];
21
22      // Reihenfolge der Werte basierend auf Heatmap
23      int[] valuesToTry;
24      if (heatmap[row][col] >= 0.5) {
25          valuesToTry = new int[]{1, 0};
26      } else {
27          valuesToTry = new int[]{0, 1};
28      }
29
30      for (int value : valuesToTry) {
31          // Forward Check
32          if (!isValueFeasible(grid, row, col, value)) {
33              continue;
34          }
35
36          Grid newGrid = grid.copy();
37          newGrid.set(row, col, value);
38
39          // Constraint Propagation
40          if (propagate(newGrid)) {
41              backtrack(newGrid);
42          }
43      }
44  }
```

Listing 5: Backtracking-Hauptschleife

4.3 Constraint Propagation

Die Propagation prüft alle Zeilen, Spalten und Diagonalen. Wenn eine Linie „erzwungen“ ist, werden die Zellen automatisch gesetzt.

```java
private int propagateLine(Grid grid, int index, boolean isRow) {
    int sum = isRow ? grid.getRowSum(index) : grid.getColSum(index);
    int unassigned = isRow ? grid.getRowUnassigned(index) : grid.
        getColUnassigned(index);
    int target = isRow ? constraints.getRowSum(index) : constraints.
        getColSum(index);

    int remaining = target - sum;

    // Konflikt: Mehr gefuellt als erlaubt
    if (remaining < 0) return -1;

    // Konflikt: Nicht genug Platz
    if (remaining > unassigned) return -1;

    // Alle verbleibenden muessen 1 sein
    if (remaining == unassigned) {
        for (int k = 0; k < size; k++) {
            int i = isRow ? index : k;
            int j = isRow ? k : index;
            if (!grid.isAssigned(i, j)) {
                grid.set(i, j, 1);
            }
        }
    }

    // Summe erreicht: alle verbleibenden muessen 0 sein
    if (remaining == 0) {
        for (int k = 0; k < size; k++) {
            int i = isRow ? index : k;
            int j = isRow ? k : index;
            if (!grid.isAssigned(i, j)) {
                grid.set(i, j, 0);
            }
        }
    }

    return 0;
}
```

Listing 6: Propagation fuer eine Zeile/Spalte

4.4 Forward Checking

Forward Checking prüft, ob ein Wert überhaupt noch möglich ist, bevor wir ihn ausprobieren. Das spart viele unnötige Backtracking-Schritte.

```java
private boolean isValueFeasible(Grid grid, int row, int col, int value) {
    if (value == 1) {
        // Pruefe ob wir noch Platz fuer eine 1 haben
        if (grid.getRowSum(row) >= constraints.getRowSum(row)) return false;
        if (grid.getColSum(col) >= constraints.getColSum(col)) return false;

        int diagDownK = row + col;
        if (grid.getDiagDownSum(diagDownK) >= constraints.getDiagDownSum(diagDownK))
            return false;

        int diagUpK = col - row + (size - 1);
        if (grid.getDiagUpSum(diagUpK) >= constraints.getDiagUpSum(diagUpK))
            return false;
    } else {
        // Pruefe ob genug Platz fuer benoetigte 1en bleibt
        int rowRemaining = constraints.getRowSum(row) - grid.getRowSum(row);
        if (grid.getRowUnassigned(row) - 1 < rowRemaining) return false;

        // ... analog fuer Spalte und Diagonalen
    }

    return true;
}
```

Listing 7: Forward Checking

5 Eingabe und Ausgabe

5.1 Eingabeformat

Die Eingabedateien haben ein einfaches Format:

1. Zeile 1: Die Größe n des Rasters
2. Zeile 2: Die n Spaltensummen (von links nach rechts)
3. Zeile 3: Die n Zeilensummen (von oben nach unten)
4. Zeile 4: Die $2n - 1$ Hauptdiagonalsummen
5. Zeile 5: Die $2n - 1$ Nebendiagonalsummen

```java
public static Constraints parse(String filename) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(
        filename))) {
        // 1. Zeile: Groesse n
        int size = Integer.parseInt(reader.readLine().trim());

        // 2. Zeile: Spaltensummen
        int[] colSums = parseIntArray(reader.readLine(), size);

        // 3. Zeile: Zeilensummen
        int[] rowSums = parseIntArray(reader.readLine(), size);

        // 4. Zeile: Hauptdiagonalsummen (2*n - 1 Werte)
        int[] diagDownSums = parseIntArray(reader.readLine(), 2 * size
            - 1);

        // 5. Zeile: Nebendiagonalsummen
        int[] diagUpSums = parseIntArray(reader.readLine(), 2 * size -
            1);

        return new Constraints(size, colSums, rowSums, diagDownSums,
            diagUpSums);
    }
}
```

Listing 8: Parsen der Eingabedatei

5.2 Ausgabeformat

Das Programm gibt die Lösung als Raster aus, wobei # für gefüllte und . für leere Zellen steht. Bei mehreren Lösungen wird zusätzlich eine kombinierte Ansicht gezeigt, wo ? für mehrdeutige Zellen steht.

6 Beispiele aus dem Wettbewerb

Hier zeige ich die Ergebnisse für einige der Testdateien aus dem Wettbewerb.

6.1 tomograph00.txt (8×8)

Das erste Beispiel ist ein 8×8 Raster:

Constraints:

```
Column sums: 1 1 5 6 5 1 2 1
Row sums: 3 2 3 2 6 2 2 2
```

```
DiagDown sums: 0 0 1 3 2 3 3 1 2 3 1 2 0 1 0
DiagUp sums: 0 0 1 0 4 2 1 3 5 3 2 1 0 0 0
```

Ausgabe (1 Lösung):

```
..###...
..#.#...
..###...
#..#....
.######.
...#..#.
...#...#
..#.#...
```

Statistik: Gefundene Lösungen: 1, Durchsuchte Knoten: 6, Laufzeit: 37 ms

6.2 tomograph01.txt (2×2)

Das kleinste Beispiel:

Constraints:

```
Column sums: 1 0
Row sums: 0 1
```

Ausgabe (1 Lösung):

```
..
#.
```

Der Solver findet sofort die eindeutige Lösung mit nur 1 Knoten.

6.3 tomograph02.txt (4×4)

Constraints:

```
Column sums: 3 2 2 2
Row sums: 1 2 4 2
```

Ausgabe (1 Lösung):

```
..#.
##..
####
#..#
```

Auch hier gibt es genau eine Lösung. Die Constraint Propagation reicht aus – es wird nur 1 Knoten durchsucht.

6.4 tomograph03.txt (4×4) – Mehrere Lösungen

Dieser Testfall ist besonders interessant, weil er **mehrere gültige Lösungen** hat:

Constraints:

```
Column sums: 1 2 2 3
Row sums: 2 3 1 2
```

Kombinierte Ausgabe (? = mehrdeutig):

```
.??#
?##?
?..?
.??#
```

Lösung 1:

```
.#.#
.###
#...
..##
```

Lösung 2:

```
..##
###.
...#
.#.#
```

Die Zellen mit ? können sowohl gefüllt als auch leer sein. Das zeigt, dass die Randsummen allein nicht immer eine eindeutige Lösung bestimmen. Der Solver findet beide Lösungen in nur 5 Knoten.

7 Fazit und Ausblick

7.1 Was funktioniert gut?

Mein Solver löst alle Testfälle des Wettbewerbs korrekt und effizient:

- Die Kombination aus Constraint Propagation und Heatmap-Heuristik reduziert die Anzahl der durchsuchten Knoten erheblich.
- Selbst bei größeren Rastern (8×8) ist die Laufzeit sehr kurz (unter 100 ms).
- Das Programm erkennt korrekt, wenn es mehrere Lösungen gibt.

7.2 Was war schwierig?

Die größten Herausforderungen waren:

1. **Diagonalen-Indexierung**: Die korrekte Berechnung der Indizes für die Diagonalen ($k = \text{row} + \text{col}$ und $k = \text{col} - \text{row} + (n-1)$) hat am meisten Zeit gekostet.
2. **Debugging**: Bei Backtracking-Algorithmen ist es schwer zu sehen, wo genau ein Fehler liegt. Ich habe viel mit Ausgaben gearbeitet.
3. **Kopieren von Zuständen**: Beim Backtracking muss man aufpassen, dass man das Grid richtig kopiert, sonst ändert man versehentlich den ursprünglichen Zustand.

7.3 Mögliche Verbesserungen

- **Arc Consistency**: Eine stärkere Form der Propagation, die noch mehr Suchraum eliminieren könnte.
- **Parallelisierung**: Für sehr große Raster könnte man verschiedene Teilbäume parallel durchsuchen.
- **GUI**: Eine grafische Oberfläche würde die Ergebnisse anschaulicher machen.

8 Quellcode

Im Folgenden sind die wichtigsten Teile des Quellcodes dokumentiert.

8.1 Constraints.java – Vollständig

```
1  public class Constraints {
2      private final int size;
3      private final int[] colSums;
4      private final int[] rowSums;
5      private final int[] diagDownSums;
6      private final int[] diagUpSums;
7
8      public Constraints(int size, int[] colSums, int[] rowSums,
9                         int[] diagDownSums, int[] diagUpSums) {
10         this.size = size;
11         this.colSums = colSums;
12         this.rowSums = rowSums;
13         this.diagDownSums = diagDownSums;
14         this.diagUpSums = diagUpSums;
15     }
16
17     public int getSize() { return size; }
```

```
18      public int getColSum(int col) { return colSums[col]; }
19      public int getRowSum(int row) { return rowSums[row]; }
20      public int getDiagDownSum(int k) { return diagDownSums[k]; }
21      public int getDiagUpSum(int k) { return diagUpSums[k]; }
22  }
```

Listing 9: Constraints.java

8.2 Grid.java – toString Methode

```
1  @Override
2  public String toString() {
3      StringBuilder sb = new StringBuilder();
4      for (int i = 0; i < size; i++) {
5          for (int j = 0; j < size; j++) {
6              if (cells[i][j] == 1) {
7                  sb.append('#');
8              } else if (cells[i][j] == 0) {
9                  sb.append('.');
10             } else {
11                 sb.append('?');
12             }
13         }
14         sb.append('\n');
15     }
16     return sb.toString();
17 }
```

Listing 10: Ausgabe des Rasters

9 Quellenverzeichnis

1. Bundeswettbewerb Informatik: Aufgabenblatt 44. Wettbewerb 2025/26, Aufgabe: Bibertomograph
   https://bwinf.de/bundeswettbewerb/44/#c5529
   (letzter Abruf: 7.12.2025)

2. Oracle: Java SE Documentation – ArrayList
   https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
   (letzter Abruf: 6.12.2025)

3. YouTube: Backtracking Algorithm Tutorial
   https://www.youtube.com/watch?v=m-b51C82-UE
   (letzter Abruf: 2.12.2025)

4. ResearchGate: Heuristic vs Brute-Force Comparison
https://www.researchgate.net/figure/Heuristic-brute-force-comparison_tbl1_24017925
(letzter Abruf: 28.11.2025)

Eidesstattliche Erklärung

Ich versichere, dass die vorliegende Seminararbeit mit dem Titel „Seminar-Aufgabe: 43. BwInf Bibertomograph“ von mir selbstständig erarbeitet wurde und ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Diejenigen Teile der Arbeit, die anderen Werken im Wortlaut oder dem Sinn nach entnommen wurden, sind als solche kenntlich gemacht.

_____

Ort, Datum, Unterschrift