

Aufgabe 3

Sudoku

P-II SS 2006

Maximilian Störzer, Daniel Wasserrab, Dennis Giffhorn

LS Softwaresysteme

22. Mai 2006



Was ist ein Sudoku?

- Ein 9x9 Brett, aufgeteilt in 9 Blöcke
- In jeder Zeile, Spalte, Block darf jede Zahl von 1 bis 9 genau *einmal* vorkommen

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



Was ist ein Sudoku?

- Ein 9x9 Brett, aufgeteilt in 9 Blöcke
- In jeder Zeile, Spalte, Block darf jede Zahl von 1 bis 9 genau *einmal* vorkommen

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



Was ist ein Sudoku?

- Ein 9x9 Brett, aufgeteilt in 9 Blöcke
- In jeder Zeile, Spalte, Block darf jede Zahl von 1 bis 9 genau *einmal* vorkommen

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



Was ist ein Sudoku?

- Ein 9x9 Brett, aufgeteilt in 9 Blöcke
- In jeder Zeile, Spalte, Block darf jede Zahl von 1 bis 9 genau *einmal* vorkommen

Einfache Aufgabenstellung,
jedoch Lösung kann kompliziert werden
(NP-hart)

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



Was ist ein Sudoku?

- Ein 9x9 Brett, aufgeteilt in 9 Blöcke
- In jeder Zeile, Spalte, Block darf jede Zahl von 1 bis 9 genau *einmal* vorkommen

Einfache Aufgabenstellung,
jedoch Lösung kann kompliziert werden
(NP-hart)

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



Wie löst man ein Sudoku?

- Anwenden von logischen Regeln \Rightarrow *Constraint Solver*
- Raten \Rightarrow *Backtracking*



Wie löst man ein Sudoku?

- Anwenden von logischen Regeln \Rightarrow *Constraint Solver*
- Raten \Rightarrow *Backtracking*



Motivation

Wie löst man ein Sudoku?

- Anwenden von logischen Regeln \Rightarrow *Constraint Solver*
- Raten \Rightarrow *Backtracking*



Wie löst man ein Sudoku?

- Anwenden von logischen Regeln \Rightarrow *Constraint Solver*
- Raten \Rightarrow *Backtracking*

Manche Sudokus können rein logisch gelöst werden,
bei anderen muss geraten werden

Je nach Anfangsbelegung können Sudokus auch mehrdeutig oder unlösbar sein



Implementierung: Zustand

Wir verwenden für die Implementierung ein 'intelligentes' Board:
wir speichern für jedes Feld, welche Zahlen dort noch möglich sind.

Dazu Bit-Repräsentation: wenn Zahl i noch möglich, i -tes Bit gesetzt.
Beispiel: 001011001 heisst, 3,5,6 und 9 hier noch möglich.



Implementierung: Zustand

Zwei Implementierungsvarianten:

- *BitSets*

Vorteil: bequemes Handling

Nachteil: grosse Datenstruktur

- *shorts*

Vorteil: sehr kleine Datenstruktur

Nachteil: etwas unhandlich durch logische Bitoperationen



Implementierung: Zustand

Zwei Implementierungsvarianten:

- *BitSets*

Vorteil: bequemes Handling

Nachteil: grosse Datenstruktur

- *shorts*

Vorteil: sehr kleine Datenstruktur

Nachteil: etwas unhandlich durch logische Bitoperationen



Implementierung: Zustand

Zwei Implementierungsvarianten:

- *BitSets*
Vorteil: bequemes Handling
Nachteil: grosse Datenstruktur
- *shorts*
Vorteil: sehr kleine Datenstruktur
Nachteil: etwas unhandlich durch logische Bitoperationen



Implementierung: Zustand

Zwei Implementierungsvarianten:

- *BitSets*
Vorteil: bequemes Handling
Nachteil: grosse Datenstruktur
- *shorts*
Vorteil: sehr kleine Datenstruktur
Nachteil: etwas unhandlich durch logische Bitoperationen

Beispiel: Setzen des i -ten Bits in Feld *field*:

BitSet: `field.set(i);`

short: `field &= (1 << i);`



Implementierung: Zustand

Wie merkt man, dass in einem Feld eine Zahl explizit gesetzt wurde?

⇒ zweite Datenstruktur: Array mit 81 `boolean` Einträgen



Implementierung: Zustand

Wie merkt man, dass in einem Feld eine Zahl explizit gesetzt wurde?

⇒ zweite Datenstruktur: Array mit 81 `boolean` Einträgen

Zustandsimplementation abgeleitet von einem Interface `State`:

- *nur über dieses Interface* kommunizieren Algorithmen mit Zustand
- ermöglicht Erweiterbarkeit (benötigt für GUI-Aufgabe)



Implementierung: Exceptions

Was passiert, wenn Operation zu unlösbarem Brett führt?

Beispiele:

- Entfernen der letzten Möglichkeit
- Setzen einer Zahl, die nicht unter den möglichen ist



Implementierung: Exceptions

Was passiert, wenn Operation zu unlösbarem Brett führt?

Beispiele:

- Entfernen der letzten Möglichkeit
- Setzen einer Zahl, die nicht unter den möglichen ist

⇒ Werfen einer `NotSolvableException`



Implementierung: Exceptions

Was passiert, wenn Operation zu unlösbarem Brett führt?

Beispiele:

- Entfernen der letzten Möglichkeit
- Setzen einer Zahl, die nicht unter den möglichen ist

⇒ Werfen einer `NotSolvableException`

Einzige selbstdefinierte Exception!



Implementierung: Constraint Solver

- Constraint Solver verändern Zustand nach bestimmten logischen Regeln
- lösbarer Zustand wird garantiert in lösbaren Zustand überführt
- zugrundeliegender Algorithmus kann oftmals unabhängig von Struktur (Zeile, Spalte, Block) angewandt werden
- werden solange ausgeführt, bis keine Zustandsänderung mehr (Fixpunkt)
- implementieren Interface *ConstraintSolver*, das Programm kommuniziert damit, abstrahiert von der Implementierung



Implementierung: Constraint Solver

Zwei Constraint Solver Pflicht:

“**erzwungenes Feld**”: wenn nur noch eine Zahl in einem Feld möglich, entsprechende Zahl setzen

“**erzwungene Zahl**”: wenn in einer Zeile, Spalte oder Block eine Zahl nur in einem Feld möglich, entsprechende Zahl setzen

Bei Interesse können weitere Constraint Solver implementiert werden



Implementierung: Backtracking

Falls Anwendung von Constraint Solvern zu keiner Lösung führt, muss man “Raten”, d.h. Backtracking (oft auch “Versuch-und-Irrtum”):

- wähle ein Feld mit mehreren Möglichkeiten
 - erzeuge für jede Möglichkeit neuen Zustand (Klonen!)
- für jeden Zustand
 - Constraint Solving
 - evtl. rekursiver Backtrack-Schritt
 - falls Lösung gefunden, merken und nächsten Zustand wählen
 - falls nicht lösbar, nächsten Zustand wählen



Implementierung: Backtracking

Falls Anwendung von Constraint Solvern zu keiner Lösung führt, muss man “Raten”, d.h. Backtracking (oft auch “Versuch-und-Irrtum”):

- wähle ein Feld mit mehreren Möglichkeiten
 - erzeuge für jede Möglichkeit neuen Zustand (Klonen!)
- für jeden Zustand
 - Constraint Solving
 - evtl. rekursiver Backtrack-Schritt
 - falls Lösung gefunden, merken und nächsten Zustand wählen
 - falls nicht lösbar, nächsten Zustand wählen



Implementierung: Backtracking

Falls Anwendung von Constraint Solvern zu keiner Lösung führt, muss man “Raten”, d.h. Backtracking (oft auch “Versuch-und-Irrtum”):

- wähle ein Feld mit mehreren Möglichkeiten
 - erzeuge für jede Möglichkeit neuen Zustand (Klonen!)
- für jeden Zustand
 - Constraint Solving
 - evtl. rekursiver Backtrack-Schritt
 - falls Lösung gefunden, merken und nächsten Zustand wählen
 - falls nicht lösbar, nächsten Zustand wählen



Implementierung: Backtracking

Falls Anwendung von Constraint Solvern zu keiner Lösung führt, muss man “Raten”, d.h. Backtracking (oft auch “Versuch-und-Irrtum”):

- wähle ein Feld mit mehreren Möglichkeiten
 - erzeuge für jede Möglichkeit neuen Zustand (Klonen!)
- für jeden Zustand
 - Constraint Solving
 - evtl. rekursiver Backtrack-Schritt
 - falls Lösung gefunden, merken und nächsten Zustand wählen
 - falls nicht lösbar, nächsten Zustand wählen



Implementierung: Backtracking

Falls Anwendung von Constraint Solvern zu keiner Lösung führt, muss man “Raten”, d.h. Backtracking (oft auch “Versuch-und-Irrtum”):

- wähle ein Feld mit mehreren Möglichkeiten
 - erzeuge für jede Möglichkeit neuen Zustand (Klonen!)
- für jeden Zustand
 - Constraint Solving
 - evtl. rekursiver Backtrack-Schritt
 - falls Lösung gefunden, merken und nächsten Zustand wählen
 - falls nicht lösbar, nächsten Zustand wählen



Implementierung: Backtracking

Falls Anwendung von Constraint Solvern zu keiner Lösung führt, muss man “Raten”, d.h. Backtracking (oft auch “Versuch-und-Irrtum”):

- wähle ein Feld mit mehreren Möglichkeiten
 - erzeuge für jede Möglichkeit neuen Zustand (Klonen!)
- für jeden Zustand
 - Constraint Solving
 - evtl. rekursiver Backtrack-Schritt
 - falls Lösung gefunden, merken und nächsten Zustand wählen
 - falls nicht lösbar, nächsten Zustand wählen



Implementierung: Backtracking

Falls Anwendung von Constraint Solvern zu keiner Lösung führt, muss man “Raten”, d.h. Backtracking (oft auch “Versuch-und-Irrtum”):

- wähle ein Feld mit mehreren Möglichkeiten
 - erzeuge für jede Möglichkeit neuen Zustand (Klonen!)
- für jeden Zustand
 - Constraint Solving
 - evtl. rekursiver Backtrack-Schritt
 - falls Lösung gefunden, merken und nächsten Zustand wählen
 - falls nicht lösbar, nächsten Zustand wählen



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[14] = 010000000$
$\text{state}[14] = 000010000$
$\text{state}[14] = 000001000$



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[14] = 000010000$

$\text{state}[14] = 000001000$

$\text{state}[14] = 010000000$

$\text{state}[63] = 100000001$



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

$\text{state}[63] = 100000000$

$\text{state}[63] = 000000001$

$\text{state}[14] = 000010000$

$\text{state}[14] = 000001000$

Lösungen:



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[63] = 000000001$

$\text{state}[14] = 000010000$

$\text{state}[14] = 000001000$

$\text{state}[63] = 100000000$

unlösbar



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[14] = 000010000$

$\text{state}[14] = 000001000$

$\text{state}[63] = 000000001$

$\text{state}[63] = 000000001$

Lösung!



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[14] = 000001000$

$\text{state}[63] = 000000001$

$\text{state}[14] = 000010000$

unlösbar



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[63] = 000000001$

$\text{state}[14] = 000001000$

$\text{state}[42] = 001001000$



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[42] = 001000000$

$\text{state}[42] = 000001000$

$\text{state}[63] = 000000001$



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[42] = 000001000$

$\text{state}[42] = 001000000$

$\text{state}[63] = 000000001$

$\text{state}[42] = 001000000$

Lösung!



Implementierung: Backtracking

Backtracking-Beispiel: rate bei Feld $\text{state}[14] = 010011000$

Stack:

Lösungen:

$\text{state}[42] = 001000000$

$\text{state}[63] = 000000001$

$\text{state}[42] = 000001000$

unlösbar



Implementierung: Lösungsmethode

Zusammenfassung der Methode `solve(State s)`:

```
while(Zustand hat sich geaendert){  
    wende Constraint Solver an;  
}  
if(aktueller Zustand Loesung) return (aktueller Zustand);  
else Backtrack-Schritt, erhaltene Zustaende auf Stack;  
while(Stack nicht leer){  
    nehme ersten Zustand aus Stack;  
    try{  
        solve(entnommener Zustand);  
        gefundene Loesungen merken;  
    } catch (passende Exception) {  
        // aktueller Zustand unloesbar  
    }  
}  
return (gefundene Loesungen);
```



Fragen?

Fragen?

