Stadtteilschule Walddörfer

Fach: Seminar

Tutor: Frau Dr. Schmidt

Bibertomograph

44. Bundeswettbewerb Informatik 2025/26 – 1. Runde

Aufgabe 4

Leitfrage: Wie können aus Sichtlinien-Summen die ursprünglichen Figuren rekonstruiert werden?

Abgabedatum:	8.12.2025
Name:	Jakob Allweiss
Profil:	Zukunftstechnologien

Inhaltsverzeichnis

1 Aufgabe Bibertomograph

Diese Ausarbeitung beschäftigt sich mit der Aufgabe „Bibertomograph“ aus dem 44. Bundeswettbewerb Informatik (2025/26) und entstand im Rahmen des Seminars im Zeitraum vom 6.10.2025 bis zum 8.12.2025.

1.1 Fragestellung

Die originale Aufgabenstellung des Wettbewerbs lautet:

> „Bei den jährlichen Biberland-Games werden in einer Disziplin aus der Luft sichtbare Figuren auf einer gerasterten Fläche gestaltet. Ein Rasterfeld kann dabei entweder mit Holzspänen gefüllt sein oder nicht. Die Fläche wird vor der Jurywertung den Blicken der neugierigen Biberazzi durch einen Bretterzaun entzogen. Biberazzo Azze hat nun festgestellt, dass er durch kleine Lücken im Zaun senkrecht und diagonal über die Fläche schauen kann und aufgrund der Helligkeit erahnt, wie viele gefüllte Felder auf seiner jeweiligen Sichtlinie liegen. Azze will unbedingt möglichst viele Figuren bereits vorher herausfinden und versucht, durch logische Schlussfolgerungen und Ausprobieren die korrekten Figuren zu rekonstruieren. [...] Hilf Azze mit einem Programm, das für eine beliebige $n \times n$-Fläche die Summen einliest und daraufhin eine dazu passende Figur ausgibt. Falls sie nicht die einzig mögliche ist, sollte das Programm in der Figur die nicht eindeutigen Felder durch Fragezeichen kennzeichnen.“ [1]

Bei der Aufgabe geht es also um die Rekonstruktion einer Figur aus ihren Randsummen – wie viele gefüllte Felder auf jeder horizontalen, vertikalen und diagonalen Sichtlinie liegen.

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:

- `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 und koordiniert den gesamten Ablauf:

1. Liest den Dateinamen aus den Kommandozeilenargumenten
2. Parst die Eingabedatei mit `InputParser.parse()`
3. Berechnet die Heatmap für die Heuristik
4. Erstellt einen `HeuristicSolver` und sammelt bis zu 100 Lösungen
5. Gibt die Ergebnisse formatiert aus (Heatmap, Lösungen, Statistiken)

3.3 Grid.java – Das Raster

Die Grid-Klasse repräsentiert das Spielfeld als 2D-Array. Jede Zelle kann drei Zustände haben:

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

Die Klasse bietet Methoden zum Setzen/Abfragen von Zellen und zum Berechnen der aktuellen Summen. Besonders wichtig ist die `copy()`-Methode für das Backtracking, da wir bei Fehlversuchen zum vorherigen Zustand zurückkehren müssen.

Diagonalen-Indexierung: Für ein $n \times n$-Raster gibt es $2n - 1$ Haupt- und Nebendiagonalen. Die Indexierung war einer der schwierigsten Teile:

- Hauptdiagonale ($\searrow$): $k = \text{row} + \text{col}$, Bereich: 0 bis $2n - 2$
- Nebendiagonale ($\nearrow$): $k = \text{col} - \text{row} + (n - 1)$

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. Die Berechnung basiert auf der **Dichte** der vier Linien durch jede Zelle:

$$\text{Dichte einer Linie} = \frac{\text{Zielsumme}}{\text{Länge der Linie}}$$

Der Heatmap-Wert einer Zelle ist der Durchschnitt der vier Dichten (Zeile, Spalte, beide Diagonalen). Ein hoher Wert (nahe 1) bedeutet, dass die Zelle wahrscheinlich gefüllt ist; ein niedriger Wert (nahe 0) bedeutet wahrscheinlich leer. Werte nahe 0.5 sind unsicher.

```
1 // Durchschnitt der vier Liniendichten
2 heatmap[i][j] = (rowDensity + colDensity +
3                  diagDownDensity + diagUpDensity) / 4.0;
```

Listing 1: Kernidee der Heatmap

4.2 Backtracking mit Heuristik

Der Backtracking-Algorithmus wählt immer die Zelle aus, bei der wir am „sichersten“ sind – also Zellen, deren Heatmap-Wert weit von 0.5 entfernt ist.

Ablauf:

1. **Zellauswahl:** Wähle die unzugewiesene Zelle mit dem extremsten Heatmap-Wert
2. **Wertreihenfolge:** Bei Heatmap ≥ 0.5 probiere zuerst 1, sonst zuerst 0
3. **Forward Check:** Prüfe, ob der Wert überhaupt noch möglich ist
4. **Setzen & Propagieren:** Setze den Wert und führe Constraint Propagation durch
5. **Rekursion:** Gehe zur nächsten Zelle oder speichere die Lösung
6. **Backtrack:** Bei Konflikt probiere den anderen Wert oder gehe zurück

```
1 private void backtrack(Grid grid) {
2     if (solutions.size() >= maxSolutions) return;
3     nodeCount++;
4
5     // Waehle naechste Zelle basierend auf Heuristik
6     int[] nextCell = selectNextCell(grid);
```

```
7
8       // Alle Zellen zugewiesen? Pruefe Loesung
9       if (nextCell == null) {
10          if (isValidSolution(grid)) solutions.add(grid.copy());
11          return;
12      }
13
14      int row = nextCell[0], col = nextCell[1];
15
16      // Reihenfolge basierend auf Heatmap
17      int[] valuesToTry = (heatmap[row][col] >= 0.5)
18          ? new int[]{1, 0} : new int[]{0, 1};
19
20      for (int value : valuesToTry) {
21          if (!isValueFeasible(grid, row, col, value)) continue;
22          Grid newGrid = grid.copy();
23          newGrid.set(row, col, value);
24          if (propagate(newGrid)) backtrack(newGrid);
25      }
26  }
```

Listing 2: Backtracking-Algorithmus

4.3 Constraint Propagation

Die Propagation prüft nach jedem gesetzten Wert alle Zeilen, Spalten und Diagonalen auf zwei wichtige Fälle:

- **Konflikt erkennen:** Wenn bereits mehr Zellen gefüllt sind als die Zielsumme erlaubt, oder wenn nicht genug leere Zellen übrig sind, liegt ein Widerspruch vor → Backtrack.
- **Erzwungene Werte setzen:**
  - Wenn remaining = unassigned: Alle verbleibenden Zellen müssen 1 sein
  - Wenn remaining = 0: Alle verbleibenden Zellen müssen 0 sein

Durch das automatische Setzen erzwungener Werte können oft ganze Bereiche ohne Backtracking gelöst werden. Die Propagation wird wiederholt, bis keine neuen Werte mehr gesetzt werden.

4.4 Forward Checking

Forward Checking prüft *vor* dem Setzen eines Wertes, ob dieser überhaupt noch möglich ist. Das spart viele unnötige Rekursionen:

- **Für value = 1:** Prüfe, ob in allen vier Linien noch Platz für eine weitere gefüllte Zelle ist (aktuelle Summe < Zielsumme).
- **Für value = 0:** Prüfe, ob nach dem Setzen auf 0 noch genügend leere Zellen übrig bleiben, um die benötigte Anzahl gefüllter Zellen zu erreichen.

Durch diese Vorprüfung werden offensichtlich ungültige Werte sofort verworfen, ohne dass erst ein Widerspruch durch Propagation gefunden werden muss.

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

Die `InputParser`-Klasse liest diese Werte zeilenweise ein und erzeugt ein `Constraints`-Objekt.

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.

5.3 Ausführung

Um das Programm auszuführen, benötigt man Java (mindestens Version 11). Alle Befehle müssen aus dem Hauptverzeichnis des Projekts ausgeführt werden (dort wo sich die Ordner `src/` und `testdata/` befinden).

Der einfachste Aufruf erfolgt über:

```
java src/Main.java testdata/tomograph00.txt
```

Alternativ kann man zuerst kompilieren und dann ausführen:

```
javac src/*.java
java -cp src Main testdata/tomograph00.txt
```

Das Programm gibt dann die Heatmap, die gefundenen Lösungen und Statistiken aus.

6 Beispiele aus dem Wettbewerb

Hier zeige ich die Ergebnisse für drei Testdateien aus dem Wettbewerb nebeneinander:

tomograph01 (2×2) **Eindeutig**	**tomograph00 (8×8)** **Eindeutig**	**tomograph03 (4×4)** **Mehrdeutig**
.. #.	..###... ..#.#... ..###... #..#.... .######. ...#..#. ...#...# ..#.#...	.??# ?##? ?..? .??#
1 Lösung, 1 Knoten	1 Lösung, 6 Knoten	2 Lösungen, 5 Knoten

6.1 Erläuterung der Beispiele

tomograph01 ist das kleinste Beispiel (2×2). Die Constraints sind so restriktiv, dass der Solver die Lösung sofort durch Constraint Propagation findet – ohne jegliches Backtracking.

tomograph00 zeigt ein größeres 8×8 Raster. Trotz der Größe findet der Solver die eindeutige Lösung in nur 6 Knoten, weil die Heuristik die richtigen Zellen zuerst wählt.

tomograph03 ist besonders interessant: Es gibt **zwei gültige Lösungen**. Die Zellen mit `?` können sowohl gefüllt als auch leer sein. Das zeigt, dass die Randsummen allein nicht immer eine eindeutige Lösung bestimmen.

Die beiden Lösungen für tomograph03 sind:

Lösung 1	**Lösung 2**
.#.#	..##
.###	###.
#...	...#
..##	.#.#

7 Laufzeit und Performance

7.1 Theoretische Komplexität

Die theoretische Laufzeitkomplexität des Bibertomograph-Problems lässt sich wie folgt analysieren:

Worst-Case (naives Backtracking): Bei einem $n \times n$ Raster hat jede Zelle zwei mögliche Werte (0 oder 1). Ohne Optimierungen müsste man im schlimmsten Fall alle 2^{n^2} Kombinationen durchprobieren. Das ergibt eine Komplexität von:

$$O(2^{n^2})$$

Für ein 8×8 Raster wären das $2^{64} \approx 1.8 \cdot 10^{19}$ Möglichkeiten – praktisch unlösbar.

Mit Constraint Propagation: Durch das frühzeitige Erkennen von Widersprüchen und das automatische Setzen von erzwungenen Zellen wird der Suchbaum erheblich beschnitten. Die tatsächliche Komplexität hängt stark von den Constraints ab:

- Bei stark eingeschränkten Problemen (viele Nullen oder volle Zeilen) kann die Lösung oft in $O(n^2)$ gefunden werden, weil die Propagation alles löst.
- Bei weniger eingeschränkten Problemen liegt die praktische Komplexität typischerweise bei $O(k \cdot n^2)$, wobei k die Anzahl der Backtracking-Schritte ist.

7.2 Praktische Messungen

In der Praxis zeigt sich, dass mein Solver sehr effizient arbeitet:

Testfall	**Größe**	**Knoten**	**Zeit**
tomograph00	8×8	6	36 ms
tomograph01	2×2	1	33 ms
tomograph02	4×4	1	44 ms
tomograph03	4×4	5	34 ms

Die geringe Anzahl der durchsuchten Knoten zeigt, dass die Heuristiken gut funktionieren. Die Laufzeit liegt bei allen Testfällen bei etwa 30-45 ms, wobei der Großteil davon für das Laden und die JVM-Initialisierung verwendet wird. Die eigentliche Lösung wird in unter 1 ms gefunden.

7.3 Warum ist der Solver so schnell?

Drei Faktoren machen den Solver effizient:

1. **Frühe Erkennung von Sackgassen:** Forward Checking prüft vor jedem Setzen, ob der Wert überhaupt noch möglich ist. Dadurch werden viele unnötige Rekursionen vermieden.
2. **Automatisches Füllen:** Wenn eine Zeile/Spalte/Diagonale nur noch eine Möglichkeit hat, wird sie sofort gesetzt. Das reduziert die Anzahl der Entscheidungspunkte.
3. **Kluge Reihenfolge:** Die Heatmap-Heuristik wählt zuerst die Zellen, bei denen wir am sichersten sind. Dadurch werden Fehler früh erkannt.

8 Fazit und Ausblick

8.1 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.

8.2 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.

8.3 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.

9 Arbeitsprozessbericht

Datum	**Geplant**	**Erreicht**	**Probleme**	**Lösung**
Mo 06.10.2025	Konzepte Umsetzung, erste Experimente in Java	Lösungswege überlegt (Voxel-Projection, Brute-Force)	-	-

Datum	Geplant	Erreicht	Probleme	Lösung
Mo 13.10.2025	Mit Programmieren anfangen	Grid-Klasse angefangen	-	-
Mo 03.11.2025	Weiter am Code arbeiten	Eingabe-Parser fertig, Constraints-Klasse erstellt	Diagonalen verwirrend	Erstmal ignoriert
Mo 10.11.2025	Solver schreiben	Backtracking-Ansatz implementiert	Funktioniert nicht bei allen Testfällen	-
Mo 17.11.2025	Bugs fixen	Diagonalen-Bug gefunden und gefixt, läuft jetzt	-	-
Mo 24.11.2025	Dokumentation anfangen	LaTeX-Vorlage erstellt	pdflatex ging nicht	Pakete installiert
Mo 01.12.2025	Arbeit fertig schreiben	Text geschrieben, Beispiele eingefügt	-	-
Mo 08.12.2025	Nochmal drüberlesen	Kleinigkeiten korrigiert	-	-

10 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. TH Köln: Backtracking-Algorithmus
   https://www.gm.th-koeln.de/~hk/lehre/ala/ws0506/Praktikum/Projekt/C_blau/Backtracking_final.pdf
   (letzter Abruf: 5.11.2025)

4. GeeksforGeeks: Constraint Satisfaction Problems (CSP) in Artificial Intelligence
   https://www.geeksforgeeks.org/artificial-intelligence/constraint-satisfaction-pro
   (letzter Abruf: 4.12.2025)

5. YouTube: Voxeltracing für Heatmap
   https://www.youtube.com/watch?v=m-b51C82-UE

(letzter Abruf: 2.12.2025)

6. ResearchGate: Heuristic vs Brute-Force Comparison
   https://www.researchgate.net/figure/Heuristic-brute-force-comparison-The-results-tbl1_24017925
   (letzter Abruf: 28.11.2025)

Verwendete Werkzeuge: VSCode mit Java Extension Pack, GitHub, GitHub Copilot

Repository: https://github.com/Stayroh/BwInf_Seminararbeit

Eidesstattliche Erklärung

Ich versichere, dass die Präsentation von mir selbstständig erarbeitet wurde und ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Diejenigen Teile der Präsentation, die anderen Werken im Wortlaut oder dem Sinn nach entnommen wurden, sind als solche kenntlich gemacht.

Ort, Datum, Unterschrift