Informatik I: Einführung in die Programmierung

28. Constraint Satisfaction, Backtracking und Constraint Propagierung

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

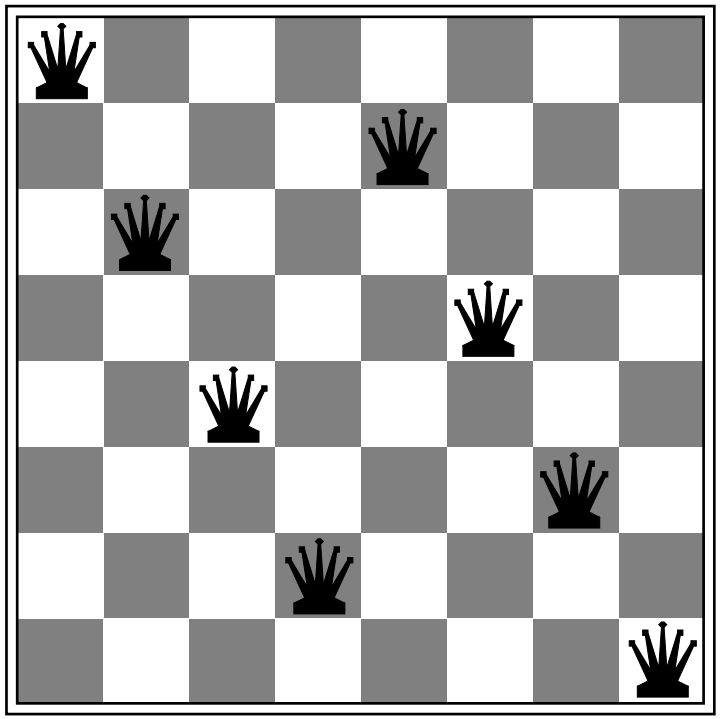
Bernhard Nebel

07.02.2017

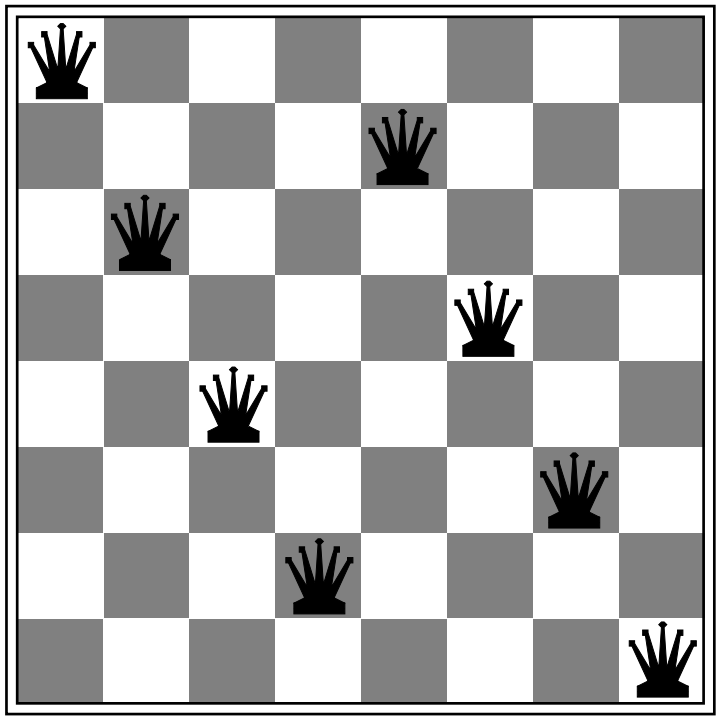
UNI FREIBURG

Motivation

Schwierige Probleme (1)

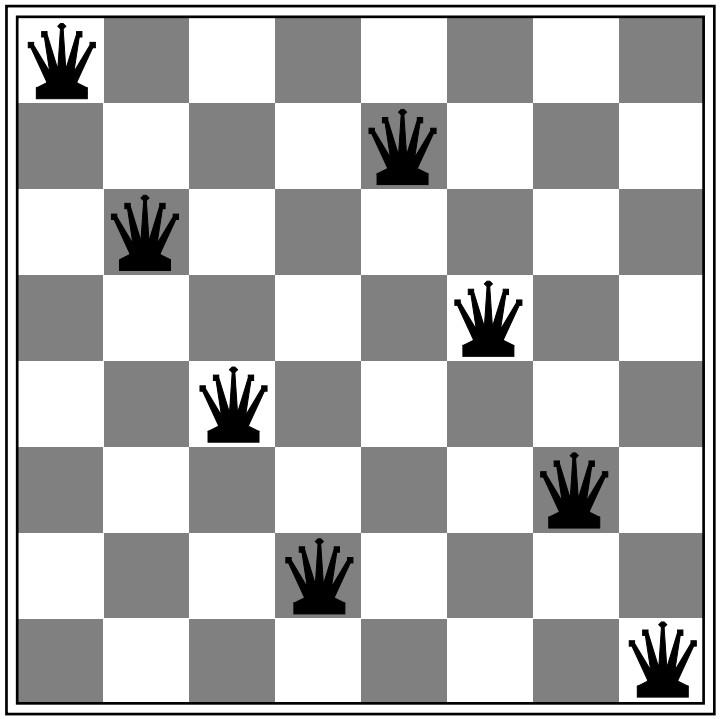


Schwierige Probleme (1)



Platziere die 8 Damen so, dass sie sich nicht schlagen können

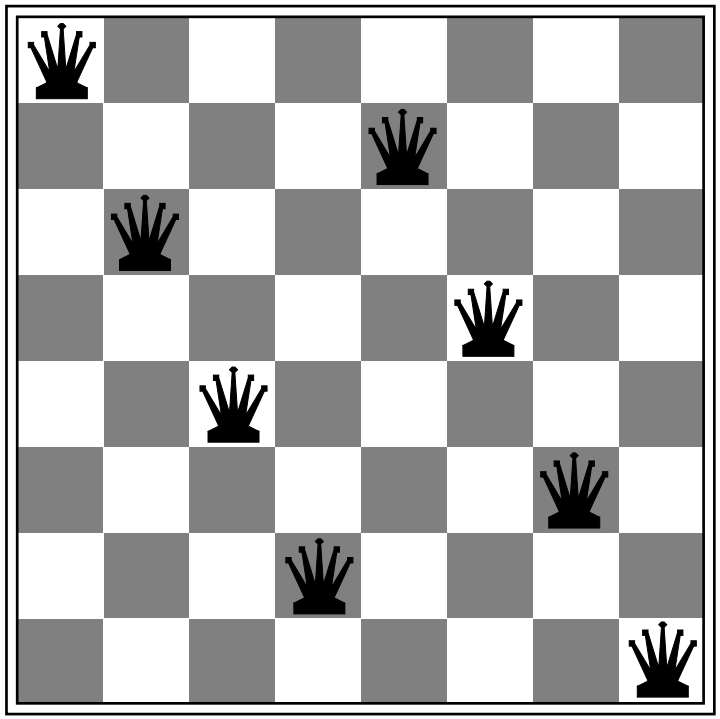
Schwierige Probleme (1)



Platziere die 8 Damen so, dass sie sich nicht schlagen können

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

Schwierige Probleme (1)

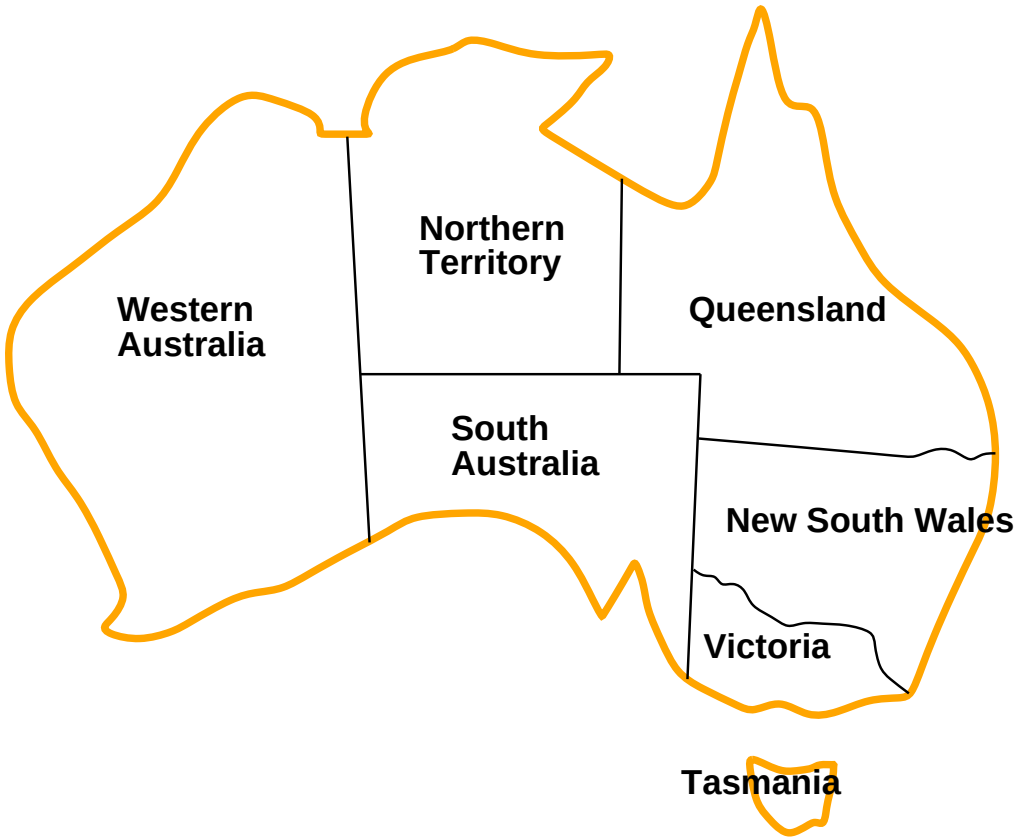


Platziere die 8 Damen so, dass sie sich nicht schlagen können

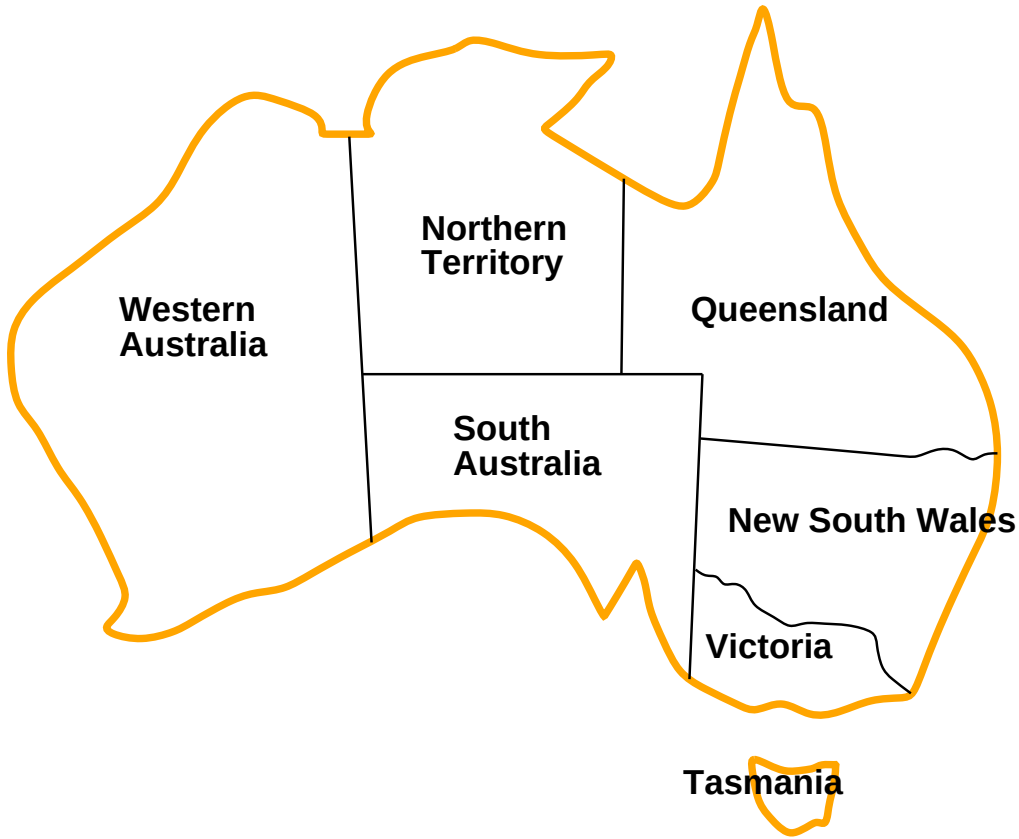
			9			7	2	8
2	7	8			3		1	
	9					6	4	
	5			6		2		
		6				3		
	1			5				
1			7		6		3	4
			5		4			
7		9	1			8		5

Fülle die leeren Felder entsprechend der Sudoku-Regeln

Schwierige Probleme (2)



Schwierige Probleme (2)



Färbe die australischen Bundesstaaten so mit drei Farben ein, dass zwei aneinander stoßende Staaten nicht die gleiche Farbe haben.

Wo liegt der Fehler auf der letzten Folie?

UNI FREIBURG

Wo liegt der Fehler auf der letzten Folie?

Wo liegt der Fehler auf der letzten Folie?

Sicht auf die ANU (Australian National University) und den Telstra-Turm in der Hauptstadt Canberra.

Wo liegt der Fehler auf der letzten Folie?

Sicht auf die ANU (Australian National University) und den Telstra-Turm in der Hauptstadt Canberra. Canberra liegt innerhalb des *Australian Capital Territory* (ACT), das wiederum innerhalb von NSW liegt.

UNI FREIBURG

Constraint-Satisfaction-Probleme

Was haben 8 Damen, Sudokus, und das Färben einer Landkarte gemeinsam?

UNI FREIBURG

- Es handelt sich um kombinatorische Probleme, auch Constraint-Satisfaction-Probleme (CSP) genannt:

Was haben 8 Damen, Sudokus, und das Färben einer Landkarte gemeinsam?

UNI FREIBURG

- Es handelt sich um kombinatorische Probleme, auch Constraint-Satisfaction-Probleme (CSP) genannt:
  - Es existieren n Variablen X_i, die Werte aus einem Bereich $D = \{d_1, d_2, \ldots, d_m\}$ annehmen können.

Was haben 8 Damen, Sudokus, und das Färben einer Landkarte gemeinsam?

UNI FREIBURG

- Es handelt sich um kombinatorische Probleme, auch Constraint-Satisfaction-Probleme (CSP) genannt:
  - Es existieren n Variablen X_i, die Werte aus einem Bereich $D = \{d_1, d_2, \ldots, d_m\}$ annehmen können.
  - Es gibt Bedingungen (Constraints) für die Belegung der Variablen, die erfüllt sein müssen, z.B. $X_i \neq X_{2i}$ für alle i.

Was haben 8 Damen, Sudokus, und das Färben einer Landkarte gemeinsam?

UNI FREIBURG

- Es handelt sich um kombinatorische Probleme, auch Constraint-Satisfaction-Probleme (CSP) genannt:
  - Es existieren n Variablen X_i, die Werte aus einem Bereich $D = \{d_1, d_2, \ldots, d_m\}$ annehmen können.
  - Es gibt Bedingungen (Constraints) für die Belegung der Variablen, die erfüllt sein müssen, z.B. $X_i \neq X_{2i}$ für alle i.
  - Eine Lösung eines CSP ist eine Belegung der Variablen mit Werten, so dass alle Constraints erfüllt sind.

Was haben 8 Damen, Sudokus, und das Färben einer Landkarte gemeinsam?

UNI FREIBURG

- Es handelt sich um kombinatorische Probleme, auch Constraint-Satisfaction-Probleme (CSP) genannt:
  - Es existieren n Variablen X_i, die Werte aus einem Bereich $D = \{d_1, d_2, \ldots, d_m\}$ annehmen können.
  - Es gibt Bedingungen (Constraints) für die Belegung der Variablen, die erfüllt sein müssen, z.B. $X_i \neq X_{2i}$ für alle i.
  - Eine Lösung eines CSP ist eine Belegung der Variablen mit Werten, so dass alle Constraints erfüllt sind.
- Diese Probleme zeichnen sich dadurch aus, dass der Raum der möglichen Lösungen (der Suchraum) oft astronomisch groß ist, und deshalb nicht vollständig abgesucht werden kann.

Was haben 8 Damen, Sudokus, und das Färben einer Landkarte gemeinsam?

UNI FREIBURG

- Es handelt sich um kombinatorische Probleme, auch Constraint-Satisfaction-Probleme (CSP) genannt:
  - Es existieren n Variablen X_i, die Werte aus einem Bereich $D = \{d_1, d_2, \ldots, d_m\}$ annehmen können.
  - Es gibt Bedingungen (Constraints) für die Belegung der Variablen, die erfüllt sein müssen, z.B. $X_i \neq X_{2i}$ für alle i.
  - Eine Lösung eines CSP ist eine Belegung der Variablen mit Werten, so dass alle Constraints erfüllt sind.
- Diese Probleme zeichnen sich dadurch aus, dass der Raum der möglichen Lösungen (der Suchraum) oft astronomisch groß ist, und deshalb nicht vollständig abgesucht werden kann.
- Beispiel Sudoku: Meist müssen $81 - 17 = 64$ Felder mit den Ziffern 1 bis 9 belegt werden. Das sind $9^{64} \approx 10^{61}$ Möglichkeiten.

Australien einfärben

UNI FREIBURG

- Wir haben 7 CSP-Variablen: WA, NT, SA, Q, NSW, V, T.

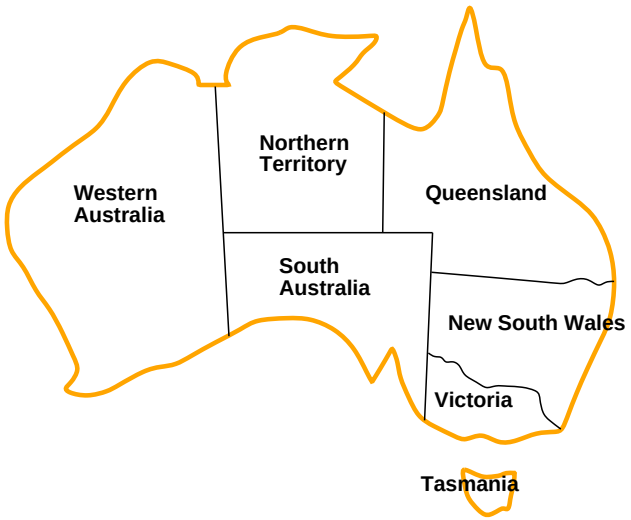
Australien einfärben

UNI FREIBURG

- Wir haben 7 CSP-Variablen: WA, NT, SA, Q, NSW, V, T.
- Diese können die Werte *red*, *blue*, *green* annehmen.

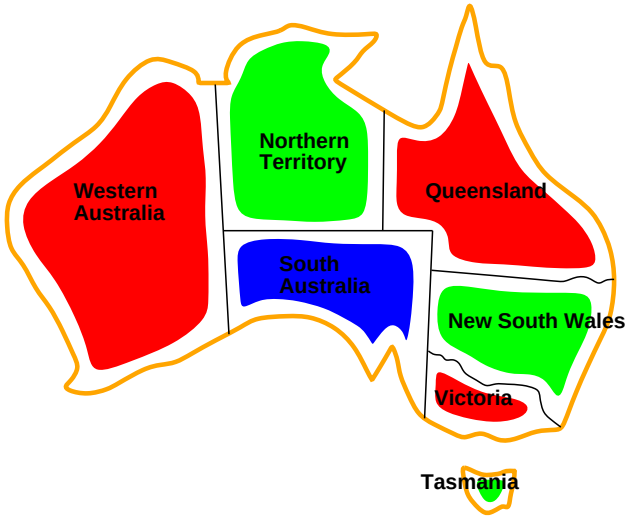
Australien einfärben

- Wir haben 7 CSP-Variablen: WA, NT, SA, Q, NSW, V, T.
- Diese können die Werte *red*, *blue*, *green* annehmen.
- Die Constraints sind: $WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q, SA \neq NSW, SA \neq V, Q \neq NSW, NSW \neq V$.



Australien einfärben

- Wir haben 7 CSP-Variablen: WA, NT, SA, Q, NSW, V, T.
- Diese können die Werte *red*, *blue*, *green* annehmen.
- Die Constraints sind: $WA \neq NT, WA \neq SA, NT \neq SA, NT \neq Q, SA \neq Q, SA \neq NSW, SA \neq V, Q \neq NSW, NSW \neq V$.
- Eine mögliche Lösung wäre:
  WA = *red*, NT = *green*, SA = *blue*, Q = *red*, NSW = *green*, V = *red*, T = *green*.



8 Damen platzieren (1)

UNI FREIBURG

- 16 CSP-Variablen: R_i, C_i (row, column) für die Damen $i = 1, \ldots, 8$

8 Damen platzieren (1)

UNI FREIBURG

- 16 CSP-Variablen: R_i, C_i (row, column) für die Damen $i = 1, \ldots, 8$
- 8 verschiedene Werte: $k = 1, \ldots, 8$ (für die jeweilige Reihe oder Spalte)

8 Damen platzieren (1)

UNI FREIBURG

- 16 CSP-Variablen: R_i, C_i (row, column) für die Damen $i = 1, \ldots, 8$
- 8 verschiedene Werte: $k = 1, \ldots, 8$ (für die jeweilige Reihe oder Spalte)
- Constraints:

8 Damen platzieren (1)

UNI FREIBURG

- 16 CSP-Variablen: R_i, C_i (row, column) für die Damen $i = 1, \ldots, 8$
- 8 verschiedene Werte: $k = 1, \ldots, 8$ (für die jeweilige Reihe oder Spalte)
- Constraints:
  1. $R_i \neq R_j$ für alle $i \neq j$ (die Damen sollen in unterschiedlichen Reihen stehen)

8 Damen platzieren (1)

UNI FREIBURG

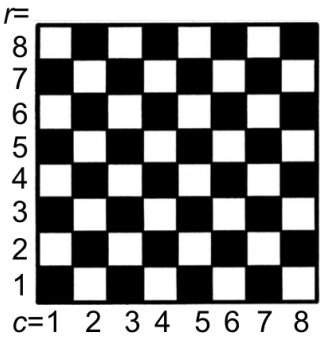
- 16 CSP-Variablen: R_i, C_i (row, column) für die Damen $i = 1, \ldots, 8$
- 8 verschiedene Werte: $k = 1, \ldots, 8$ (für die jeweilige Reihe oder Spalte)
- Constraints:
  1. $R_i \neq R_j$ für alle $i \neq j$ (die Damen sollen in unterschiedlichen Reihen stehen)
  2. $C_i \neq C_j$ für alle $i \neq j$ (die Damen sollen in unterschiedlichen Spalten stehen)

8 Damen platzieren (1)

UNI FREIBURG

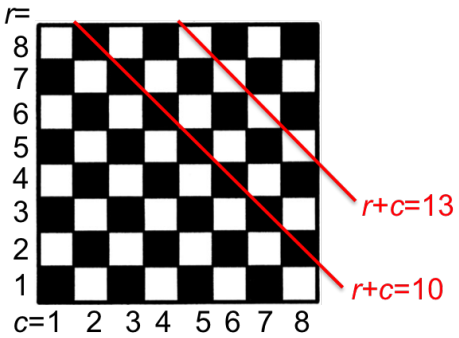
- 16 CSP-Variablen: R_i, C_i (row, column) für die Damen $i = 1, \ldots, 8$
- 8 verschiedene Werte: $k = 1, \ldots, 8$ (für die jeweilige Reihe oder Spalte)
- Constraints:
  1. $R_i \neq R_j$ für alle $i \neq j$ (die Damen sollen in unterschiedlichen Reihen stehen)
  2. $C_i \neq C_j$ für alle $i \neq j$ (die Damen sollen in unterschiedlichen Spalten stehen)
  3. die Damen sollen nicht auf einer gemeinsamen Diagonalen stehen

8 Damen platzieren (2): Diagonalen-Constraints



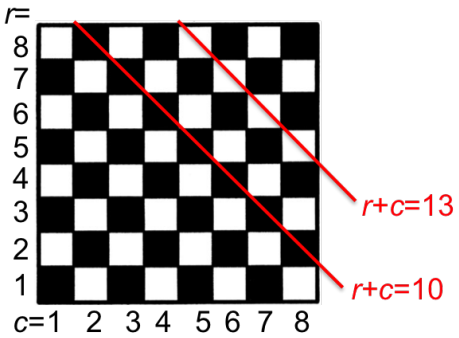
- Auf dem Schachbrett kann man die Diagonalen durch Summen bzw. Differenzen der Reihen- und Spalten-Indizes charakterisieren.

8 Damen platzieren (2): Diagonalen-Constraints



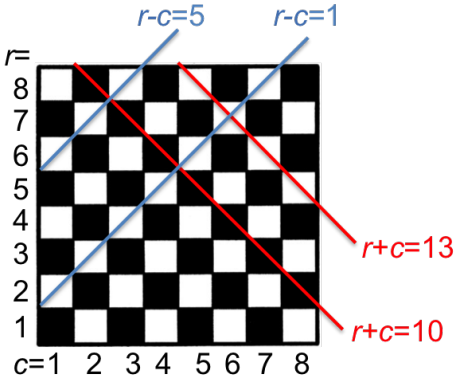
- Auf dem Schachbrett kann man die Diagonalen durch Summen bzw. Differenzen der Reihen- und Spalten-Indizes charakterisieren.
- Die Diagonalen von links oben nach rechts unten haben konstante Summen, die alle verschieden sind.

8 Damen platzieren (2): Diagonalen-Constraints



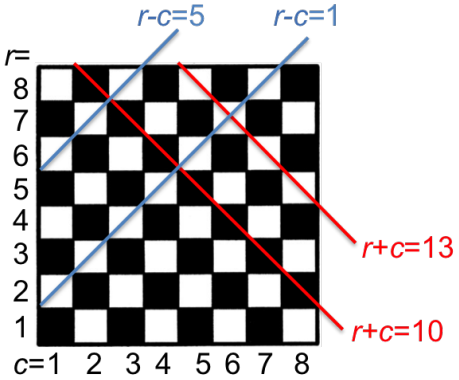
- Auf dem Schachbrett kann man die Diagonalen durch Summen bzw. Differenzen der Reihen- und Spalten-Indizes charakterisieren.
- Die Diagonalen von links oben nach rechts unten haben konstante Summen, die alle verschieden sind.
- D.h. $R_i + C_i \neq R_j + C_j$ für alle Damen i, j mit $i \neq j$ beschreibt die gewünschten Constraints.

8 Damen platzieren (2): Diagonalen-Constraints



- Auf dem Schachbrett kann man die Diagonalen durch Summen bzw. Differenzen der Reihen- und Spalten-Indizes charakterisieren.
- Die Diagonalen von links oben nach rechts unten haben konstante Summen, die alle verschieden sind.
- D.h. $R_i + C_i \neq R_j + C_j$ für alle Damen i, j mit $i \neq j$ beschreibt die gewünschten Constraints.
- Die Diagonalen von links unten nach rechts oben haben konstante Differenzen, die ebenfalls alle verschieden sind.

8 Damen platzieren (2): Diagonalen-Constraints



- Auf dem Schachbrett kann man die Diagonalen durch Summen bzw. Differenzen der Reihen- und Spalten-Indizes charakterisieren.
- Die Diagonalen von links oben nach rechts unten haben konstante Summen, die alle verschieden sind.
- D.h. $R_i + C_i \neq R_j + C_j$ für alle Damen i, j mit $i \neq j$ beschreibt die gewünschten Constraints.
- Die Diagonalen von links unten nach rechts oben haben konstante Differenzen, die ebenfalls alle verschieden sind.
- D.h. $R_i - C_i \neq R_j - C_j$ für $i \neq j$ sind die Constraints.

Damen platzieren (3): Suchraum-Reduktion

UNI FREIBURG

- Es dauert rund 10^{-6} Sekunden, um eine Stellung zu testen.

Damen platzieren (3): Suchraum-Reduktion

UNI FREIBURG

- Es dauert rund 10^{-6} Sekunden, um eine Stellung zu testen.
- Wir können die erste Dame auf 64 verschiedene Felder stellen, die zweite auf 63, ...

Damen platzieren (3): Suchraum-Reduktion

UNI FREIBURG

- Es dauert rund 10^{-6} Sekunden, um eine Stellung zu testen.
- Wir können die erste Dame auf 64 verschiedene Felder stellen, die zweite auf 63, ...
- Wir haben $64!/(64-8)! \approx 1.8 \cdot 10^{14}$ Möglichkeiten. D.h. wir brauchen rund $1.8 \cdot 10^{8}$ Sekunden $\approx$ 7 Jahre Rechenzeit, um alle Stellungen zu testen.

Damen platzieren (3): Suchraum-Reduktion

UNI FREIBURG

- Es dauert rund 10^{-6} Sekunden, um eine Stellung zu testen.
- Wir können die erste Dame auf 64 verschiedene Felder stellen, die zweite auf 63, ...
- Wir haben $64!/(64-8)! \approx 1.8 \cdot 10^{14}$ Möglichkeiten. D.h. wir brauchen rund $1.8 \cdot 10^{8}$ Sekunden $\approx$ 7 Jahre Rechenzeit, um alle Stellungen zu testen.
- Da die Damen aber nicht unterscheidbar sind, und in jeder Reihe genau eine Dame stehen muss, können wir die Reihenvariaben mit $R_i = i$ vorbelegen.

Damen platzieren (3): Suchraum-Reduktion

UNI FREIBURG

- Es dauert rund 10^{-6} Sekunden, um eine Stellung zu testen.
- Wir können die erste Dame auf 64 verschiedene Felder stellen, die zweite auf 63, ...
- Wir haben $64!/(64-8)! \approx 1.8 \cdot 10^{14}$ Möglichkeiten. D.h. wir brauchen rund $1.8 \cdot 10^8$ Sekunden $\approx$ 7 Jahre Rechenzeit, um alle Stellungen zu testen.
- Da die Damen aber nicht unterscheidbar sind, und in jeder Reihe genau eine Dame stehen muss, können wir die Reihenvariaben mit $R_i = i$ vorbelegen.
- Damit ergeben sich dann nur noch $8^8 \approx 1.7 \cdot 10^7$ Möglichkeiten, entsprechend 17 Sekunden Rechenzeit.

Sudoku

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.

Sudoku

UNI FREIBURG

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

Sudoku

UNI FREIBURG

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.

Sudoku

UNI FREIBURG

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.

Sudoku

UNI FREIBURG

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.

Sudoku

UNI FREIBURG

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.

Sudoku

UNI FREIBURG

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.
- In jeder Gruppe müssen die Ziffern 1 bis 9 genau einmal vorkommen.

Sudoku

UNI FREIBURG

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.
- In jeder Gruppe müssen die Ziffern 1 bis 9 genau einmal vorkommen.
- Für eine gegebene Zelle heißen alle Zellen, die in einer Gruppe mit dieser Zelle vorkommen, *Peers* dieser Zelle.

Sudoku

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.
- In jeder Gruppe müssen die Ziffern 1 bis 9 genau einmal vorkommen.
- Für eine gegebene Zelle heißen alle Zellen, die in einer Gruppe mit dieser Zelle vorkommen, *Peers* dieser Zelle.

Sudoku

- Ein Sudoku-Feld besteht aus 81 Zellen, in denen jeweils die Ziffern 1 bis 9 eingetragen werden sollen.
- Diese werden gerne wie folgt durchnummeriert:

A1	A2	A3	A4	A5	A6	A7	A8	A9
B1	B2	B3	B4	B5	B6	B7	B8	B9
C1	C2	C3	C4	C5	C6	C7	C8	C9
D1	D2	D3	D4	D5	D6	D7	D8	D9
E1	E2	E3	E4	E5	E6	E7	E8	E9
F1	F2	F3	F4	F5	F6	F7	F8	F9
G1	G2	G3	G4	G5	G6	G7	G8	G9
H1	H2	H3	H4	H5	H6	H7	H8	H9
I1	I2	I3	I4	I5	I6	I7	I8	I9

- Jeweils neun Zellen einer Zeile, einer Spalte oder eines Blocks bilden eine Gruppe.
- In jeder Gruppe müssen die Ziffern 1 bis 9 genau einmal vorkommen.
- Für eine gegebene Zelle heißen alle Zellen, die in einer Gruppe mit dieser Zelle vorkommen, *Peers* dieser Zelle.
- Die Peers einer Zelle müssen alle einen anderen Wert als die Zelle haben!

Sudoku (2): CSP-Formulierung

UNI FREIBURG

- Wir haben 81 CSP-Variablen: A1 . . . I8,

Sudoku (2): CSP-Formulierung

UNI FREIBURG

- Wir haben 81 CSP-Variablen: A1 . . . I8,
- Diese können die Werte 1, 2, . . . 9 annehmen.

Sudoku (2): CSP-Formulierung

UNI FREIBURG

- Wir haben 81 CSP-Variablen: A1 . . . I8,
- Diese können die Werte 1, 2, . . . 9 annehmen.
- Die Constraints sind: Jede Zelle muss einen Wert besitzen, der verschieden ist von den Werten ihrer Peers.

Sudoku (3): Suchraum

UNI FREIBURG

- Der Suchraum hat in den meisten Fällen (17 Vorgaben) eine Größe von ca. 10^{61} möglichen Kombinationen.
- Würden wir eine Milliarde (10^9) Kombinationen pro Sekunde testen können, wäre die benötigte Rechenzeit $10^{61}/(10^9 \cdot 3 \cdot 10^7) \approx 3 \cdot 10^{44}$ Jahre.
- Die Lebensdauer des Weltalls wird mit 10^{11} Jahren angenommen (falls das Weltall geschlossen ist).
- Selbst bei einer Beschleunigung um den Faktor 10^{30} würde die Rechnung nicht innerhalb der Lebensdauer des Weltalls abgeschlossen werden können.
- Trotzdem scheint das Lösen von Sudokus ja nicht so schwierig zu sein ...

UNI FREIBURG

Backtracking-Suche

Abkürzungen wählen

UNI FREIBURG

- Bei den genannten Abschätzungen wurde ja immer davon ausgegangen, dass wir immer alle CSP-Variablen mit Werten belegen und dann testen, ob es eine Lösung ist.

Abkürzungen wählen

UNI FREIBURG

- Bei den genannten Abschätzungen wurde ja immer davon ausgegangen, dass wir immer alle CSP-Variablen mit Werten belegen und dann testen, ob es eine Lösung ist.
- Dabei würden wir aber viele Kombinationen testen, die ganz offensichtlich keine Lösungen sind.

Abkürzungen wählen

UNI FREIBURG

- Bei den genannten Abschätzungen wurde ja immer davon ausgegangen, dass wir immer alle CSP-Variablen mit Werten belegen und dann testen, ob es eine Lösung ist.
- Dabei würden wir aber viele Kombinationen testen, die ganz offensichtlich keine Lösungen sind.
- Wenn z.B. beim Australienproblem *WA* und *NT* mit der gleichen Farbe belegt wurden, dann werden alle Vervollständigungen keine Lösung sein!

Abkürzungen wählen

UNI FREIBURG

- Bei den genannten Abschätzungen wurde ja immer davon ausgegangen, dass wir immer alle CSP-Variablen mit Werten belegen und dann testen, ob es eine Lösung ist.
- Dabei würden wir aber viele Kombinationen testen, die ganz offensichtlich keine Lösungen sind.
- Wenn z.B. beim Australienproblem *WA* und *NT* mit der gleichen Farbe belegt wurden, dann werden alle Vervollständigungen keine Lösung sein!
- Man kann an dieser Stelle abkürzen und z.B. für *NT* eine andere Farbe ausprobieren.

Abkürzungen wählen

UNI FREIBURG

- Bei den genannten Abschätzungen wurde ja immer davon ausgegangen, dass wir immer alle CSP-Variablen mit Werten belegen und dann testen, ob es eine Lösung ist.
- Dabei würden wir aber viele Kombinationen testen, die ganz offensichtlich keine Lösungen sind.
- Wenn z.B. beim Australienproblem *WA* und *NT* mit der gleichen Farbe belegt wurden, dann werden alle Vervollständigungen keine Lösung sein!
- Man kann an dieser Stelle abkürzen und z.B. für *NT* eine andere Farbe ausprobieren.
- Idee: Schrittweise Werte an CSP-Variablen zuweisen, wobei die Constraints der schon zugewiesenen CSP-Variablen immer erfüllt sein müssen.

Abkürzungen wählen

UNI FREIBURG

- Bei den genannten Abschätzungen wurde ja immer davon ausgegangen, dass wir immer alle CSP-Variablen mit Werten belegen und dann testen, ob es eine Lösung ist.
- Dabei würden wir aber viele Kombinationen testen, die ganz offensichtlich keine Lösungen sind.
- Wenn z.B. beim Australienproblem *WA* und *NT* mit der gleichen Farbe belegt wurden, dann werden alle Vervollständigungen keine Lösung sein!
- Man kann an dieser Stelle abkürzen und z.B. für *NT* eine andere Farbe ausprobieren.
- Idee: Schrittweise Werte an CSP-Variablen zuweisen, wobei die Constraints der schon zugewiesenen CSP-Variablen immer erfüllt sein müssen.
- Wichtig: Dabei muss man manchmal auch Entscheidungen rückgängig machen, wenn wir keine Vervollständigung finden können.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.
2. Weise der CSP-Variablen einen Wert zu, der alle Constraints mit schon belegten CSP-Variablen erfüllt.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.
2. Weise der CSP-Variablen einen Wert zu, der alle Constraints mit schon belegten CSP-Variablen erfüllt.
3. Versuche rekursiv eine Belegung für die restlichen CSP-Variablen zu finden.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.
2. Weise der CSP-Variablen einen Wert zu, der alle Constraints mit schon belegten CSP-Variablen erfüllt.
3. Versuche rekursiv eine Belegung für die restlichen CSP-Variablen zu finden.
4. Gelingt dies, sind wir fertig und geben die Belegung zurück.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.
2. Weise der CSP-Variablen einen Wert zu, der alle Constraints mit schon belegten CSP-Variablen erfüllt.
3. Versuche rekursiv eine Belegung für die restlichen CSP-Variablen zu finden.
4. Gelingt dies, sind wir fertig und geben die Belegung zurück.
5. Nimm ansonsten die Belegung der CSP-Variablen zurück, wähle einen bisher noch nicht ausprobierten Wert und belege die CSP-Variable damit. Mache mit Schritt 3 weiter.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.
2. Weise der CSP-Variablen einen Wert zu, der alle Constraints mit schon belegten CSP-Variablen erfüllt.
3. Versuche rekursiv eine Belegung für die restlichen CSP-Variablen zu finden.
4. Gelingt dies, sind wir fertig und geben die Belegung zurück.
5. Nimm ansonsten die Belegung der CSP-Variablen zurück, wähle einen bisher noch nicht ausprobierten Wert und belege die CSP-Variable damit. Mache mit Schritt 3 weiter.
6. Wurden alle Werte erfolglos probiert, gebe `False` zurück.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.
2. Weise der CSP-Variablen einen Wert zu, der alle Constraints mit schon belegten CSP-Variablen erfüllt.
3. Versuche rekursiv eine Belegung für die restlichen CSP-Variablen zu finden.
4. Gelingt dies, sind wir fertig und geben die Belegung zurück.
5. Nimm ansonsten die Belegung der CSP-Variablen zurück, wähle einen bisher noch nicht ausprobierten Wert und belege die CSP-Variable damit. Mache mit Schritt 3 weiter.
6. Wurden alle Werte erfolglos probiert, gebe `False` zurück.

Man nennt diese Art der Suche auch Backtracking-Suche, da man im Schritt 5 einen Schritt zurück nimmt und etwas anderes probiert.

Rekursive Suche mit Rücksetzen

UNI FREIBURG

1. Wähle eine noch unbelegte CSP-Variable aus.
2. Weise der CSP-Variablen einen Wert zu, der alle Constraints mit schon belegten CSP-Variablen erfüllt.
3. Versuche rekursiv eine Belegung für die restlichen CSP-Variablen zu finden.
4. Gelingt dies, sind wir fertig und geben die Belegung zurück.
5. Nimm ansonsten die Belegung der CSP-Variablen zurück, wähle einen bisher noch nicht ausprobierten Wert und belege die CSP-Variable damit. Mache mit Schritt 3 weiter.
6. Wurden alle Werte erfolglos probiert, gebe `False` zurück.

Man nennt diese Art der Suche auch Backtracking-Suche, da man im Schritt 5 einen Schritt zurück nimmt und etwas anderes probiert.

Statt Rücksetzen kann man beim rekursiven Aufruf in Schritt 3 natürlich eine Kopie der Variablenbelegung nutzen.

Backtracking in Oz

UNI FREIBURG

Für unser Beispiel zum Einfärben der australischen Landkarte könnte das so aussehen:

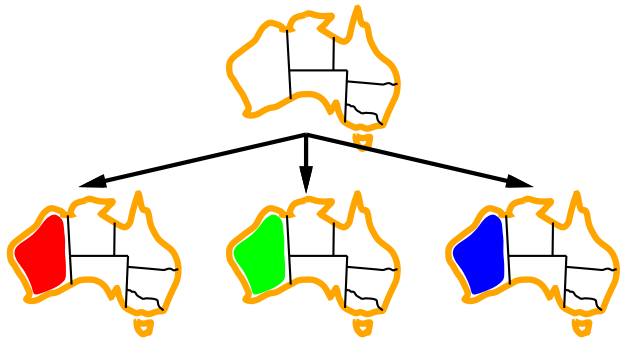
Backtracking in Oz

Für unser Beispiel zum Einfärben der australischen Landkarte könnte das so aussehen:

Backtracking in Oz

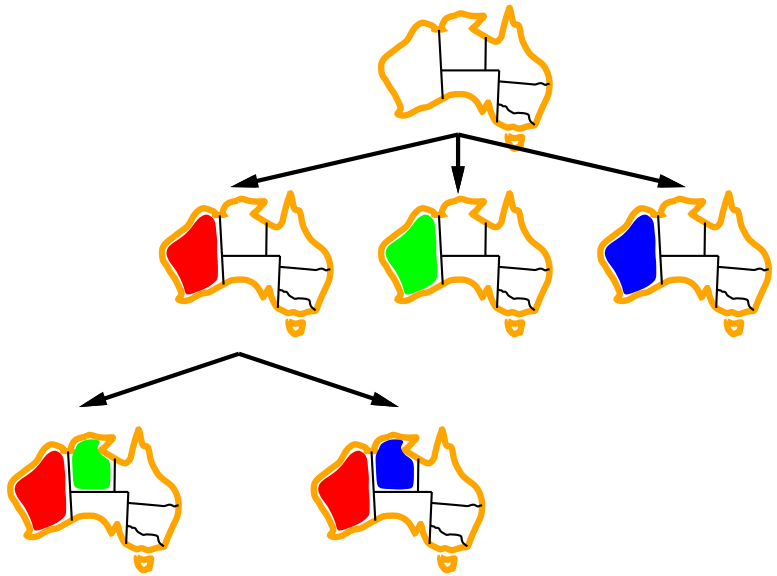
UNI FREIBURG

Für unser Beispiel zum Einfärben der australischen Landkarte könnte das so aussehen:



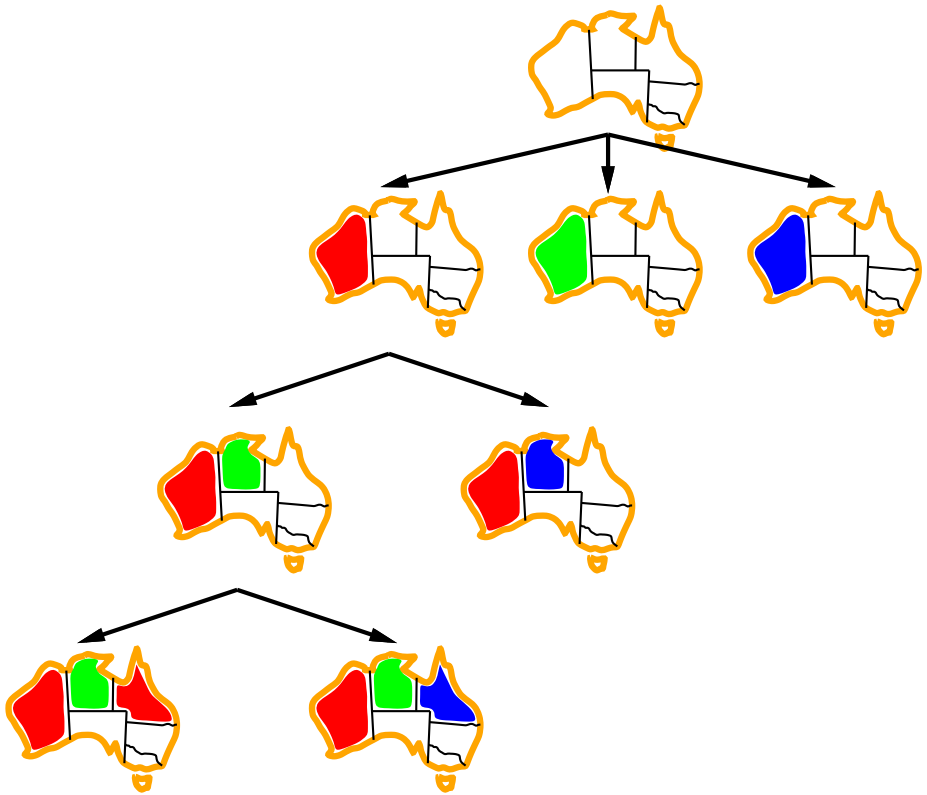
Backtracking in Oz

Für unser Beispiel zum Einfärben der australischen Landkarte könnte das so aussehen:



Backtracking in Oz

Für unser Beispiel zum Einfärben der australischen Landkarte könnte das so aussehen:



Backtracking in Oz – mit Python (1)

UNI FREIBURG

oz.py(1)

```
varlist = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T' )
domain = ('red', 'green', 'blue')
neighbor = dict(WA={'NT', 'SA'}, NT= {'WA', 'SA', 'Q'},
             SA={'WA', 'NT', 'Q', 'NSW', 'V'},
             Q={'NT', 'SA', 'NSW'}, NSW={'Q', 'SA', 'V'},
             V ={'SA', 'NSW'}, T={})
```

Backtracking in Oz – mit Python (1)

UNI FREIBURG

oz.py(1)

```
varlist = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T' )
domain = ('red', 'green', 'blue')
neighbor = dict(WA={'NT', 'SA'}, NT= {'WA', 'SA', 'Q'},
            SA={'WA', 'NT', 'Q', 'NSW', 'V'},
            Q={'NT', 'SA', 'NSW'}, NSW={'Q', 'SA', 'V'},
            V ={'SA', 'NSW'}, T={})
```

- Variablennamen und Werte als Strings innerhalb von Tupeln aufzählen.

Backtracking in Oz – mit Python (1)

UNI FREIBURG

oz.py(1)

```
varlist = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T' )
domain = ('red', 'green', 'blue')
neighbor = dict(WA={'NT', 'SA'}, NT= {'WA', 'SA', 'Q'},
            SA={'WA', 'NT', 'Q', 'NSW', 'V'},
            Q={'NT', 'SA', 'NSW'}, NSW={'Q', 'SA', 'V'},
            V ={'SA', 'NSW'}, T={})
```

- Variablennamen und Werte als Strings innerhalb von Tupeln aufzählen.
- Constraints als ein `dict`, in dem für jeden Staat die Nachbarstaaten angegeben werden.

Backtracking in Oz – mit Python (1)

UNI FREIBURG

oz.py(1)

```
varlist = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T' )
domain = ('red', 'green', 'blue')
neighbor = dict(WA={'NT', 'SA'}, NT= {'WA', 'SA', 'Q'},
            SA={'WA', 'NT', 'Q', 'NSW', 'V'},
            Q={'NT', 'SA', 'NSW'}, NSW={'Q', 'SA', 'V'},
            V ={'SA', 'NSW'}, T={})
```

- Variablennamen und Werte als Strings innerhalb von Tupeln aufzählen.
- Constraints als ein `dict`, in dem für jeden Staat die Nachbarstaaten angegeben werden.
- Belegungen werden über `dicts` realisiert, die dynamisch wachsen.

Backtracking in Oz – mit Python (2)

UNI FREIBURG

- Um ein Element aus einer Liste zu wählen, benutzen wir die Funktion `some`:

Backtracking in Oz – mit Python (2)

UNI FREIBURG

- Um ein Element aus einer Liste zu wählen, benutzen wir die Funktion `some`:

oz.py(2)

```
def some(seq):
    for e in seq:
        if e: return e
    return False
```

Backtracking in Oz – mit Python (2)

UNI FREIBURG

- Um ein Element aus einer Liste zu wählen, benutzen wir die Funktion `some`:

oz.py(2)

```
def some(seq):
    for e in seq:
        if e: return e
    return False
```

- Funktioniert ähnlich wie `any`, gibt aber ein Element zurück, wenn ein nicht-`False` Element vorhanden ist.

Backtracking in Oz – mit Python (3)

UNI FREIBURG

- Die Funktion `assign(vals, x, d)` führt die Zuweisung des Wertes `d` an die CSP-Variable `x` durch:

Backtracking in Oz – mit Python (3)

UNI FREIBURG

- Die Funktion `assign(vals, x, d)` führt die Zuweisung des Wertes `d` an die CSP-Variable `x` durch:

`oz.py(3)`

```
def assign(vals, x, d):
    "assign d to var x if feasible, otherwise return False"
    for y in vals:
        if x in neighbor[y] and vals[y] == d:
            return False
    vals[x] = d
    return vals
```

Backtracking in Oz – mit Python (3)

UNI FREIBURG

- Die Funktion `assign(vals, x, d)` führt die Zuweisung des Wertes `d` an die CSP-Variable `x` durch:

`oz.py(3)`

```
def assign(vals, x, d):
    "assign d to var x if feasible, otherwise return False"
    for y in vals:
        if x in neighbor[y] and vals[y] == d:
            return False
    vals[x] = d
    return vals
```

- `vals` ist das `dict`, in dem die Belegung aufgebaut wird.

Backtracking in Oz – mit Python (3)

UNI FREIBURG

- Die Funktion `assign(vals, x, d)` führt die Zuweisung des Wertes `d` an die CSP-Variable `x` durch:

`oz.py(3)`

```
def assign(vals, x, d):
    "assign d to var x if feasible, otherwise return False"
    for y in vals:
        if x in neighbor[y] and vals[y] == d:
            return False
    vals[x] = d
    return vals
```

- `vals` ist das `dict`, in dem die Belegung aufgebaut wird.
- Erst testen, ob der Wert `d` ein möglicher Wert für die Variable `x` ist, indem die Constraints für bereits belegte CSP-Variablen überprüft werden.

Backtracking in Oz – mit Python (3)

UNI FREIBURG

- Die Funktion `assign(vals, x, d)` führt die Zuweisung des Wertes `d` an die CSP-Variable `x` durch:

`oz.py(3)`

```
def assign(vals, x, d):
    "assign d to var x if feasible, otherwise return False"
    for y in vals:
        if x in neighbor[y] and vals[y] == d:
            return False
    vals[x] = d
    return vals
```

- `vals` ist das `dict`, in dem die Belegung aufgebaut wird.
- Erst testen, ob der Wert `d` ein möglicher Wert für die Variable `x` ist, indem die Constraints für bereits belegte CSP-Variablen überprüft werden.
- Falls nicht, `False` zurück geben.

Backtracking in Oz – mit Python (3)

UNI FREIBURG

- Die Funktion `assign(vals, x, d)` führt die Zuweisung des Wertes `d` an die CSP-Variable `x` durch:

`oz.py(3)`

```
def assign(vals, x, d):
    "assign d to var x if feasible, otherwise return False"
    for y in vals:
        if x in neighbor[y] and vals[y] == d:
            return False
    vals[x] = d
    return vals
```

- `vals` ist das `dict`, in dem die Belegung aufgebaut wird.
- Erst testen, ob der Wert `d` ein möglicher Wert für die Variable `x` ist, indem die Constraints für bereits belegte CSP-Variablen überprüft werden.
- Falls nicht, `False` zurück geben.
- Ansonsten wird `vals` erweitert und zurück gegeben.

Backtracking in Oz – mit Python (4)

UNI FREIBURG

oz.py(4)

```
def search(vals):
  "Recursively search for a satisfying assignment"
  if vals is False: return False # failed earlier
  nextvar = some(x for x in varlist if x not in vals)
  if not nextvar:
    return vals # we have found a complete assignment
  else:
    return some(search(assign(vals.copy(), nextvar, d))
                for d in domain)
```

- `vals` kann `False` werden, wenn `assign` einen Wert nicht zulässt.

Backtracking in Oz – mit Python (4)

UNI FREIBURG

oz.py(4)

```
def search(vals):
  "Recursively search for a satisfying assignment"
  if vals is False: return False # failed earlier
  nextvar = some(x for x in varlist if x not in vals)
  if not nextvar:
    return vals # we have found a complete assignment
  else:
    return some(search(assign(vals.copy(), nextvar, d))
                for d in domain)
```

- `vals` kann `False` werden, wenn `assign` einen Wert nicht zulässt.
- `vals` wird vor jedem Aufruf von assign kopiert!

Backtracking in Oz – mit Python (4)

UNI FREIBURG

oz.py(4)

```
def search(vals):
  "Recursively search for a satisfying assignment"
  if vals is False: return False # failed earlier
  nextvar = some(x for x in varlist if x not in vals)
  if not nextvar:
    return vals # we have found a complete assignment
  else:
    return some(search(assign(vals.copy(), nextvar, d))
                for d in domain)
```

- `vals` kann `False` werden, wenn `assign` einen Wert nicht zulässt.
- `vals` wird vor jedem Aufruf von assign kopiert!
- Dann müssen wir die Belegung nicht nach dem rekursiven Aufruf rückgängig machen.

Backtracking in Oz – mit Python (5)

UNI FREIBURG

oztrace.py

```
def assign(vals, x, d):
    print(" "*len(vals), "check value %s for var %s" % (d, x))
    for y in vals:
        if x in neighbor[y] and vals[y] == d:
            print(" "*len(vals), "not possible!")
            return False
    print(" "*len(vals), "trying out ...")
    vals[x] = d
    return vals
```

Python-Interpreter

```
>>> search(dict())
check value red for var WA
trying out ...
  check value red for var NT
  ...
```

Backtracking in Oz – mit Python-Generatoren

UNI FREIBURG

- Erzeuge alle Lösungen mit einem Generator.

Backtracking in Oz – mit Python-Generatoren

UNI FREIBURG

- Erzeuge alle Lösungen mit einem Generator.
- Fehlschläge müssen nicht zurück geben werden.

Backtracking in Oz – mit Python-Generatoren

UNI FREIBURG

- Erzeuge alle Lösungen mit einem Generator.
- Fehlschläge müssen nicht zurück geben werden.
- Achtung: Der rekursive Generator muss in einer `for`-Schleife aufgerufen werden.

Backtracking in Oz – mit Python-Generatoren

UNI FREIBURG

- Erzeuge alle Lösungen mit einem Generator.
- Fehlschläge müssen nicht zurück geben werden.
- Achtung: Der rekursive Generator muss in einer `for`-Schleife aufgerufen werden.
- Essentiell: Kopieren von `vals`.

Backtracking in Oz – mit Python-Generatoren

UNI FREIBURG

- Erzeuge alle Lösungen mit einem Generator.
- Fehlschläge müssen nicht zurück geben werden.
- Achtung: Der rekursive Generator muss in einer `for`-Schleife aufgerufen werden.
- Essentiell: Kopieren von `vals`.

ozgen.py

```
def search(vals):
  "Recursively search for a satisfying assignment"
  if vals is not False:
    nextvar = some(x for x in varlist if x not in vals)
    if not nextvar:
      yield vals # we have found a complete assignment
    else:
      for d in domain:
        for result in search(assign(vals.copy(),
                                    nextvar, d)): yield result
```

Bemerkungen zur Backtracking-Suche: Variablenauswahl

UNI FREIBURG

Wie sollte man die nächste zu belegende CSP-Variable auswählen?

Bemerkungen zur Backtracking-Suche: Variablenauswahl

UNI FREIBURG

Wie sollte man die nächste zu belegende CSP-Variable auswählen?

- Für die Korrektheit ist es egal, welche Variable man wählt.

Bemerkungen zur Backtracking-Suche: Variablenauswahl

UNI FREIBURG

Wie sollte man die nächste zu belegende CSP-Variable auswählen?

- Für die Korrektheit ist es egal, welche Variable man wählt.
- Es kann aber für die Laufzeit Unterschiede machen.

Bemerkungen zur Backtracking-Suche: Variablenauswahl

UNI FREIBURG

Wie sollte man die nächste zu belegende CSP-Variable auswählen?

- Für die Korrektheit ist es egal, welche Variable man wählt.
- Es kann aber für die Laufzeit Unterschiede machen.

→ Eine gute Heuristik ist es, die Variable zu wählen, die die wenigsten noch möglichen Werte besitzt.

Bemerkungen zur Backtracking-Suche: Variablenauswahl

UNI FREIBURG

Wie sollte man die nächste zu belegende CSP-Variable auswählen?

- Für die Korrektheit ist es egal, welche Variable man wählt.
- Es kann aber für die Laufzeit Unterschiede machen.

→ Eine gute Heuristik ist es, die Variable zu wählen, die die wenigsten noch möglichen Werte besitzt.

- Grund: Reduktion der Verzweigung im Aufrufbaum weit oben.

Bemerkungen zur Backtracking-Suche: Variablenauswahl

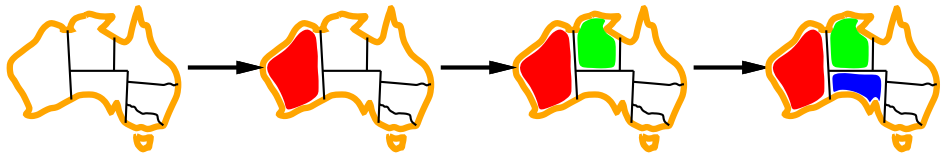
UNI FREIBURG

Wie sollte man die nächste zu belegende CSP-Variable auswählen?

- Für die Korrektheit ist es egal, welche Variable man wählt.
- Es kann aber für die Laufzeit Unterschiede machen.

→ Eine gute Heuristik ist es, die Variable zu wählen, die die wenigsten noch möglichen Werte besitzt.

- Grund: Reduktion der Verzweigung im Aufrufbaum weit oben.
- Beispiel:



Bemerkungen zur Backtracking-Suche: Werteauswahl

UNI FREIBURG

In welcher Reihenfolge sollte man die Werte durchprobieren?

- Für die Korrektheit egal.

Bemerkungen zur Backtracking-Suche: Werteauswahl

UNI FREIBURG

In welcher Reihenfolge sollte man die Werte durchprobieren?

- Für die Korrektheit egal.
- Wenn man schnell eine Lösung finden will, sollte man mit den Werten beginnen, die die anderen Variablen möglichst wenig einschränkt.

Bemerkungen zur Backtracking-Suche: Werteauswahl

UNI FREIBURG

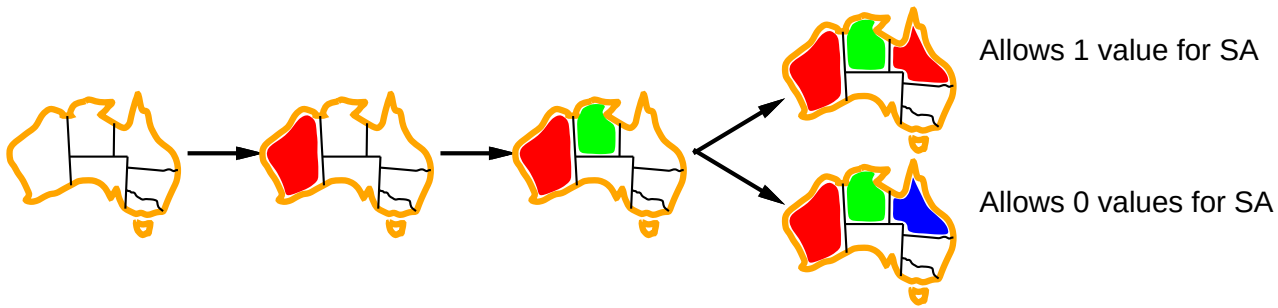
In welcher Reihenfolge sollte man die Werte durchprobieren?

- Für die Korrektheit egal.
- Wenn man schnell eine Lösung finden will, sollte man mit den Werten beginnen, die die anderen Variablen möglichst wenig einschränkt.
- Erfordert allerdings, dass wir voraus schauen und bestimmen, welche Werte bei anderen Variablen noch möglich sind.

Bemerkungen zur Backtracking-Suche: Werteauswahl

In welcher Reihenfolge sollte man die Werte durchprobieren?

- Für die Korrektheit egal.
- Wenn man schnell eine Lösung finden will, sollte man mit den Werten beginnen, die die anderen Variablen möglichst wenig einschränkt.
- Erfordert allerdings, dass wir voraus schauen und bestimmen, welche Werte bei anderen Variablen noch möglich sind.
- Beispiel:

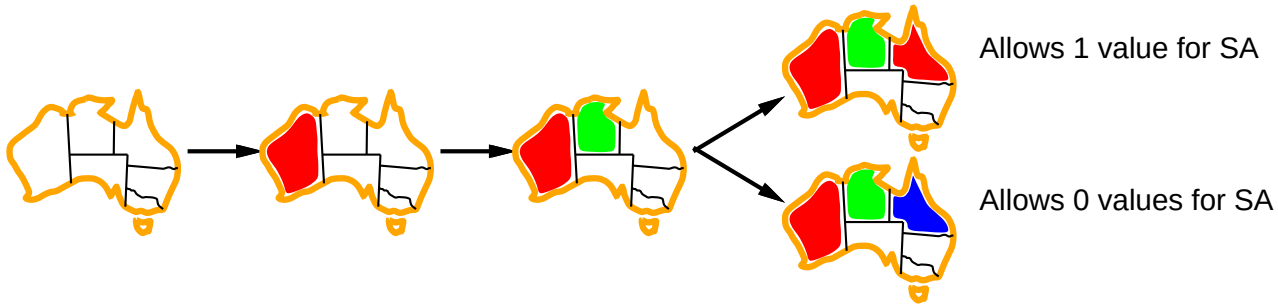


Bemerkungen zur Backtracking-Suche: Werteauswahl

UNI FREIBURG

In welcher Reihenfolge sollte man die Werte durchprobieren?

- Für die Korrektheit egal.
- Wenn man schnell eine Lösung finden will, sollte man mit den Werten beginnen, die die anderen Variablen möglichst wenig einschränkt.
- Erfordert allerdings, dass wir voraus schauen und bestimmen, welche Werte bei anderen Variablen noch möglich sind.
- Beispiel:



- Wir werden im Weiteren aber sowohl Variablen- als auch Werte-Auswahl erst einmal einfach halten.

Backtracking für 8 Damen – mit Python (1)

UNI FREIBURG

- Für die Problemrepräsentation beim 8-Dame-Problem bietet es sich an, die Belegung durch ein Tupel `col` zu repräsentieren, bei dem der i-te Eintrag für die Spalte steht, in der die i-te Dame steht, wobei Dame i in der i-ten Reihe steht ($i = 0, \ldots, 7$).

Backtracking für 8 Damen – mit Python (1)

UNI FREIBURG

- Für die Problemrepräsentation beim 8-Dame-Problem bietet es sich an, die Belegung durch ein Tupel `col` zu repräsentieren, bei dem der i-te Eintrag für die Spalte steht, in der die i-te Dame steht, wobei Dame i in der i-ten Reihe steht ($i = 0, \ldots, 7$).
- Die Constraints ergeben sich dann, wie weiter oben beschrieben.

Backtracking für 8 Damen – mit Python (1)

UNI FREIBURG

- Für die Problemrepräsentation beim 8-Dame-Problem bietet es sich an, die Belegung durch ein Tupel `col` zu repräsentieren, bei dem der i-te Eintrag für die Spalte steht, in der die i-te Dame steht, wobei Dame i in der i-ten Reihe steht ($i = 0, \ldots, 7$).
- Die Constraints ergeben sich dann, wie weiter oben beschrieben.

8queens.py (1)

```
def assign(col, x, d):
    for y in range(len(col)):
        if col[y] == d: # same column?
            return False
        if (col[y] + y == d + x or # same diagonal?
            col[y] - y == d - x):
            return False
    return col + (d,) # return copy!
```

Backtracking für 8 Damen – mit Python (2)

UNI FREIBURG

- Die eigentlich Suchfunktion sieht ganz ähnlich aus wie im Fall der 3-Färbbarkeit von Australien.

Backtracking für 8 Damen – mit Python (2)

UNI FREIBURG

- Die eigentlich Suchfunktion sieht ganz ähnlich aus wie im Fall der 3-Färbbarkeit von Australien.
- Kopiert wird hier die neue Belegung bereits in `assign`, da wir mit Tupeln arbeiten.

Backtracking für 8 Damen – mit Python (2)

UNI FREIBURG

- Die eigentlich Suchfunktion sieht ganz ähnlich aus wie im Fall der 3-Färbbarkeit von Australien.
- Kopiert wird hier die neue Belegung bereits in `assign`, da wir mit Tupeln arbeiten.

8queens.py (2)

```
def search(col):
    if col is not False:
        nextvar = len(col)
        if nextvar == 8:
            return col
        else:
            for d in range(8):
                result = search(assign(col, nextvar, d))
                if result: return result
    return False
```

Backtracking für 8 Damen – mit Python (3)

UNI FREIBURG

- Eigentlich würden wir ja gerne sehen, wie das Schachbrett dann aussieht.

Backtracking für 8 Damen – mit Python (3)

UNI FREIBURG

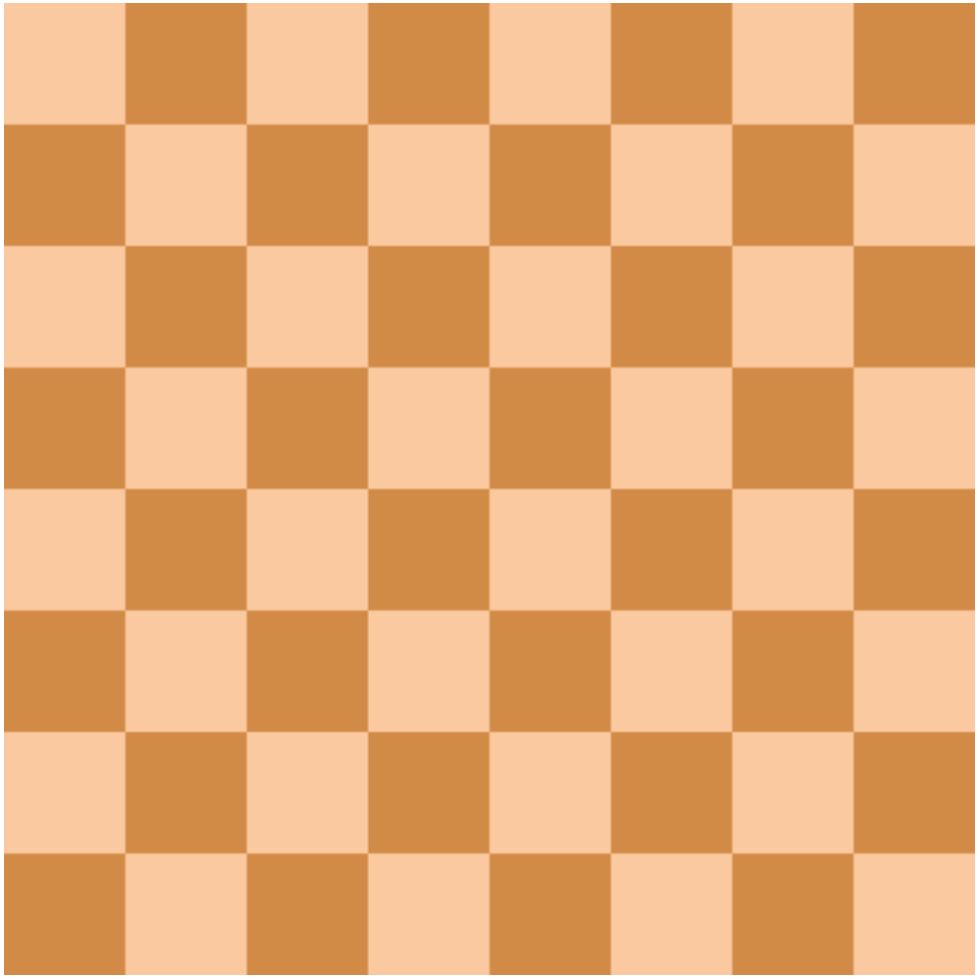
- Eigentlich würden wir ja gerne sehen, wie das Schachbrett dann aussieht.

8queens.py (3)

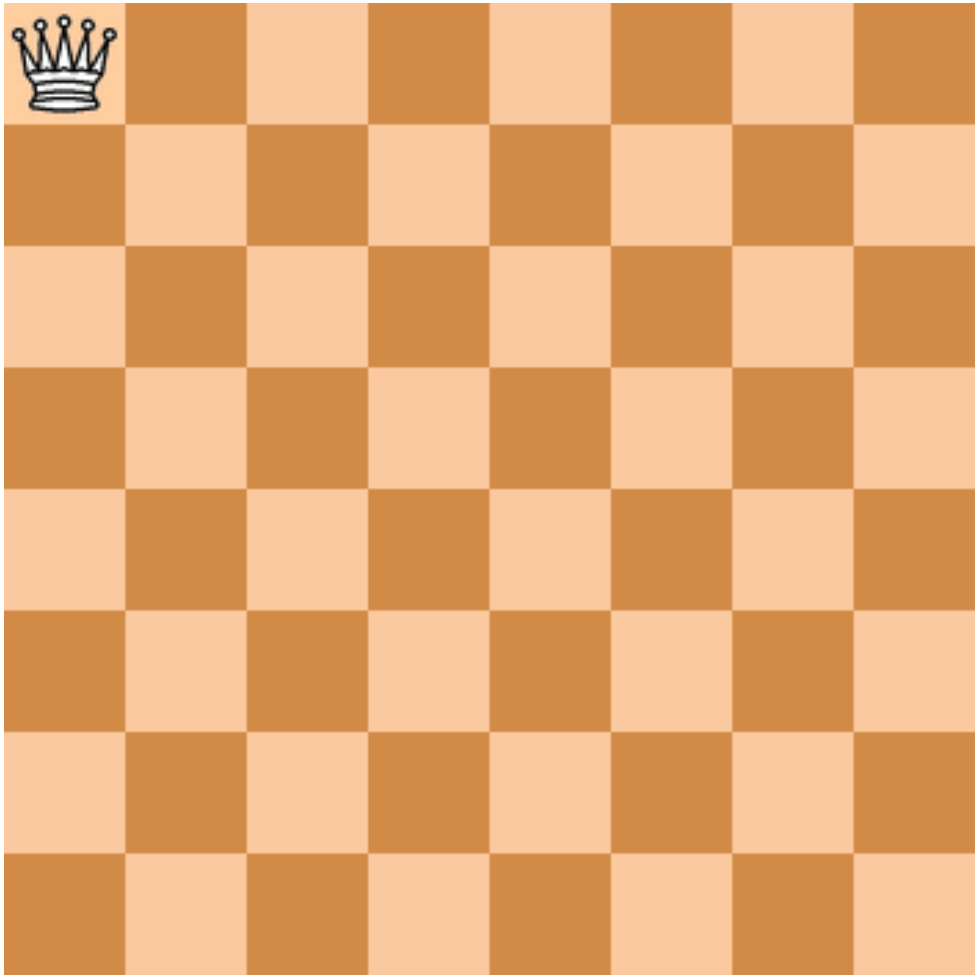
```
def display(col):
    for i in range(8):
        print(". "*col[i], "X ", ". "*(7-col[i]),
              sep="")

if __name__ == "__main__":
    display(search(()))
```

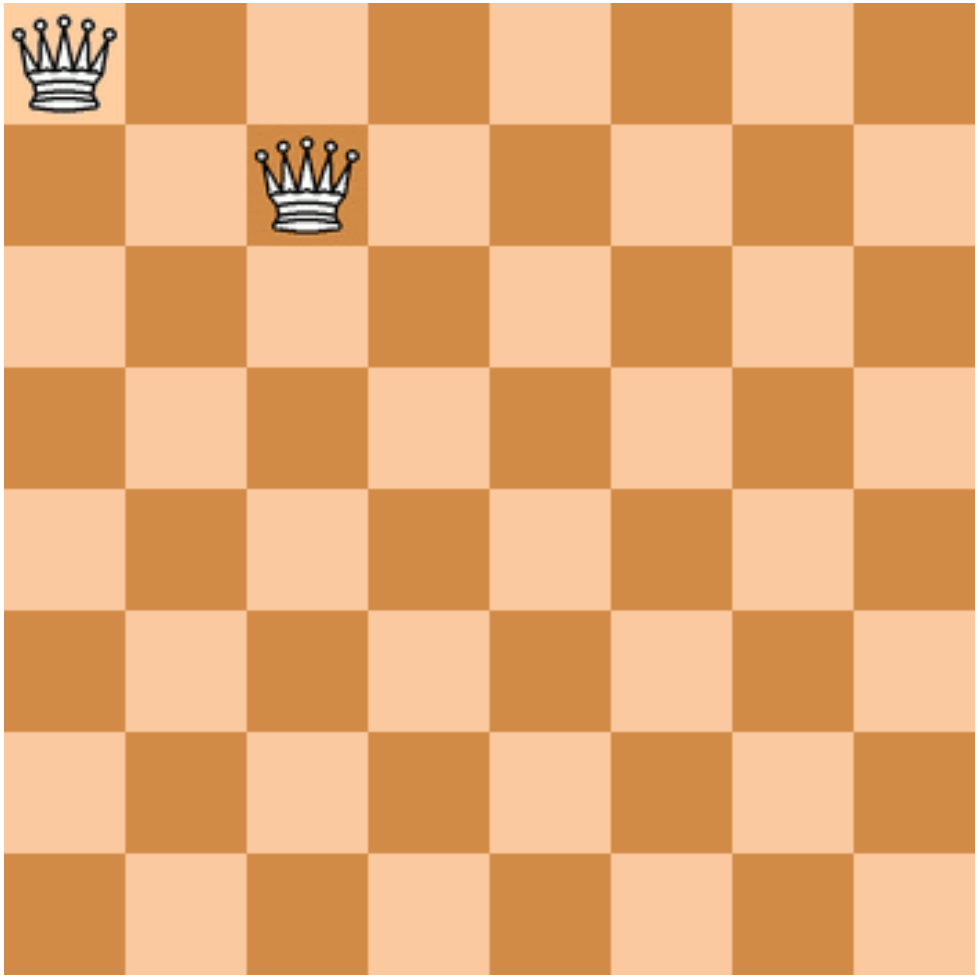
Backtracking für 8 Damen



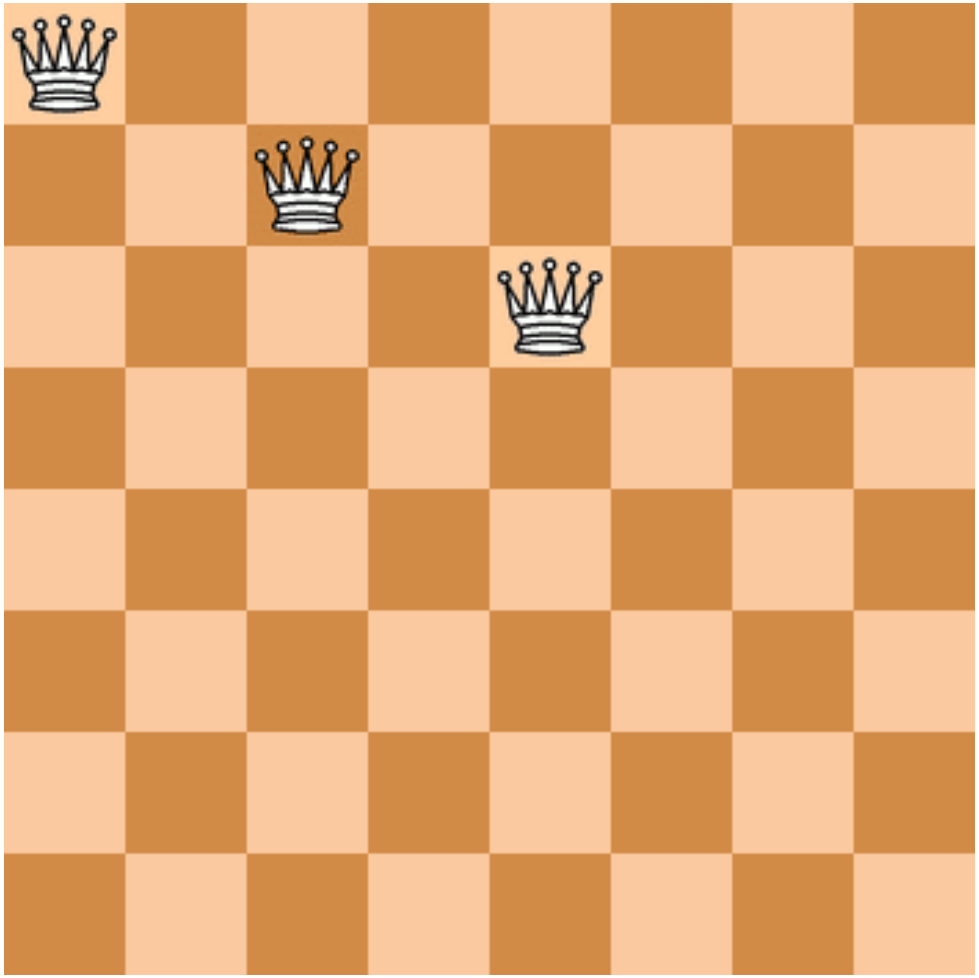
Backtracking für 8 Damen



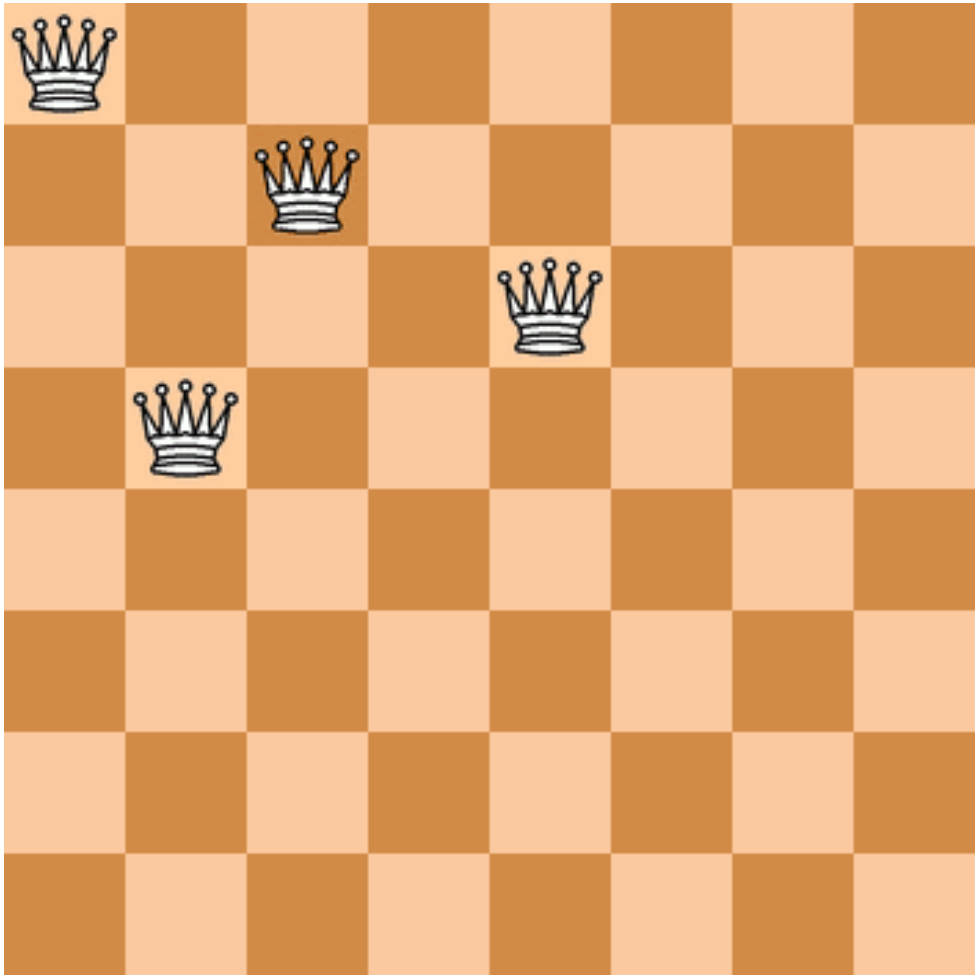
Backtracking für 8 Damen



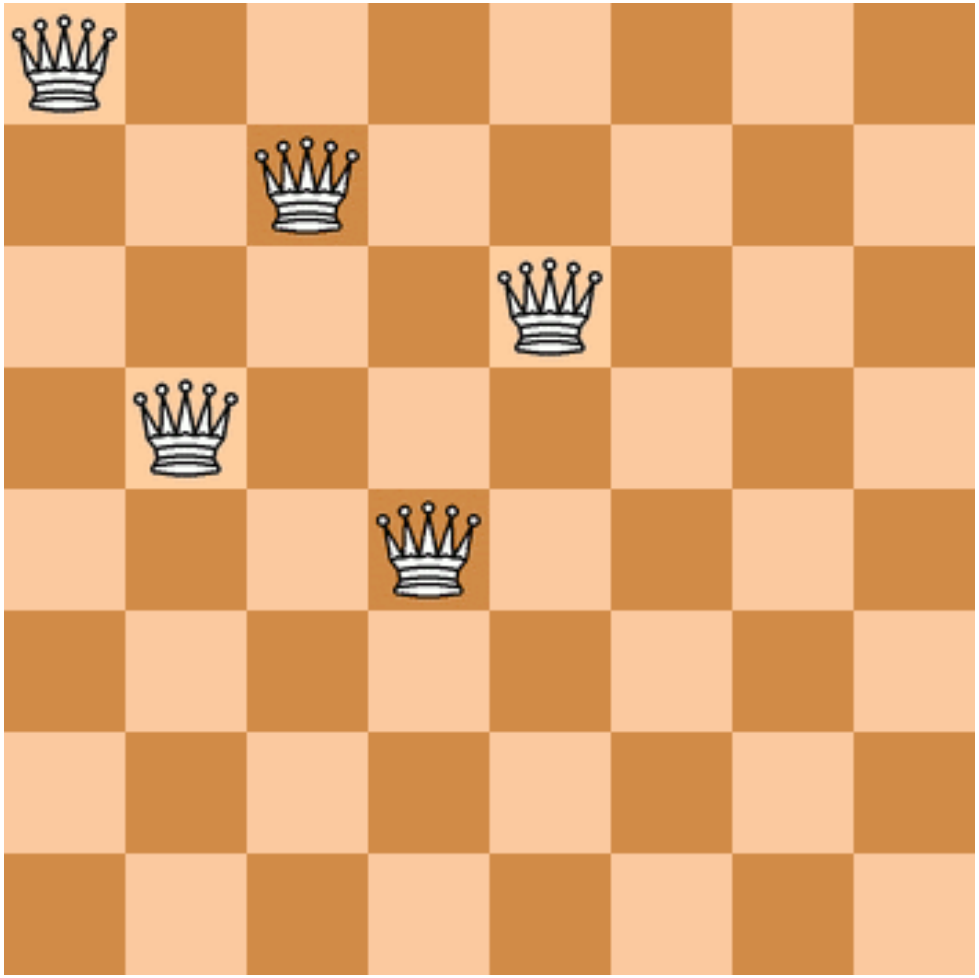
Backtracking für 8 Damen



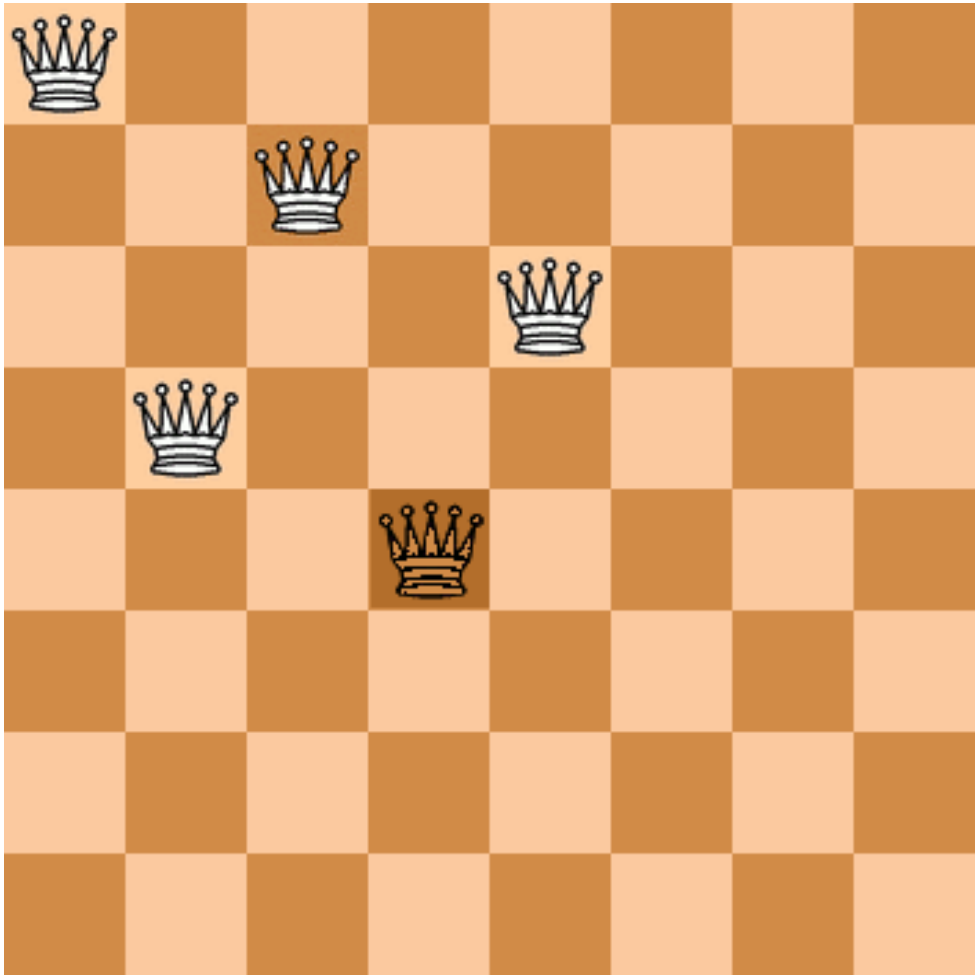
Backtracking für 8 Damen



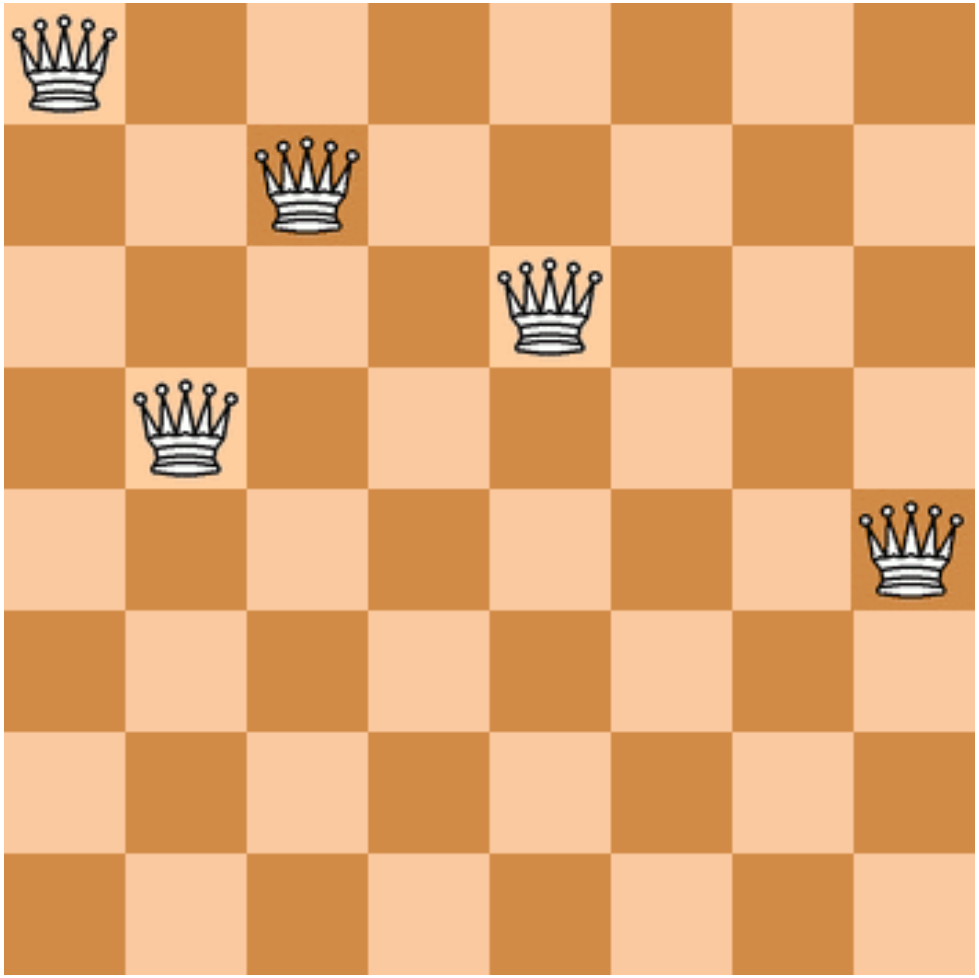
Backtracking für 8 Damen



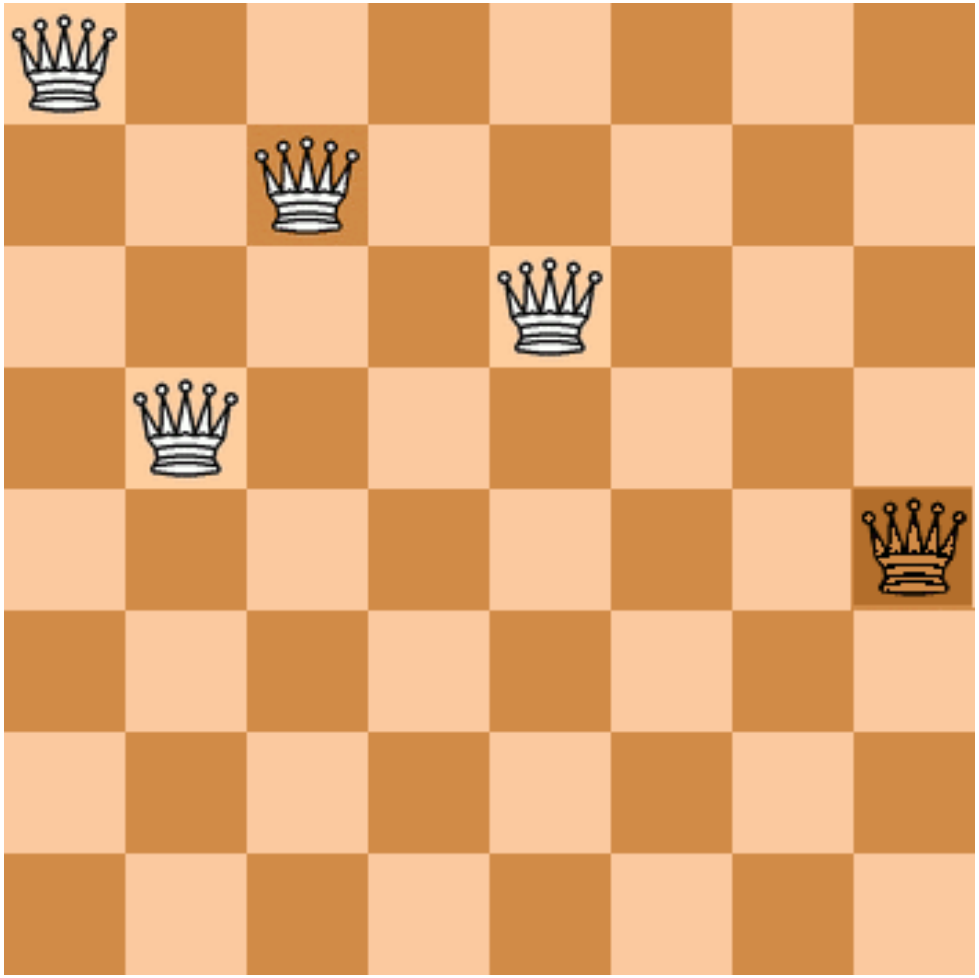
Backtracking für 8 Damen



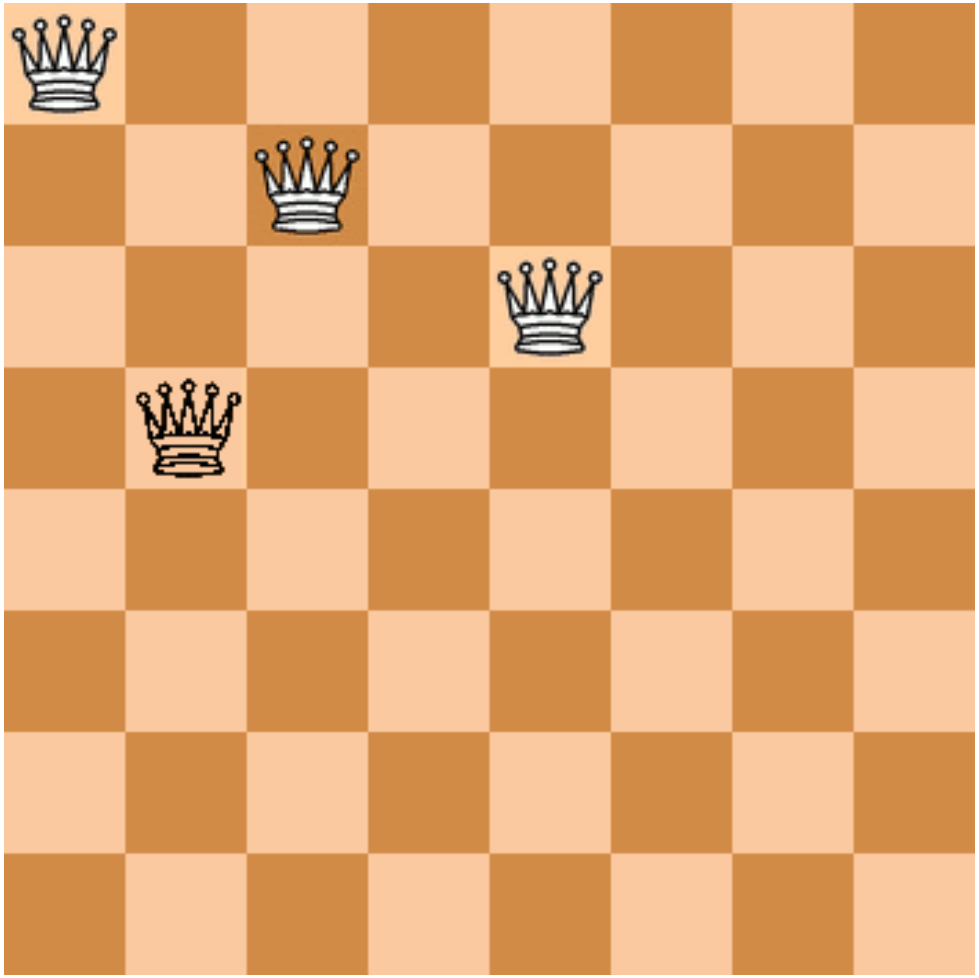
Backtracking für 8 Damen



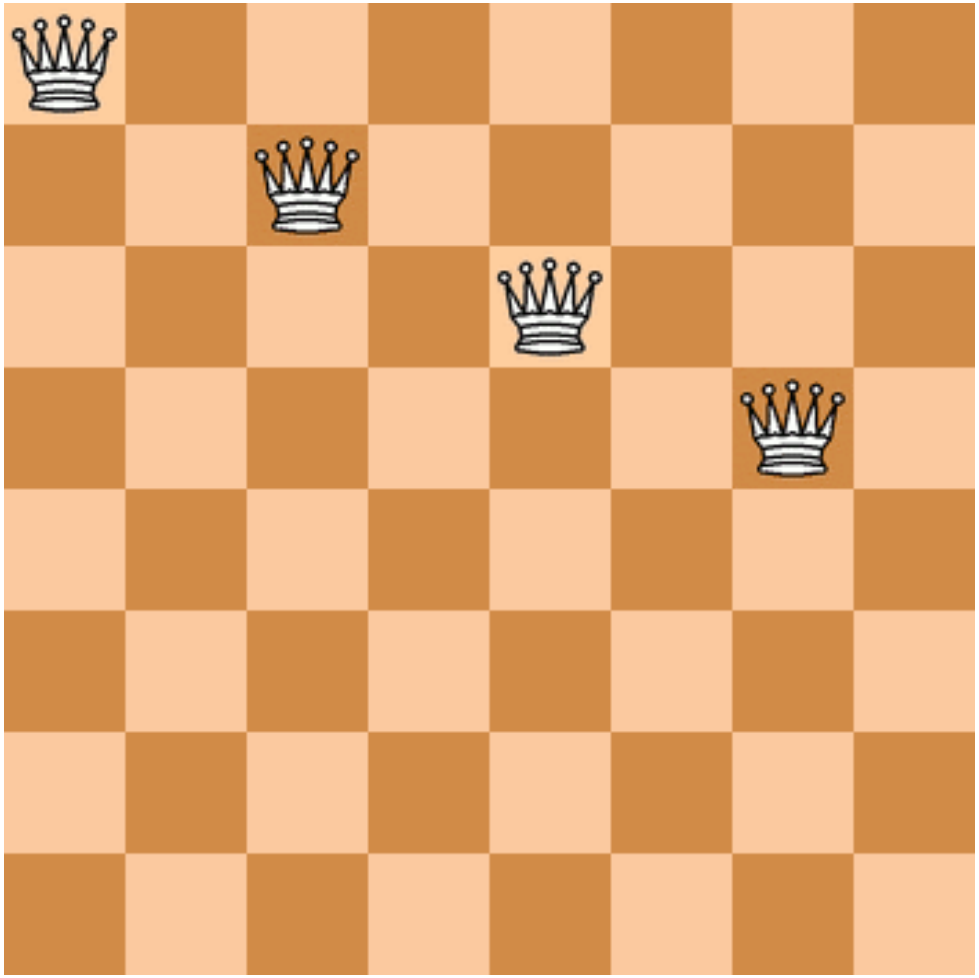
Backtracking für 8 Damen



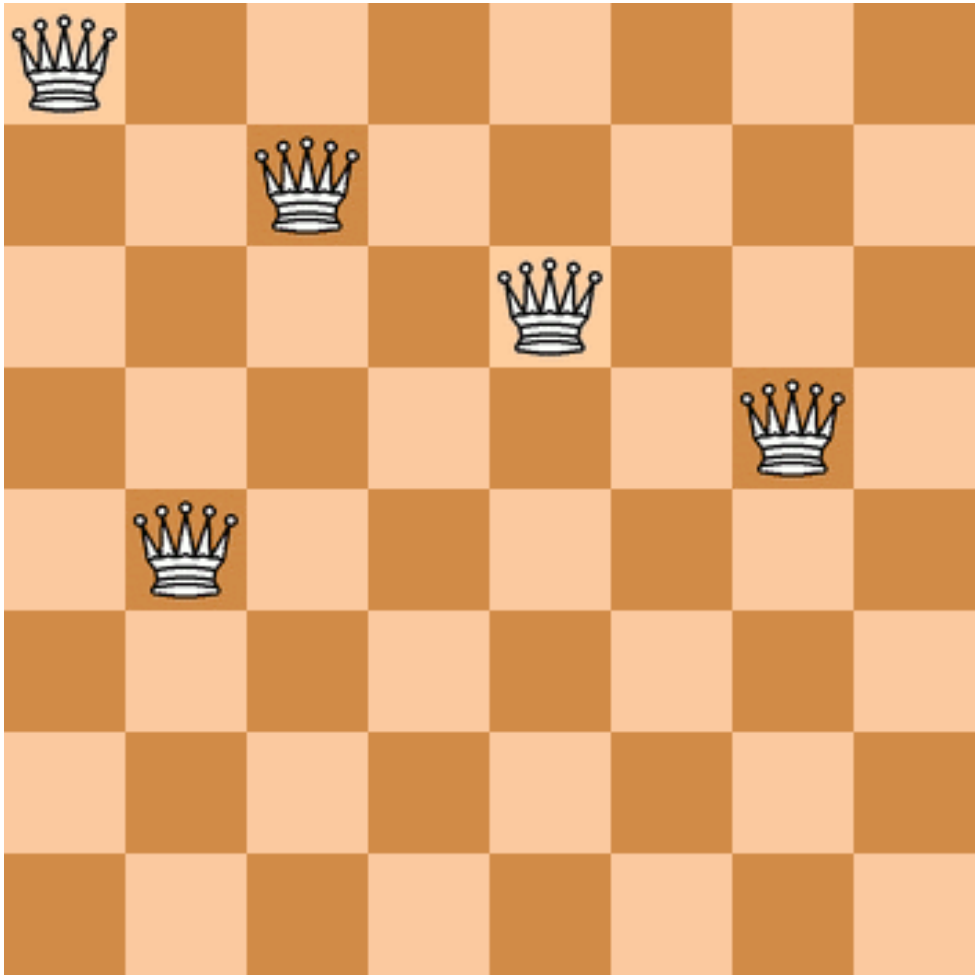
Backtracking für 8 Damen



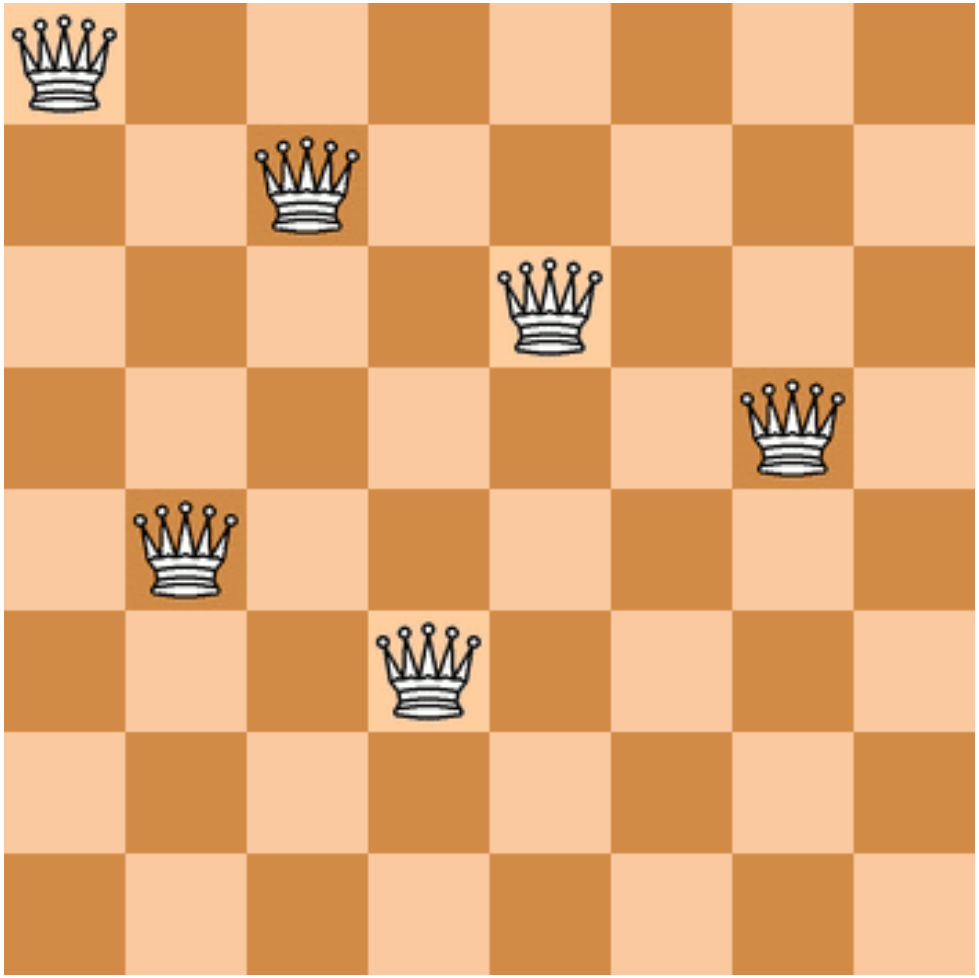
Backtracking für 8 Damen



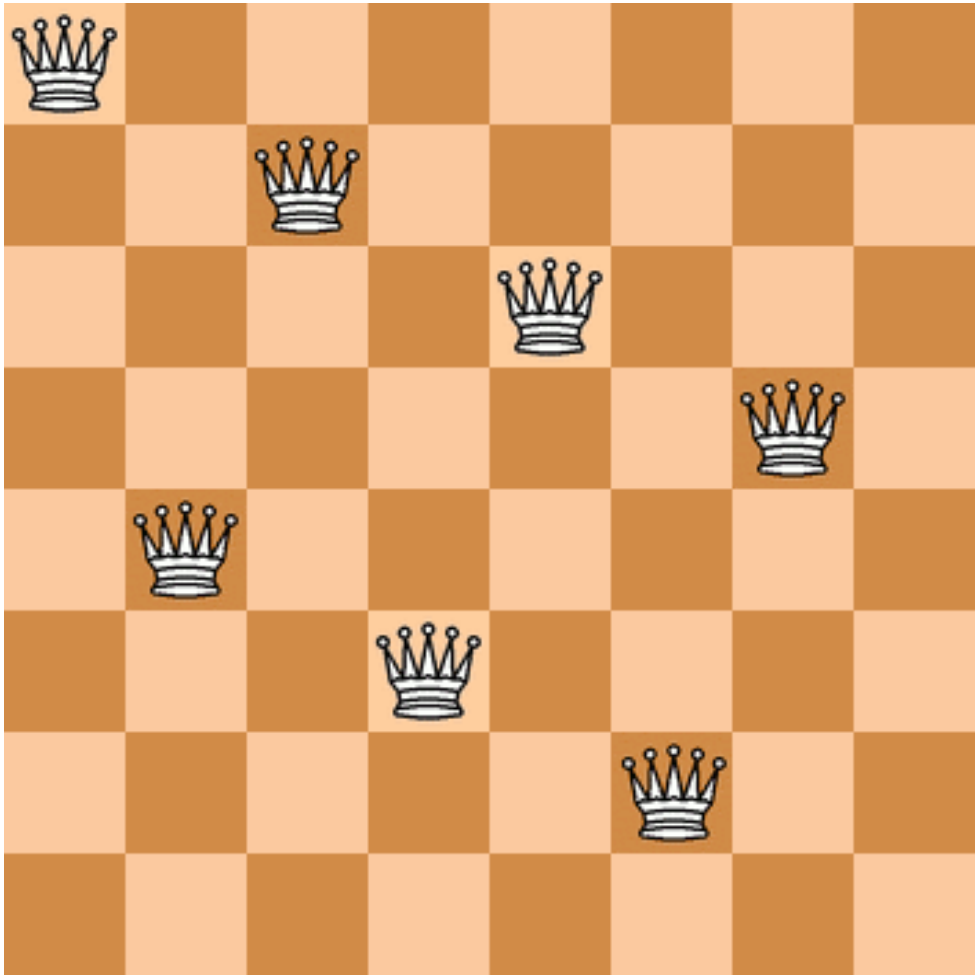
Backtracking für 8 Damen



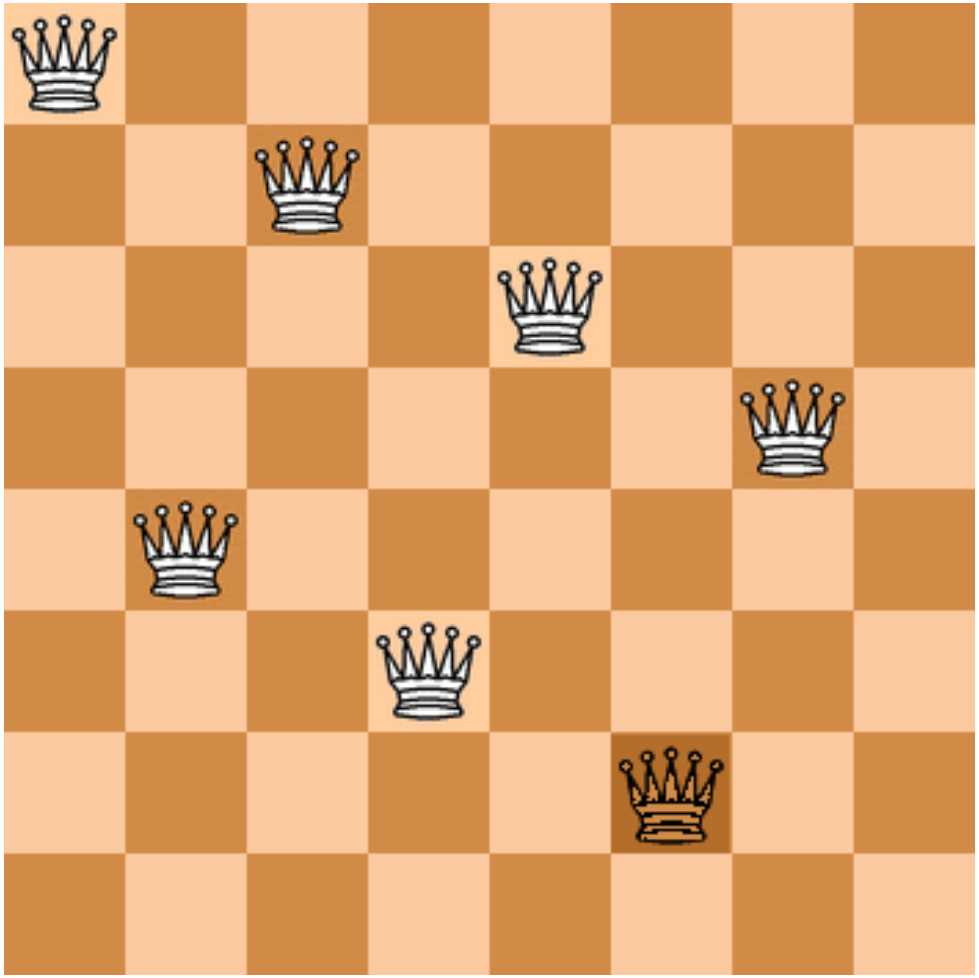
Backtracking für 8 Damen



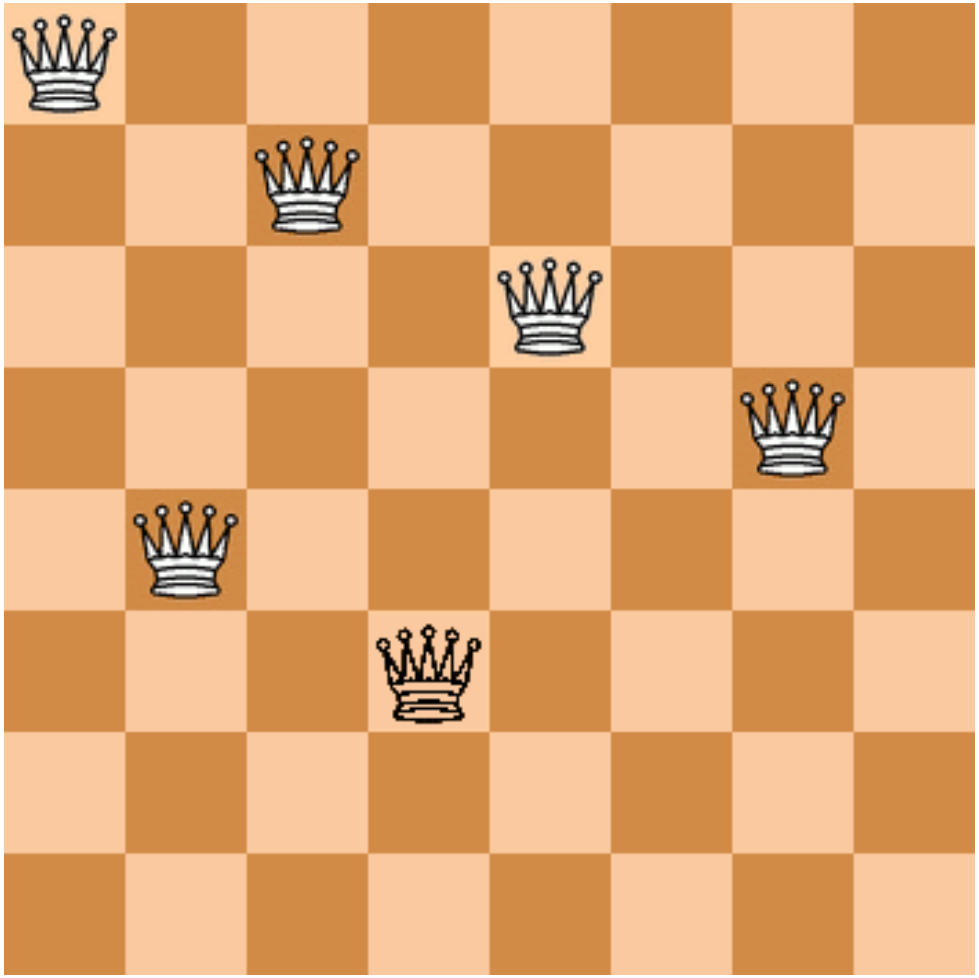
Backtracking für 8 Damen



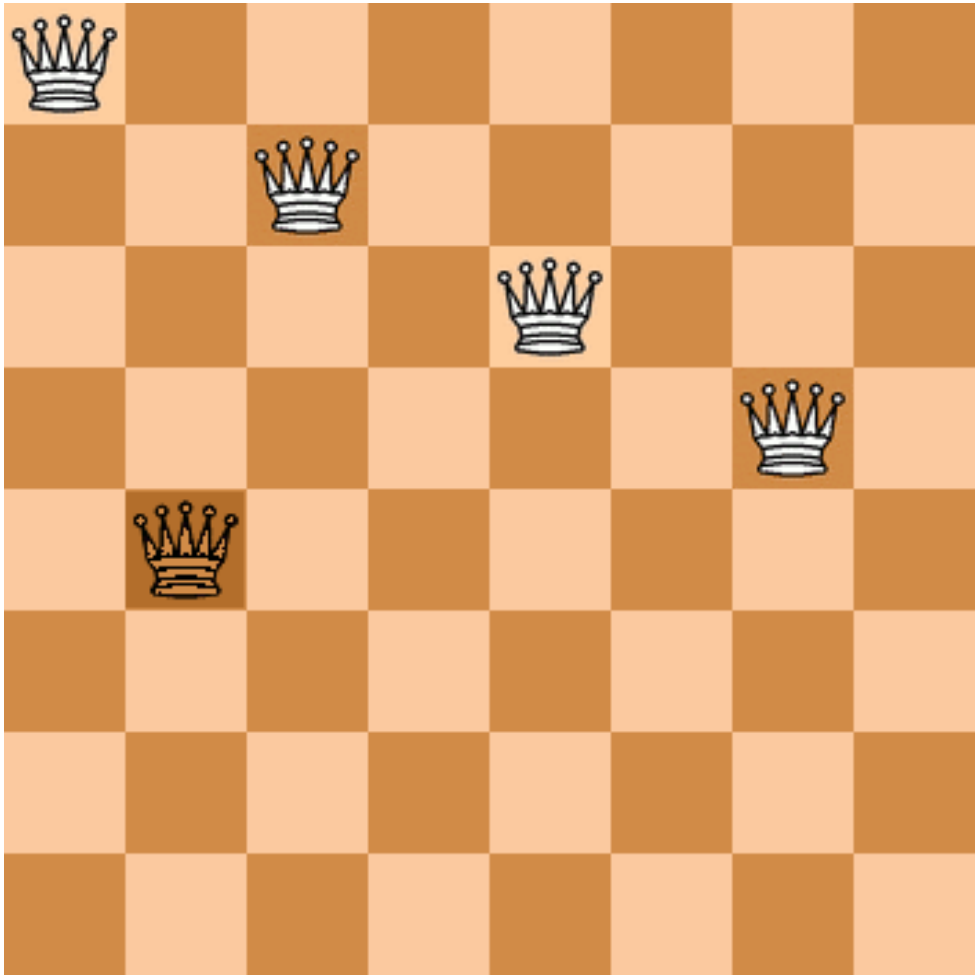
Backtracking für 8 Damen



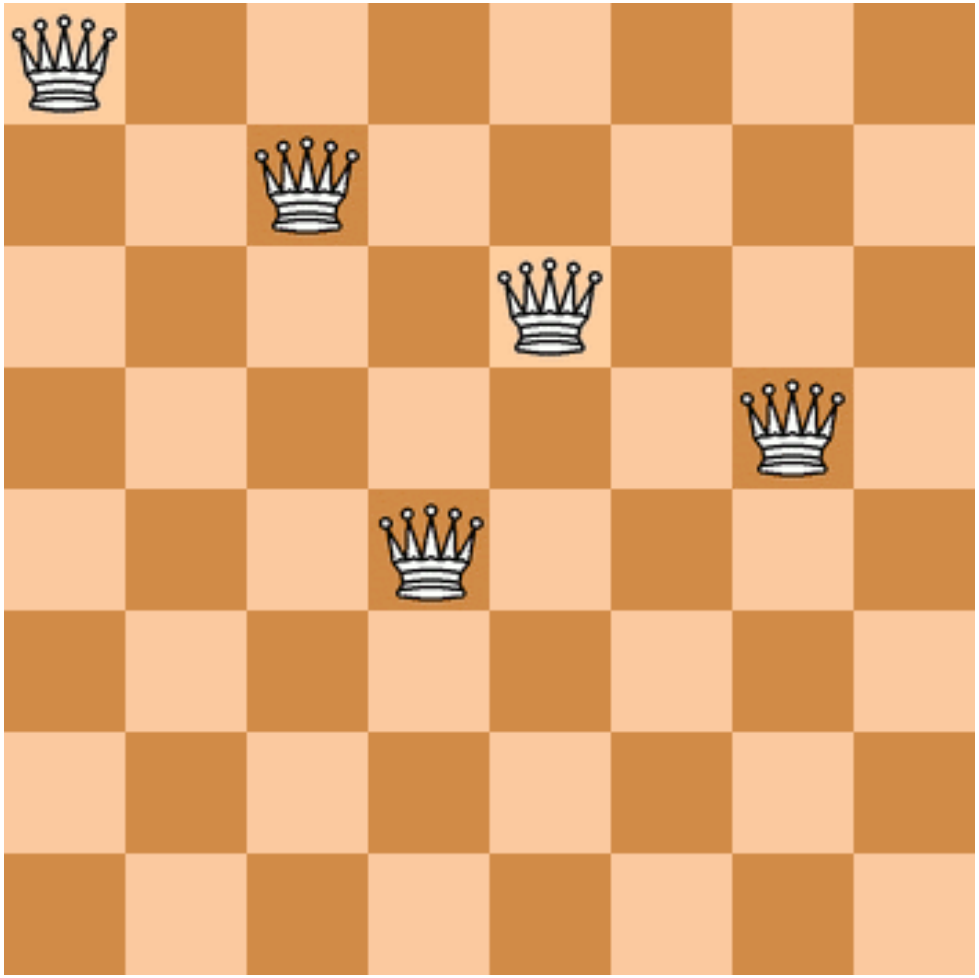
Backtracking für 8 Damen



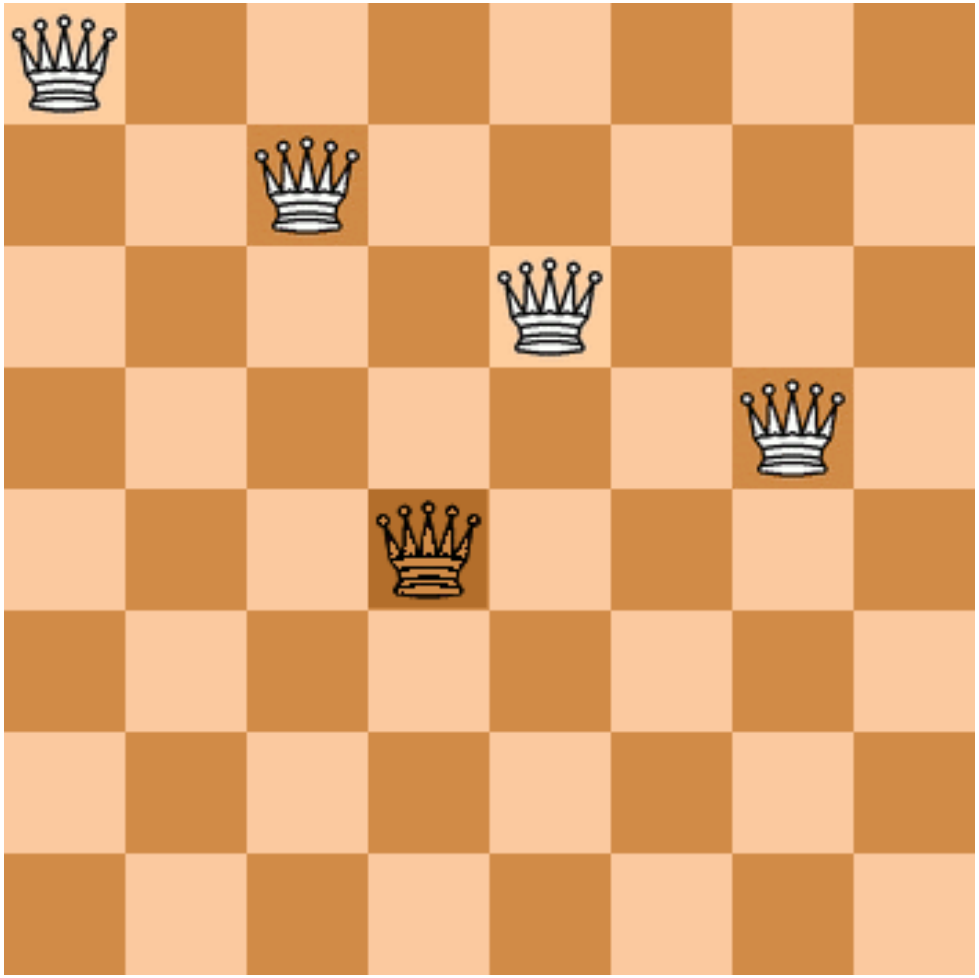
Backtracking für 8 Damen



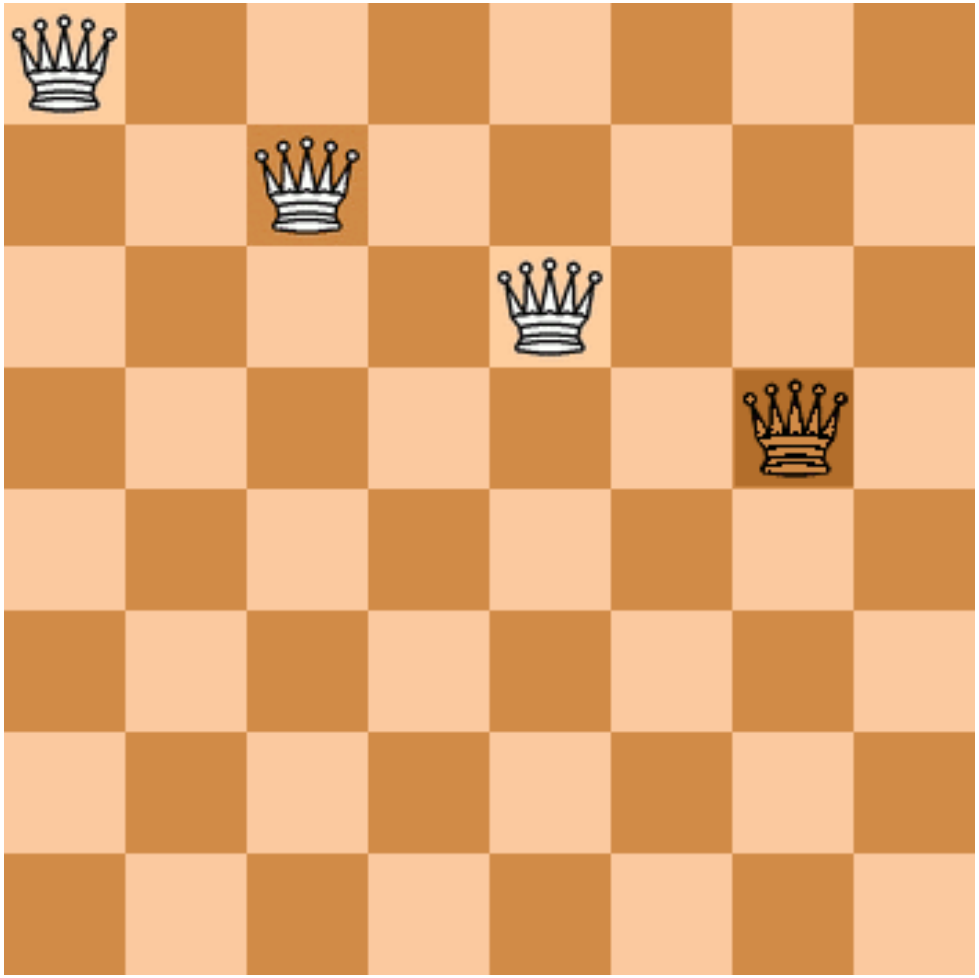
Backtracking für 8 Damen



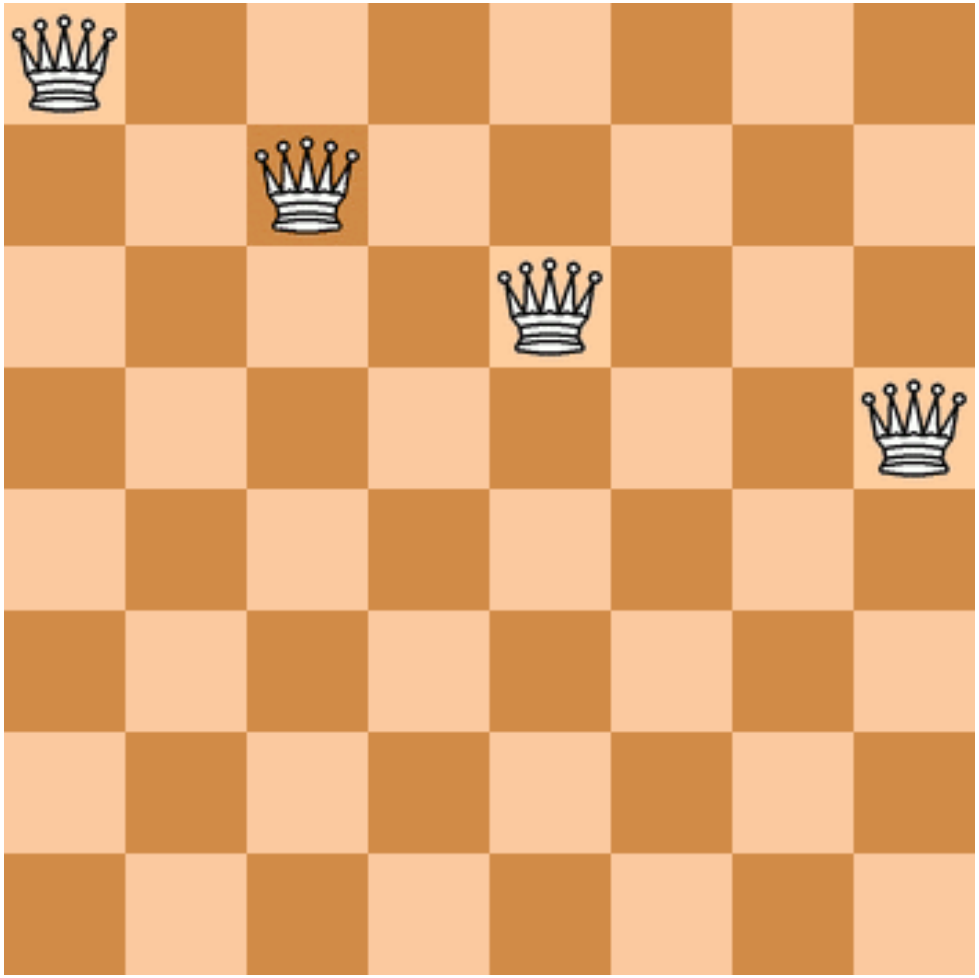
Backtracking für 8 Damen



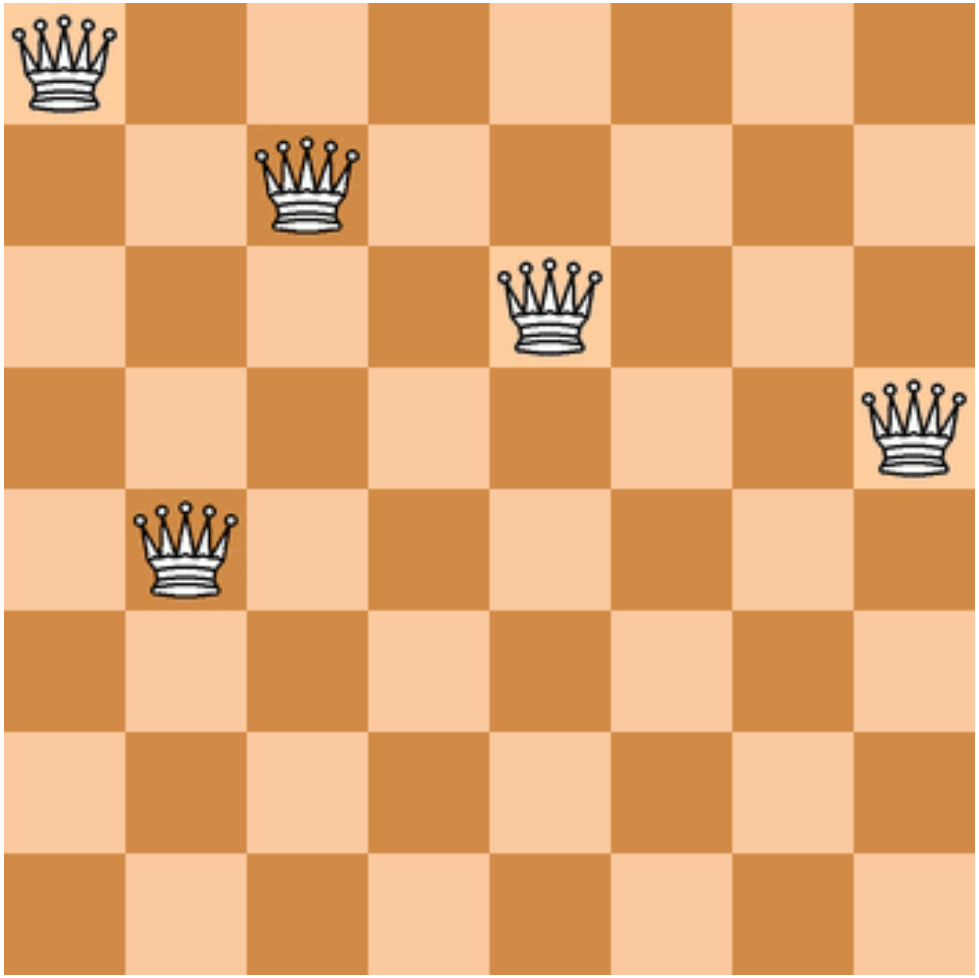
Backtracking für 8 Damen



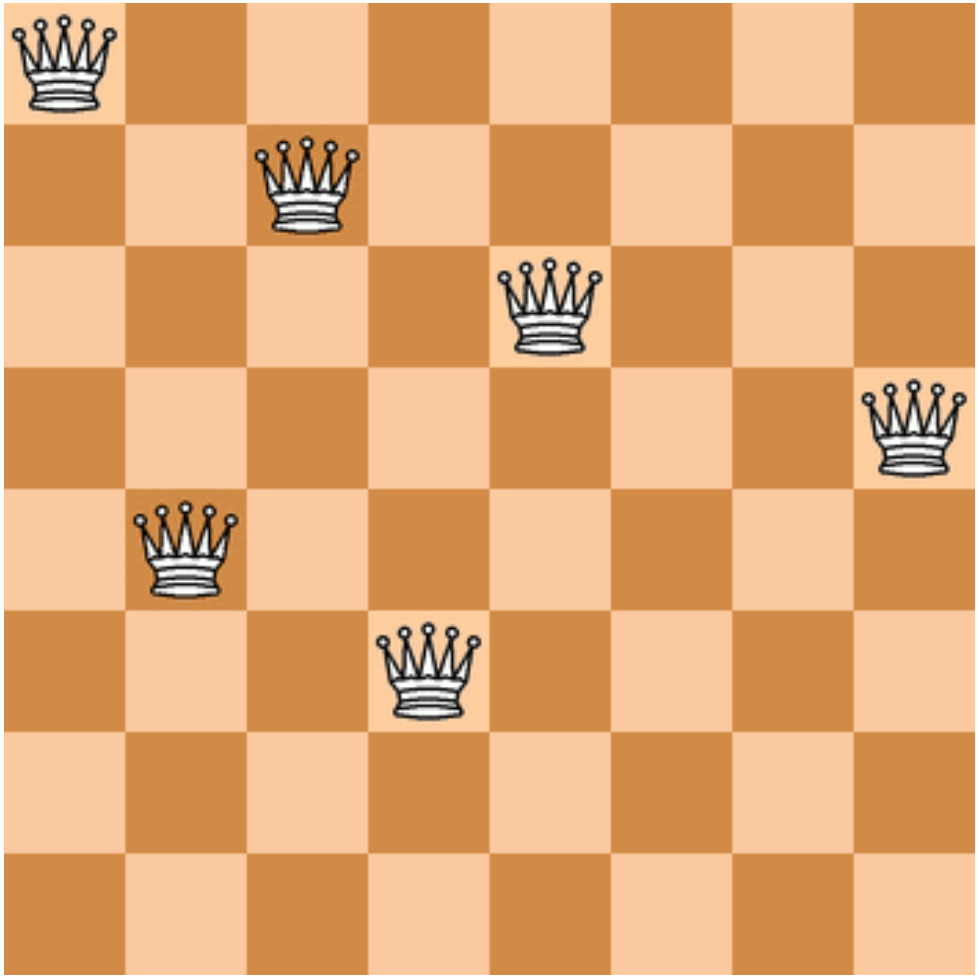
Backtracking für 8 Damen



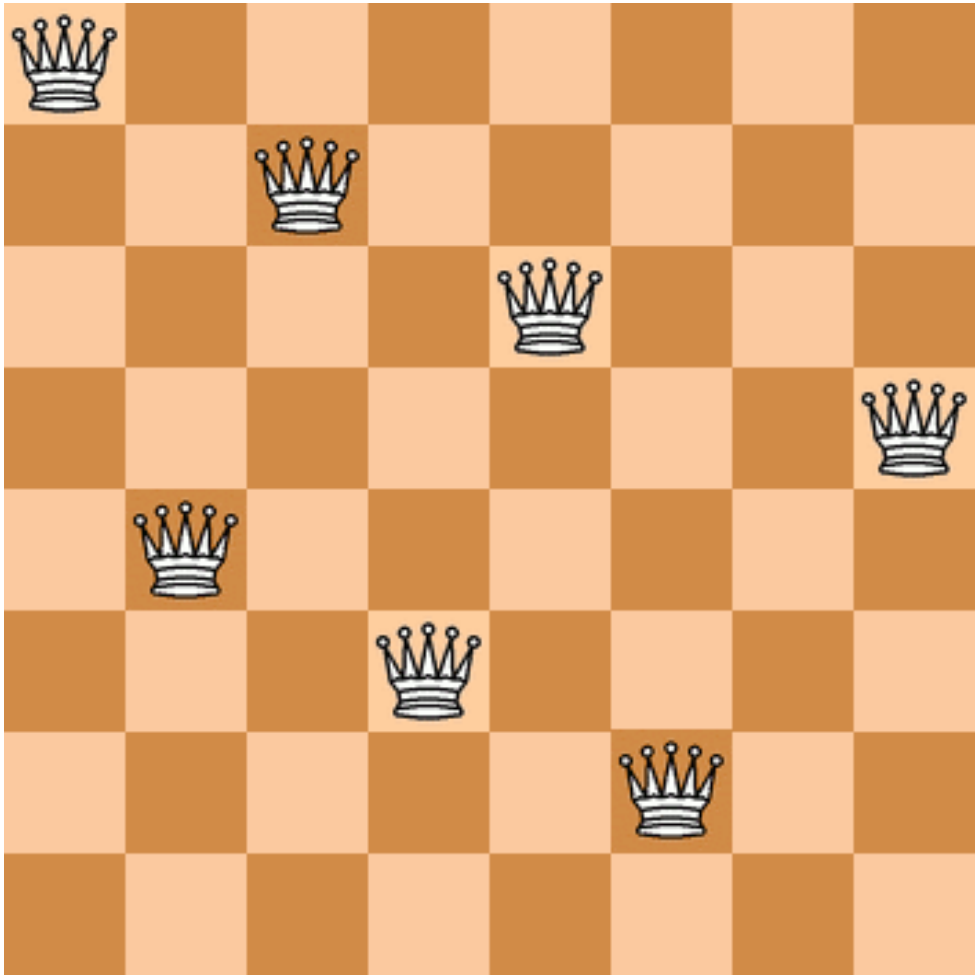
Backtracking für 8 Damen



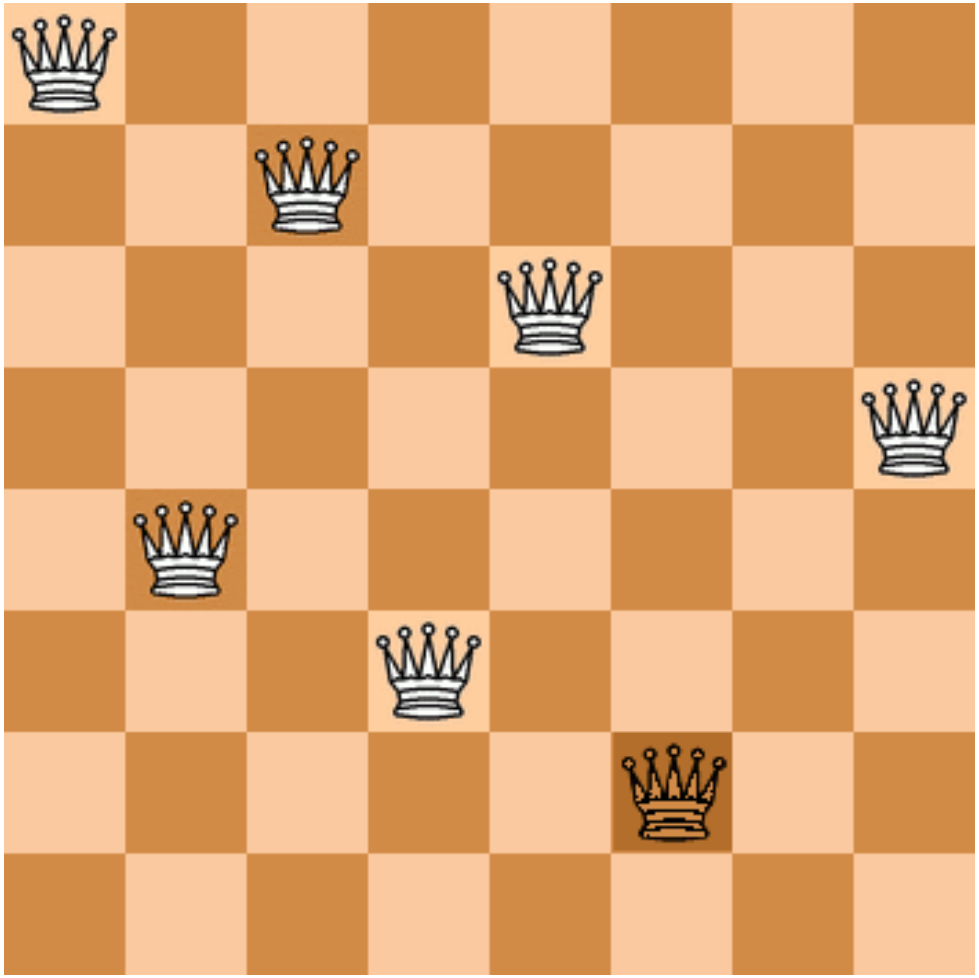
Backtracking für 8 Damen



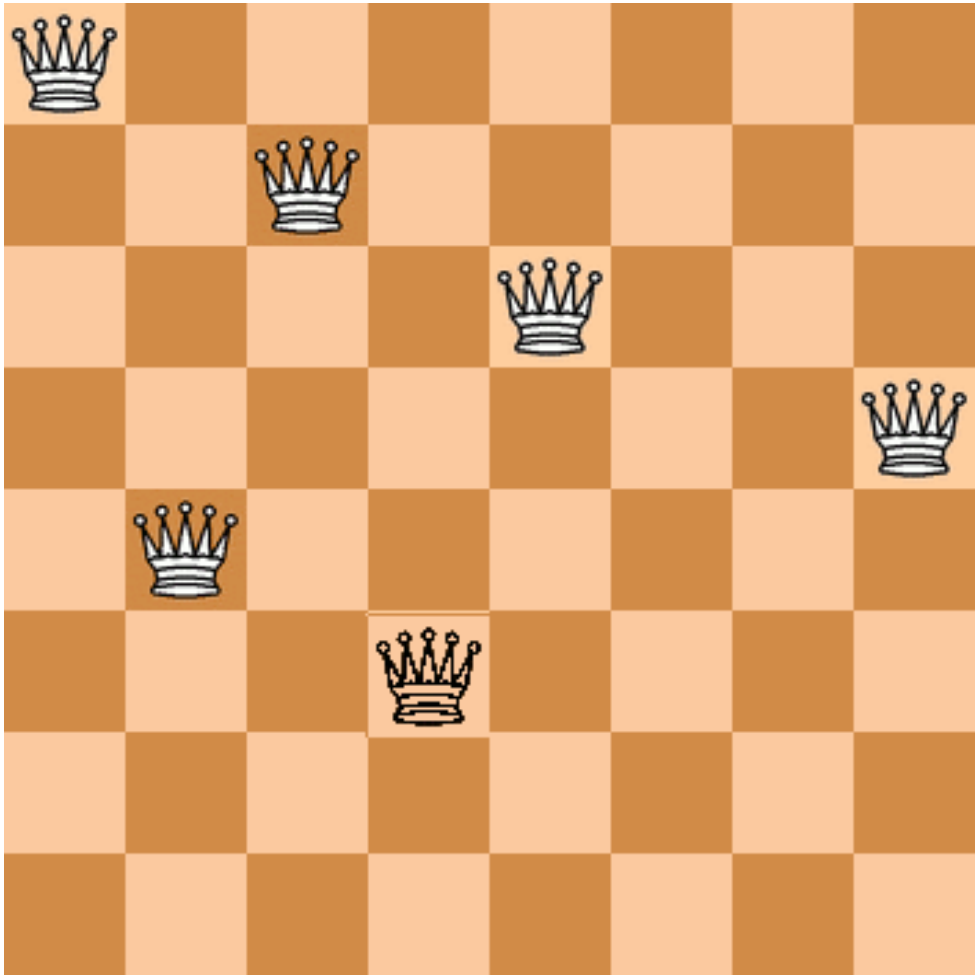
Backtracking für 8 Damen



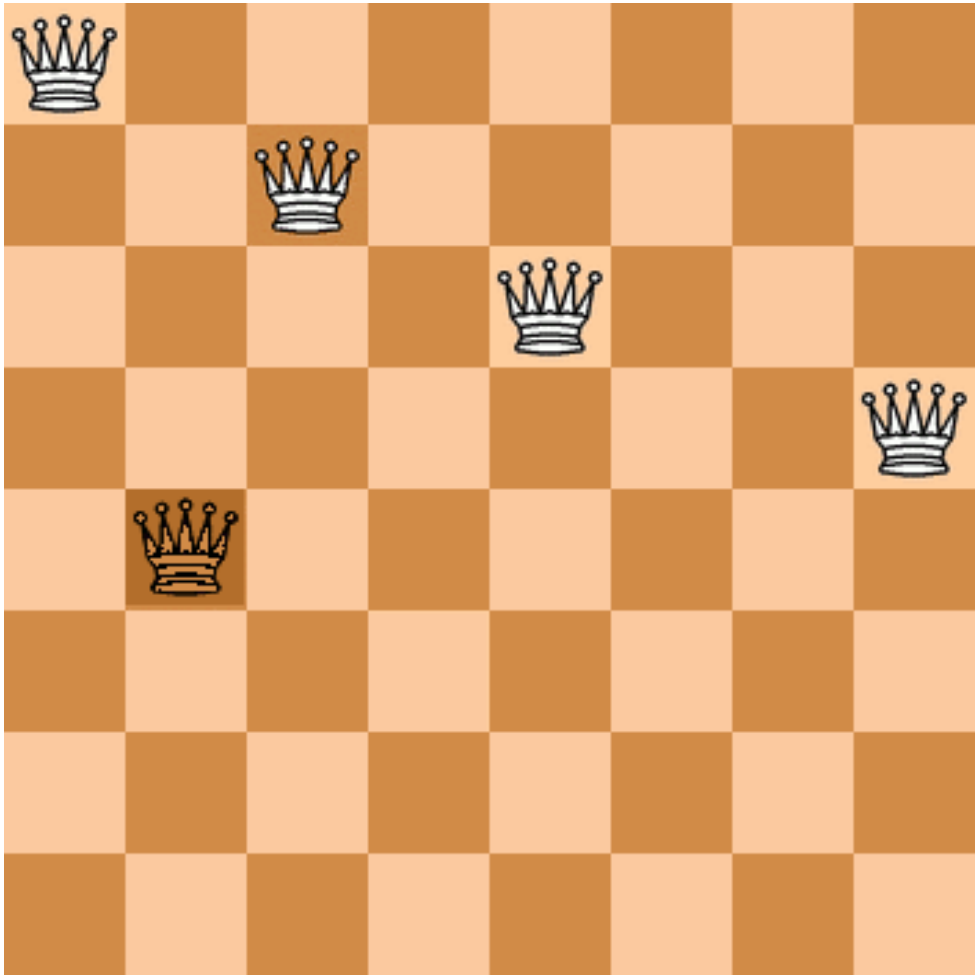
Backtracking für 8 Damen



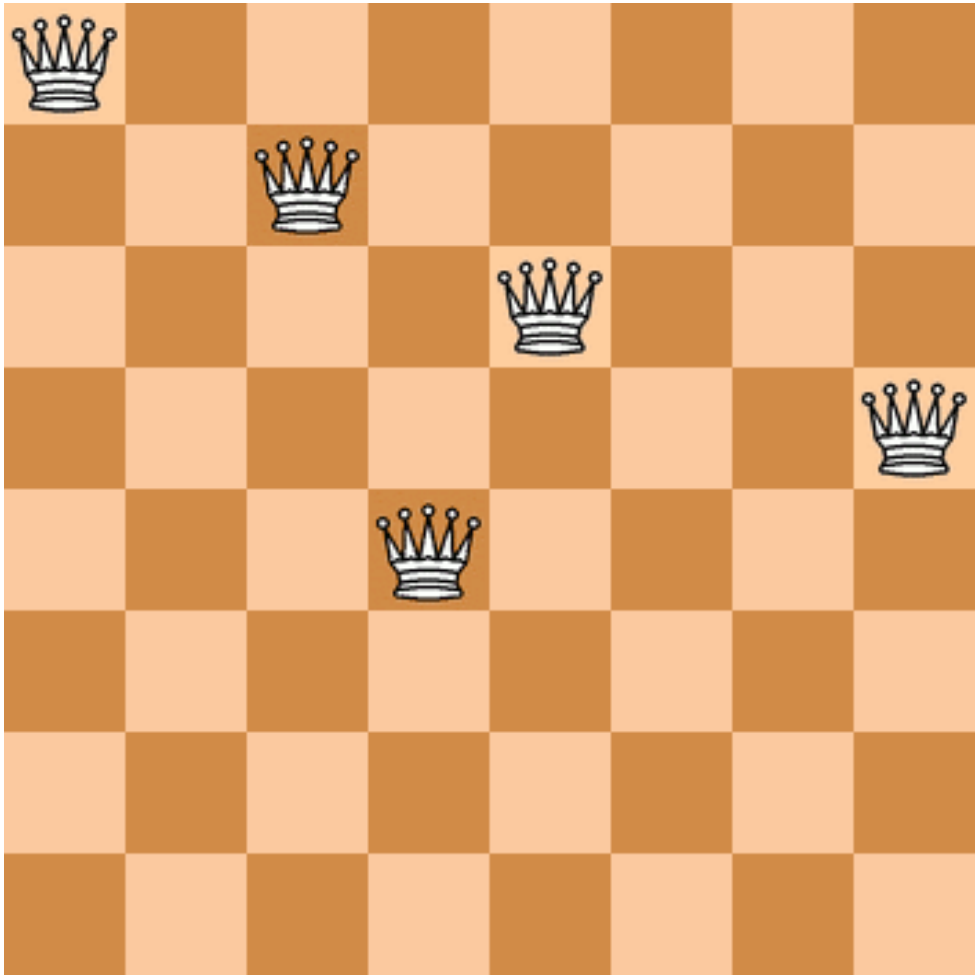
Backtracking für 8 Damen



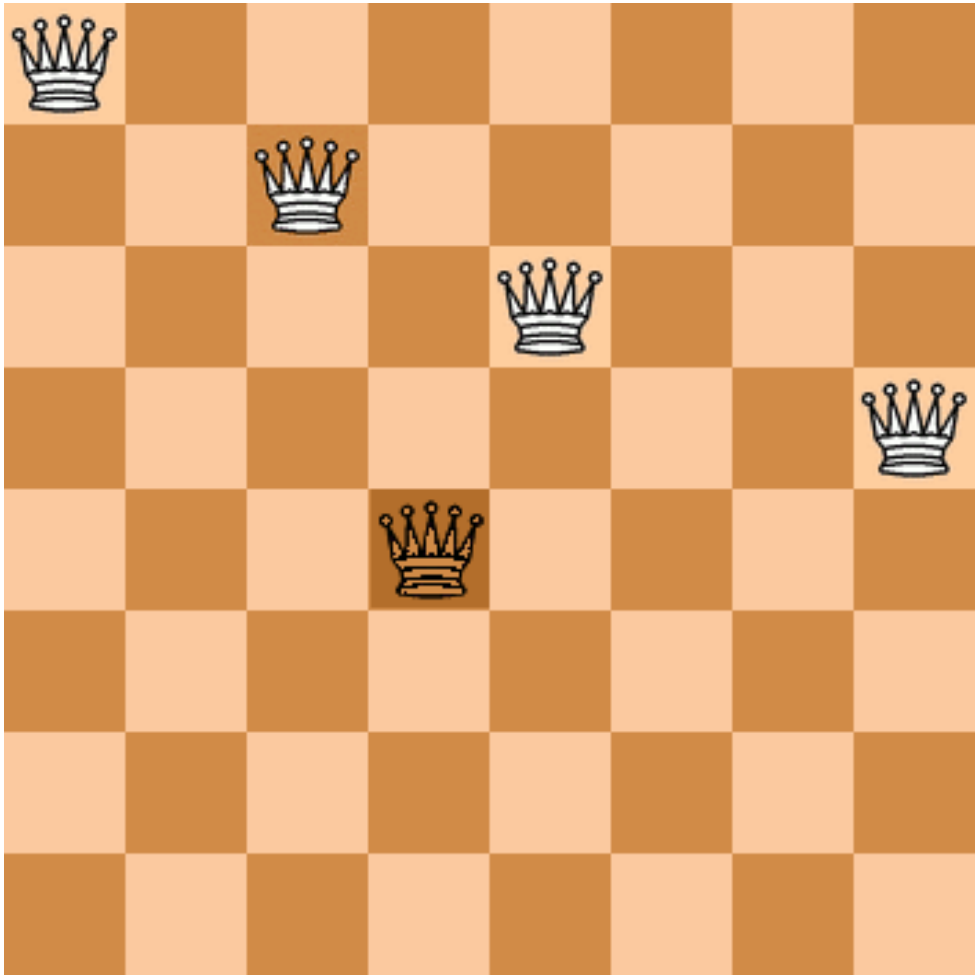
Backtracking für 8 Damen



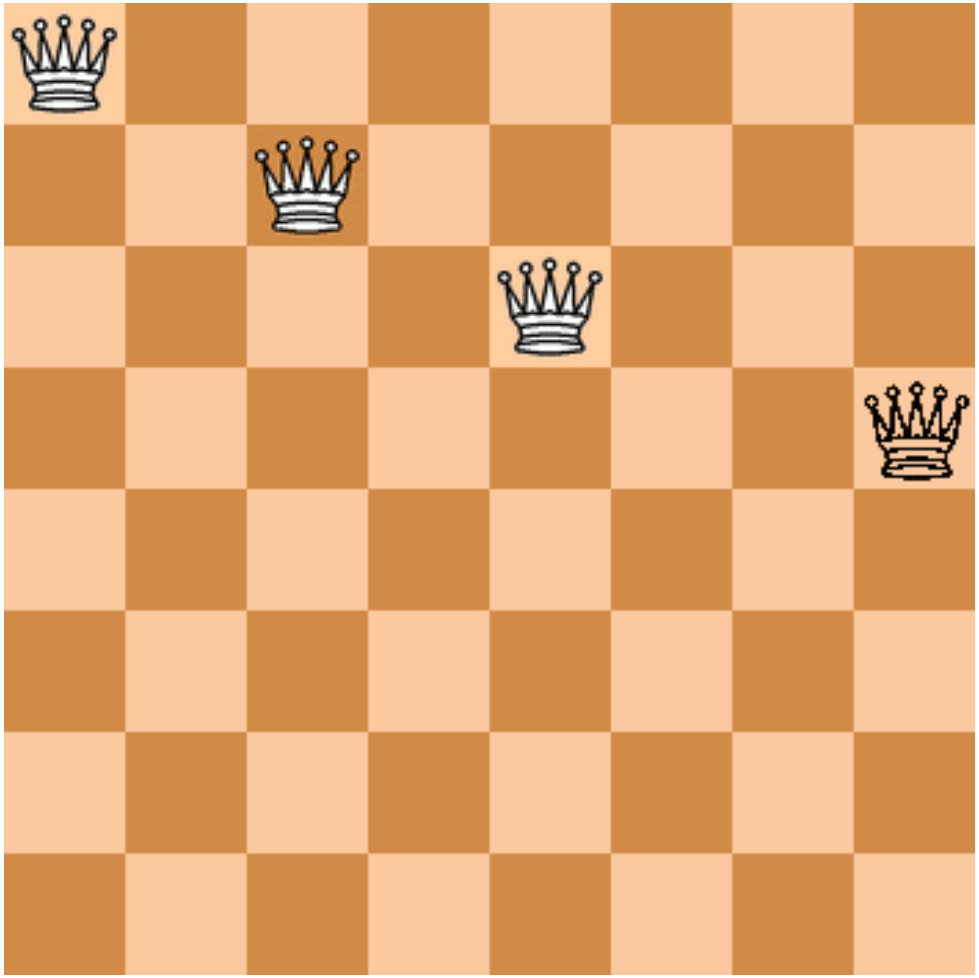
Backtracking für 8 Damen



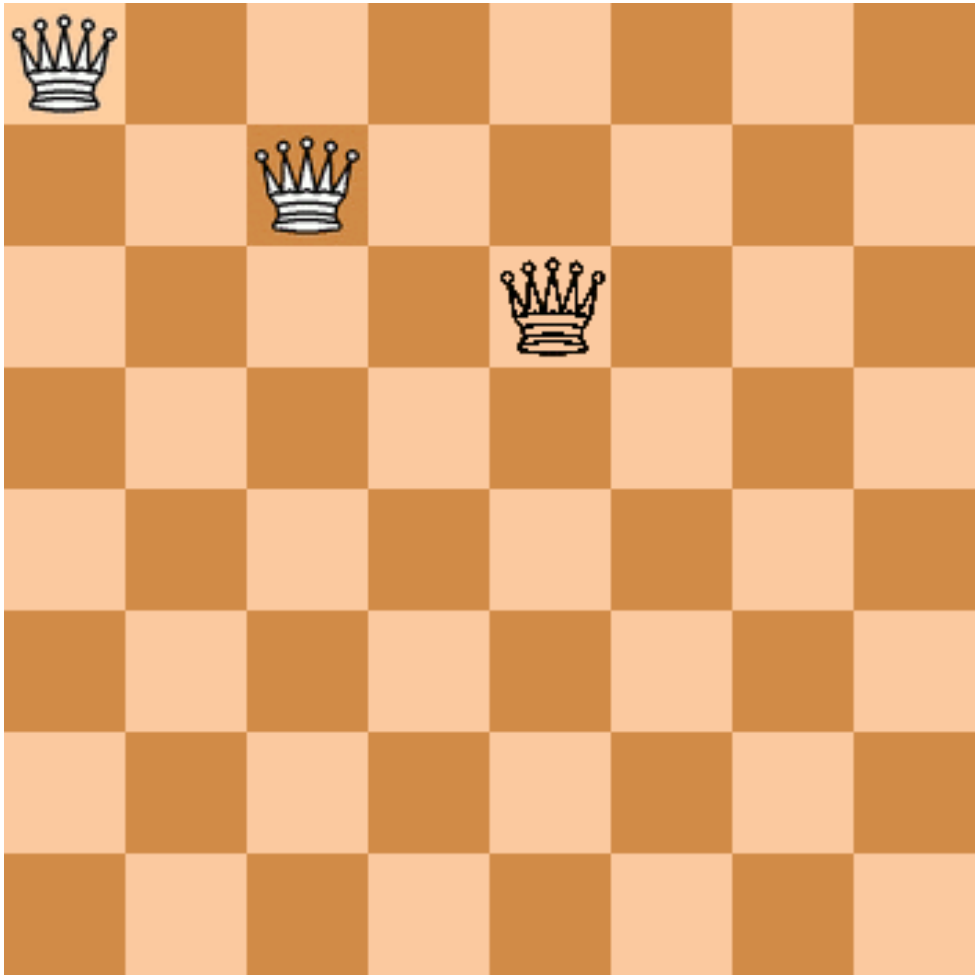
Backtracking für 8 Damen



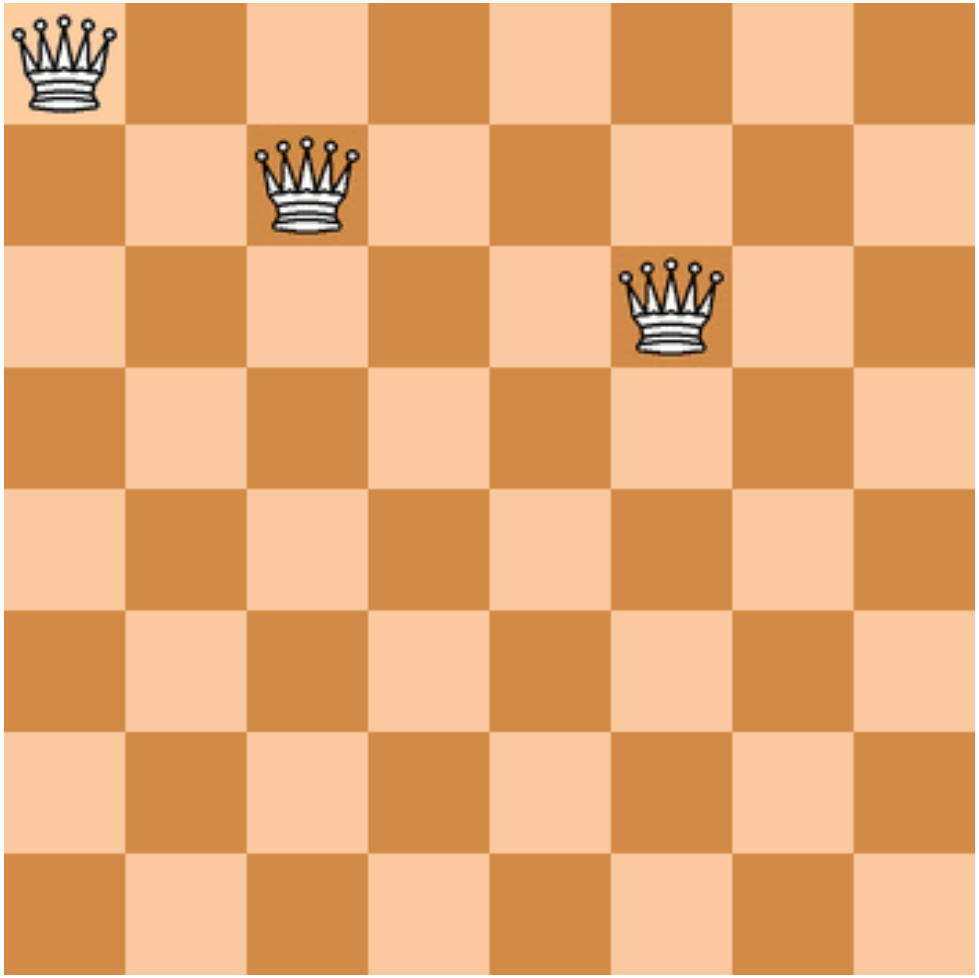
Backtracking für 8 Damen



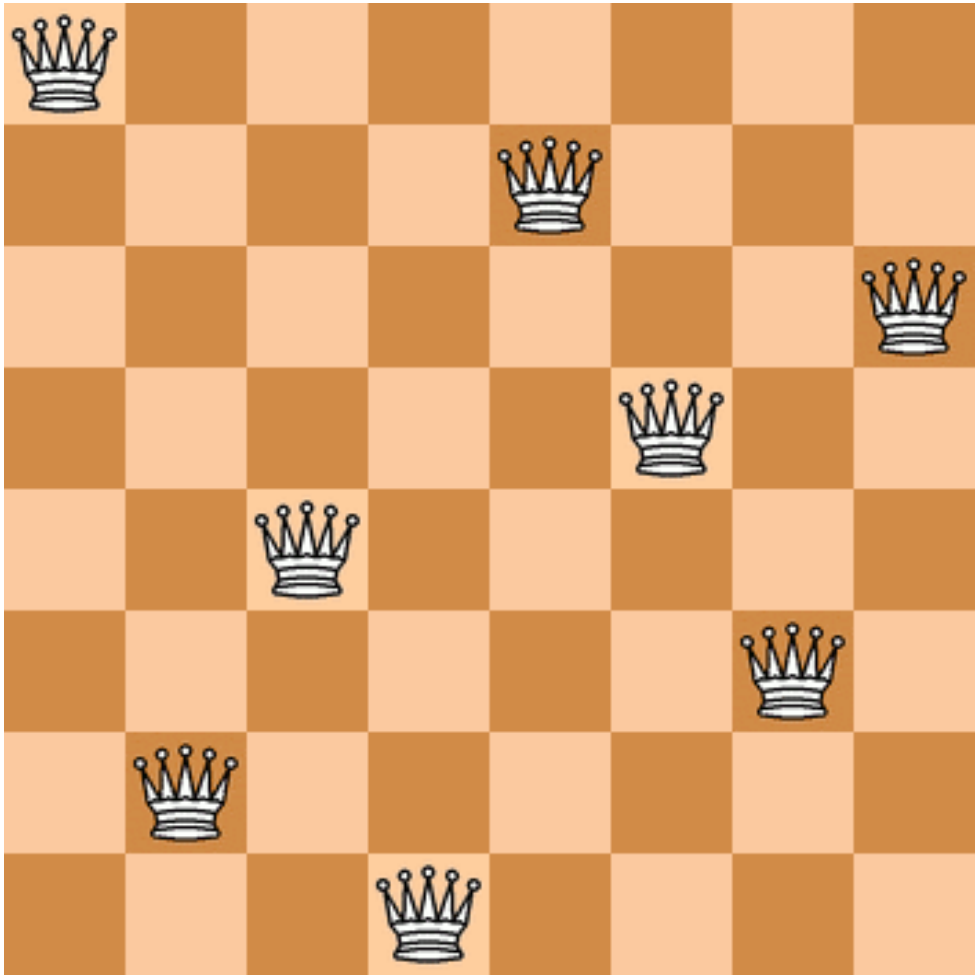
Backtracking für 8 Damen



Backtracking für 8 Damen



Backtracking für 8 Damen



Backtracking für 8 Damen - Generatoren

UNI FREIBURG

- Und wie sähe das aus, wenn wir Generatoren einsetzen wollten?

Backtracking für 8 Damen - Generatoren

UNI FREIBURG

- Und wie sähe das aus, wenn wir Generatoren einsetzen wollten?
- Statt `return`, `yield`.

Backtracking für 8 Damen - Generatoren

UNI FREIBURG

- Und wie sähe das aus, wenn wir Generatoren einsetzen wollten?
- Statt `return`, `yield`.
- Keine Fehlschläge, sondern nur die erfolgreichen Zweige weiter verfolgen!

Backtracking für 8 Damen - Generatoren

UNI FREIBURG

- Und wie sähe das aus, wenn wir Generatoren einsetzen wollten?
- Statt `return`, `yield`.
- Keine Fehlschläge, sondern nur die erfolgreichen Zweige weiter verfolgen!
- Aufrufe nur in `for`-Schleifen.

Backtracking für 8 Damen - Generatoren

UNI FREIBURG

- Und wie sähe das aus, wenn wir Generatoren einsetzen wollten?
- Statt `return`, `yield`.
- Keine Fehlschläge, sondern nur die erfolgreichen Zweige weiter verfolgen!
- Aufrufe nur in `for`-Schleifen.
- Verschiedene Lösungen unterscheidbar machen (Leerzeile nach jeder Lösung).

Backtracking für Sudokus (1): Adressierung der Felder

UNI FREIBURG

Die Formalisierung der Constraints ist aufwändig:

sudoku.py (1)

```
def cross(A, B):
    return [a+b for a in A for b in B]
digits   = '123456789'
digits0p = digits + '0.'
rows     = 'ABCDEFGHI'
cols     = digits
squares  = cross(rows, cols)
unitlist = ([cross(rows, c) for c in cols] +
            [cross(r, cols) for r in rows] +
            [cross(rs, cs) for rs in ('ABC','DEF','GHI')
                           for cs in ('123','456','789')])
units = dict((s, [u for u in unitlist if s in u])
             for s in squares) # s -> all units of s
peers = dict((s, set(sum(units[s],[]))-set([s]))
             for s in squares) # s -> set of peers of s
```

Sudoku-Backtracking (2): Belegung und Constraints

UNI FREIBURG

- Belegungen werden wie im Falle der Färbbarkeit durch ein `dict` repräsentiert.
- Die CSP-Variablen sind durch die Liste `squares` gegeben:
  `['A1', 'A2', ..., 'A9', 'B1', 'B2', ..., 'I9']`
- `unitlist` ist eine Liste, deren Elemente Listen sind, die jeweils alle Felder einer Gruppe enthalten:
  `[['A1', 'B1', ..., 'I1'], ['A2', 'B2', ..., 'I2'], ..., ['A1', 'A2', ..., 'A9'], ..., ['A1', 'A2', 'A3', 'B1', 'B2', ...'C3'], ...]`
- `units` spezifiziert für jedes Feld, in welchen Gruppen es Mitglied ist:
  `{ 'A1': [['A1', ..., 'I1'], ['A1', ..., 'A9'], ['A1', ..., 'C3']], ...}`
- `peers` spezifiziert für jedes Feld die Menge der Peers:
  `{'D8': {'E9', 'E8', 'D9', 'G8', 'D2', 'D3', 'D1', 'D6', 'D7', ...}, ...}`

Sudoku-Backtracking (3): Parsing

UNI FREIBURG

- Wir wollen ja verschiedene Sudokus lösen.

Sudoku-Backtracking (3): Parsing

UNI FREIBURG

- Wir wollen ja verschiedene Sudokus lösen.
- D.h. wir müssen die Aufgabe parsen und in eine interne Struktur überführen.

Sudoku-Backtracking (3): Parsing

UNI FREIBURG

- Wir wollen ja verschiedene Sudokus lösen.
- D.h. wir müssen die Aufgabe parsen und in eine interne Struktur überführen.
- Aufgabe besteht aus 81 Zeichen $0-9$ und '.', wobei 0 und '.' für ein leeres Feld stehen.

Sudoku-Backtracking (3): Parsing

UNI FREIBURG

- Wir wollen ja verschiedene Sudokus lösen.
- D.h. wir müssen die Aufgabe parsen und in eine interne Struktur überführen.
- Aufgabe besteht aus 81 Zeichen $0-9$ und '.', wobei 0 und '.' für ein leeres Feld stehen.
- Alle anderen Zeichen werden ignoriert. D.h. wir können die Aufgabe auch als Tabelle angeben.

Sudoku-Backtracking (3): Parsing

UNI FREIBURG

- Wir wollen ja verschiedene Sudokus lösen.
- D.h. wir müssen die Aufgabe parsen und in eine interne Struktur überführen.
- Aufgabe besteht aus 81 Zeichen $0-9$ und '.', wobei 0 und '.' für ein leeres Feld stehen.
- Alle anderen Zeichen werden ignoriert. D.h. wir können die Aufgabe auch als Tabelle angeben.

sudoku.py (2)

```
def parse_grid(grid):
  values = dict()
  for s,d in (zip(squares, [c for c in grid
                            if c in digits0p])):
    if d in digits and not assign(values, s, d):
      return False
  return values
```

Sudoku-Backtracking (3): Ausgabe

UNI FREIBURG

- Die Lösungen sollen natürlich auch dargestellt werden.

Sudoku-Backtracking (3): Ausgabe

UNI FREIBURG

- Die Lösungen sollen natürlich auch dargestellt werden.
- `display` gibt eine Belegung aus.

Sudoku-Backtracking (3): Ausgabe

UNI FREIBURG

- Die Lösungen sollen natürlich auch dargestellt werden.
- `display` gibt eine Belegung aus.

sudoku.py (3)

```
def display(values):
    "Display values as a 2-D grid."
    if not values:
        print("Empty grid")
        return
    line = '+'.join(['-'*6]*3)
    for r in rows:
        print(''.join(values.get(r+c,'.')+ ' ' +
                      ('|' if c in '36' else '')
                    for c in cols))
        if r in 'CF': print(line)
    print()
```

Sudoku-Backtracking (4): `assign`

UNI FREIBURG

- Die Zuweisung funktioniert wieder ähnlich wie in den beiden anderen Fällen.

Sudoku-Backtracking (4): `assign`

UNI FREIBURG

- Die Zuweisung funktioniert wieder ähnlich wie in den beiden anderen Fällen.
- D.h. es werden die Constraints überprüft und im Erfolgsfall die erweiterte Belegung zurück gegeben.

Sudoku-Backtracking (4): `assign`

UNI FREIBURG

- Die Zuweisung funktioniert wieder ähnlich wie in den beiden anderen Fällen.
- D.h. es werden die Constraints überprüft und im Erfolgsfall die erweiterte Belegung zurück gegeben.
- Ansonsten wird False zurück gegeben.

Sudoku-Backtracking (4): assign

UNI FREIBURG

- Die Zuweisung funktioniert wieder ähnlich wie in den beiden anderen Fällen.
- D.h. es werden die Constraints überprüft und im Erfolgsfall die erweiterte Belegung zurück gegeben.
- Ansonsten wird False zurück gegeben.

sudoku.py (4)

```
def assign(values, s, d):
    "Try to assign value d to square s"
    if s not in values and all(values[p] != d
                               for p in values
                               if p in peers[s]):
        values[s] = d
        return values
    return False
```

Sudoku-Backtracking (5): Rekursive Suche

UNI FREIBURG

- Völlig analog zu den beiden vorherigen Fällen:

Sudoku-Backtracking (5): Rekursive Suche

UNI FREIBURG

- Völlig analog zu den beiden vorherigen Fällen:

sudoku.py (5)

```
def search(values):
    "Search for solution"
    if not values: return False # failed earlier
    s = some(s for s in squares if s not in values)
    if not s: return values
    return some(search(assign(values.copy(), s, d))
                for d in digits)

import time

def timed_search(grid):
    start = time.process_time()
    search(parse_grid(grid))
    return time.process_time() - start
```

Sudoku-Backtracking (6): In Aktion ...

UNI FREIBURG

Python-Interpreter

```
>>> grid1='''003020600 900305001 001806400 008102900
... 700000008 006708200 002609500 800203009
... 005010300'''
>>> display(search(parse_grid(grid1)))
```

Sudoku-Backtracking (6): In Aktion ...

UNI FREIBURG

Python-Interpreter

```
>>> grid1='''003020600 900305001 001806400 008102900
... 700000008 006708200 002609500 800203009
... 005010300'''
>>> display(search(parse_grid(grid1)))
4 8 3 |9 2 1 |6 5 7
9 6 7 |3 4 5 |8 2 1
2 5 1 |8 7 6 |4 9 3
------+------+------
5 4 8 |1 3 2 |9 7 6
7 2 9 |5 6 4 |1 3 8
1 3 6 |7 9 8 |2 4 5
------+------+------
3 7 2 |6 8 9 |5 1 4
8 1 4 |2 5 3 |7 6 9
6 9 5 |4 1 7 |3 8 2
```

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.01417400000013913
```

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.01417400000013913
>>> timed_search(grid2)
660.3158369999999
```

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.01417400000013913
>>> timed_search(grid2)
660.3158369999999
>>> timed_search(hard1)
24.770020000000002
```

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.01417400000013913
>>> timed_search(grid2)
660.3158369999999
>>> timed_search(hard1)
24.770020000000002
>>> timed_search(hard2)
0.693335000000161
```

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.01417400000013913
>>> timed_search(grid2)
660.3158369999999
>>> timed_search(hard1)
24.770020000000002
>>> timed_search(hard2)
0.693335000000161
>>> timed_search(hard3)
28.898888999999826
```

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.01417400000013913
>>> timed_search(grid2)
660.3158369999999
>>> timed_search(hard1)
24.770020000000002
>>> timed_search(hard2)
0.693335000000161
>>> timed_search(hard3)
28.898888999999826
```

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.0141740000013913
>>> timed_search(grid2)
660.3158369999999
>>> timed_search(hard1)
24.770020000000002
>>> timed_search(hard2)
0.693335000000161
>>> timed_search(hard3)
28.898888999999826
```

- `hard1` und `hard2` sind zwei von dem finnischen Mathematiker Arto Inkala entworfene Sudokus, die er als „die schwersten“ Sudokus bezeichnet.

Sudoku-Backtracking (6): Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.0141740000013913
>>> timed_search(grid2)
660.3158369999999
>>> timed_search(hard1)
24.770020000000002
>>> timed_search(hard2)
0.693335000000161
>>> timed_search(hard3)
28.898888999999826
```

- `hard1` und `hard2` sind zwei von dem finnischen Mathematiker Arto Inkala entworfene Sudokus, die er als „die schwersten“ Sudokus bezeichnet.
- `hard3` (von Peter Norvig) ist auch für Computer eine harte Nuss; aber es ist kein Sudoku, da nicht eindeutig.

Backtracking: Fazit

UNI FREIBURG

- Mit Hilfe der Backtracking-Suche kann man auch sehr große Suchräume absuchen.

Backtracking: Fazit

UNI FREIBURG

- Mit Hilfe der Backtracking-Suche kann man auch sehr große Suchräume absuchen.
- Die Methode garantiert, dass wir eine Lösung finden, wenn eine existiert.

Backtracking: Fazit

UNI FREIBURG

- Mit Hilfe der Backtracking-Suche kann man auch sehr große Suchräume absuchen.
- Die Methode garantiert, dass wir eine Lösung finden, wenn eine existiert.
- Die tatsächlich notwendige Zeit kann stark schwanken.

Backtracking: Fazit

UNI FREIBURG

- Mit Hilfe der Backtracking-Suche kann man auch sehr große Suchräume absuchen.
- Die Methode garantiert, dass wir eine Lösung finden, wenn eine existiert.
- Die tatsächlich notwendige Zeit kann stark schwanken.
- Können wir vielleicht weitere Abkürzungen bei der Suche einsetzen?

UNI FREIBURG

Constraint-Propagierung

Constraint-Propagierung: Die Idee

UNI FREIBURG

- Im Zusammenhang mit der Auswahl der nächsten Variable und des nächsten Wertes wurde bereits erwähnt, dass man die noch möglichen Werte pro Variable kennen sollte.

Constraint-Propagierung: Die Idee

UNI FREIBURG

- Im Zusammenhang mit der Auswahl der nächsten Variable und des nächsten Wertes wurde bereits erwähnt, dass man die noch möglichen Werte pro Variable kennen sollte.
- Idee: Wann immer ein Wert fest gelegt wird, eliminiere jetzt unmögliche Werte für andere Variablen.

Constraint-Propagierung: Die Idee

UNI FREIBURG

- Im Zusammenhang mit der Auswahl der nächsten Variable und des nächsten Wertes wurde bereits erwähnt, dass man die noch möglichen Werte pro Variable kennen sollte.
- Idee: Wann immer ein Wert fest gelegt wird, eliminiere jetzt unmögliche Werte für andere Variablen.

→ Forward-Checking – erlaubt uns die Suche früher abzubrechen.

Constraint-Propagierung: Die Idee

UNI FREIBURG

- Im Zusammenhang mit der Auswahl der nächsten Variable und des nächsten Wertes wurde bereits erwähnt, dass man die noch möglichen Werte pro Variable kennen sollte.
- Idee: Wann immer ein Wert fest gelegt wird, eliminiere jetzt unmögliche Werte für andere Variablen.

→ Forward-Checking – erlaubt uns die Suche früher abzubrechen.

- Beispiel: Wenn im Färbbarkeitsbeispiel *WA* = *red* gewählt wird, dann kann man für *NT* *red* ausschließen.

Forward-Checking: Beispiel

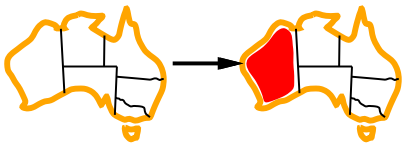
UNI FREIBURG

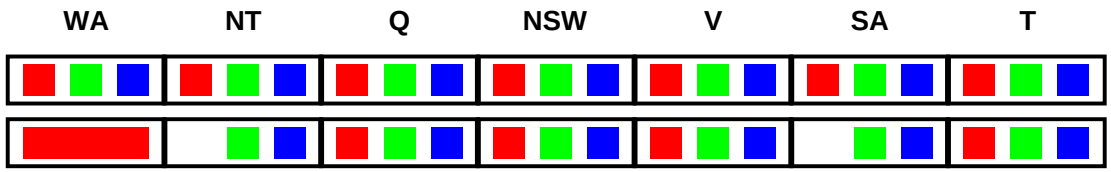
- Nach Zuweisung eines neuen Wertes an Variable X eliminiere in allen über Constraints verbundene Variablen jetzt nicht mehr möglichen Werte.
- Leite Backtracking ein, wenn für eine Variable kein Wert mehr möglich ist.

Forward-Checking: Beispiel

UNI FREIBURG

- Nach Zuweisung eines neuen Wertes an Variable X eliminiere in allen über Constraints verbundene Variablen jetzt nicht mehr möglichen Werte.
- Leite Backtracking ein, wenn für eine Variable kein Wert mehr möglich ist.



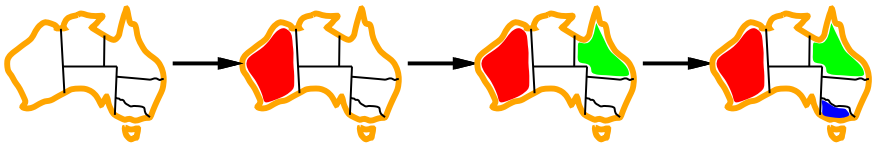


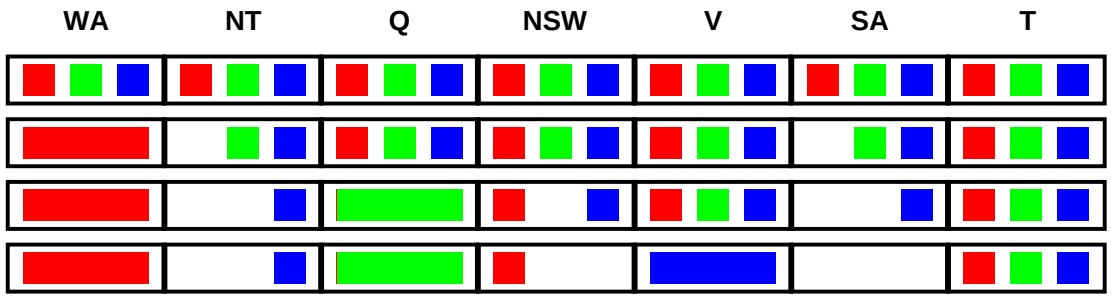
Forward-Checking: Beispiel

- Nach Zuweisung eines neuen Wertes an Variable X eliminiere in allen über Constraints verbundene Variablen jetzt nicht mehr möglichen Werte.
- Leite Backtracking ein, wenn für eine Variable kein Wert mehr möglich ist.

Forward-Checking: Beispiel

- Nach Zuweisung eines neuen Wertes an Variable *X* eliminiere in allen über Constraints verbundene Variablen jetzt nicht mehr möglichen Werte.
- Leite Backtracking ein, wenn für eine Variable kein Wert mehr möglich ist.





- Für *SA* ist jetzt kein Wert mehr möglich! Bereits jetzt kann Backtracking eingeleitet werden.

Forward-Checking: Übersehene Probleme

- Forward-Checking übersieht manchmal Probleme, da nur Information von belegten Variablen zu unbelegten Variablen fließt:

Forward-Checking: Übersehene Probleme

- Forward-Checking übersieht manchmal Probleme, da nur Information von belegten Variablen zu unbelegten Variablen fließt:

- Da *SA* und *NSW* benachbart sind, ist *blue* für *NSW* nicht mehr möglich.

Forward-Checking: Übersehene Probleme

- Forward-Checking übersieht manchmal Probleme, da nur Information von belegten Variablen zu unbelegten Variablen fließt:

- Da *SA* und *NSW* benachbart sind, ist *blue* für *NSW* nicht mehr möglich.
- Schlimmer: Da *SA* und *NT* benachbart sind, kann auch für *NT* der Wert *blue* ausgeschlossen werden.

Forward-Checking: Übersehene Probleme

- Forward-Checking übersieht manchmal Probleme, da nur Information von belegten Variablen zu unbelegten Variablen fließt:

- Da *SA* und *NSW* benachbart sind, ist *blue* für *NSW* nicht mehr möglich.
- Schlimmer: Da *SA* und *NT* benachbart sind, kann auch für *NT* der Wert *blue* ausgeschlossen werden.
- Generell: Immer wenn irgendwo ein Wert eliminiert wird, sollte man bei den über Constraints „verbundenen“ Variablen Werte eliminieren.

Constraint-Propagierung beim Sudoku

UNI FREIBURG

- Wir merken uns bei jedem Feld, welche Ziffern noch möglich sind.

Constraint-Propagierung beim Sudoku

UNI FREIBURG

- Wir merken uns bei jedem Feld, welche Ziffern noch möglich sind.
- Wird eine Ziffer eliminiert, überprüfen wir:

Constraint-Propagierung beim Sudoku

UNI FREIBURG

- Wir merken uns bei jedem Feld, welche Ziffern noch möglich sind.
- Wird eine Ziffer eliminiert, überprüfen wir:
  - Hat das Feld jetzt nur noch eine einzige Möglichkeit, dann kann die Möglichkeit bei allen Peers eliminiert werden.

Constraint-Propagierung beim Sudoku

UNI FREIBURG

- Wir merken uns bei jedem Feld, welche Ziffern noch möglich sind.
- Wird eine Ziffer eliminiert, überprüfen wir:
  - Hat das Feld jetzt nur noch eine einzige Möglichkeit, dann kann die Möglichkeit bei allen Peers eliminiert werden.
  - Ist in einer Gruppe eine bestimmte Ziffer nur noch in einem Feld möglich, so können wir die Ziffer hier platzieren (und alle anderen Möglichkeiten eliminieren).

Constraint-Propagierung beim Sudoku

UNI FREIBURG

- Wir merken uns bei jedem Feld, welche Ziffern noch möglich sind.
- Wird eine Ziffer eliminiert, überprüfen wir:
  - Hat das Feld jetzt nur noch eine einzige Möglichkeit, dann kann die Möglichkeit bei allen Peers eliminiert werden.
  - Ist in einer Gruppe eine bestimmte Ziffer nur noch in einem Feld möglich, so können wir die Ziffer hier platzieren (und alle anderen Möglichkeiten eliminieren).
- Jede Eliminierung stößt diesen Prozess wieder an.

Constraint-Propagierung beim Sudoku

UNI FREIBURG

- Wir merken uns bei jedem Feld, welche Ziffern noch möglich sind.
- Wird eine Ziffer eliminiert, überprüfen wir:
  - Hat das Feld jetzt nur noch eine einzige Möglichkeit, dann kann die Möglichkeit bei allen Peers eliminiert werden.
  - Ist in einer Gruppe eine bestimmte Ziffer nur noch in einem Feld möglich, so können wir die Ziffer hier platzieren (und alle anderen Möglichkeiten eliminieren).
- Jede Eliminierung stößt diesen Prozess wieder an.
- Man kann noch weitere Regeln aufstellen (speziell mit 2 und mehr Feldern/Werten) . . .

Verwalten der möglichen Werte: Einlesen

UNI FREIBURG

- Wir benutzen Strings von Ziffern um die Mengen der möglichen Werte zu repräsentieren.

Verwalten der möglichen Werte: Einlesen

UNI FREIBURG

- Wir benutzen Strings von Ziffern um die Mengen der möglichen Werte zu repräsentieren.
- Eigentlich wäre ja der Datentyp Set angemessener.

Verwalten der möglichen Werte: Einlesen

UNI FREIBURG

- Wir benutzen Strings von Ziffern um die Mengen der möglichen Werte zu repräsentieren.
- Eigentlich wäre ja der Datentyp Set angemessener.
- Aber das würde bedeuten, dass wir statt der copy-Methode die copy.deepcopy()-Funktion benutzen müssten, die sehr viel ineffizienter ist.

Verwalten der möglichen Werte: Einlesen

UNI FREIBURG

- Wir benutzen Strings von Ziffern um die Mengen der möglichen Werte zu repräsentieren.
- Eigentlich wäre ja der Datentyp Set angemessener.
- Aber das würde bedeuten, dass wir statt der copy-Methode die copy.deepcopy()-Funktion benutzen müssten, die sehr viel ineffizienter ist.
- Und mit Strings haben wir auch alle Mengen-Operationen, die wir benötigen.

Verwalten der möglichen Werte: Einlesen

UNI FREIBURG

- Wir benutzen Strings von Ziffern um die Mengen der möglichen Werte zu repräsentieren.
- Eigentlich wäre ja der Datentyp Set angemessener.
- Aber das würde bedeuten, dass wir statt der copy-Methode die copy.deepcopy()-Funktion benutzen müssten, die sehr viel ineffizienter ist.
- Und mit Strings haben wir auch alle Mengen-Operationen, die wir benötigen.

sudokucp.py (1)

```
def parse_grid(grid):
    values = {(s, digits) for s in squares}
    for s,d in (zip(squares, [c for c in grid
                                  if c in digits0p])):
        if d in digits and not assign(values, s, d):
            return False
    return values
```

Verwalten der möglichen Werte: Ausgabe

UNI FREIBURG

- Um auch nicht fertig gelöste Sudokus ausgegeben zu können, soll die `display`-Funktion so erweitert werden, dass sie alle Werte für die Felder ausgeben kann.

Verwalten der möglichen Werte: Ausgabe

UNI FREIBURG

- Um auch nicht fertig gelöste Sudokus ausgegeben zu können, soll die `display`-Funktion so erweitert werden, dass sie alle Werte für die Felder ausgeben kann.

sudokucp.py (2)

```
def display(values):
    "Display values as a 2-D grid."
    if not values:
        print("Empty grid")
        return
    width = 1+max(len(values[s]) for s in squares)
    line = '+'.join(['-'*(width*3)]*3)
    for r in rows:
        print(''.join(values[r+c].center(width) +
                      ('|' if c in '36' else '')
                      for c in cols))
        if r in 'CF': print(line)
    print()
```

Verwalten der möglichen Werte: Zuweisung

UNI FREIBURG

- `assign` eliminiert jetzt alle Werte außer dem zugewiesenen.

Verwalten der möglichen Werte: Zuweisung

UNI FREIBURG

- `assign` eliminiert jetzt alle Werte außer dem zugewiesenen.
- Treten bei der Eliminierung Fehler auf, dann ist die Zuweisung nicht möglich

Verwalten der möglichen Werte: Zuweisung

UNI FREIBURG

- `assign` eliminiert jetzt alle Werte außer dem zugewiesenen.
- Treten bei der Eliminierung Fehler auf, dann ist die Zuweisung nicht möglich

sudokucp.py (3)

```
def assign(values, s, d):
    "Try to assign value d to square s"
    others = values[s].replace(d, '')
    if all(eliminate(values, s, e) for e in others):
        return values
    return False
```

Verwalten der möglichen Werte: Eliminierung

UNI FREIBURG

- Nach der Eliminierung muss getestet werden, ob Lösung noch möglich.
- Dann werden die zwei Propagierungsregeln angewendet.

sudokucp.py (4)

```
def eliminate(values, s, d):
    if d not in values[s]:
        return values # already eliminated
    values[s] = values[s].replace(d, '')
    if not values[s]: # no more values left for s
        return False
    # check if value[s] has only one value left
    if not propagate_single_value(values, s):
        return False
    # check if unit has only a single square for value d
    if not propagate_single_square(values, s, d):
        return False
    return values
```

Verwalten der möglichen Werte: Propagierung

- Die beiden Propagierungsregeln:

Verwalten der möglichen Werte: Propagierung

UNI FREIBURG

- Die beiden Propagierungsregeln:

sudokucp.py (5)

```
def propagate_single_value(values, s):
    if len(values[s]) == 1:
        return all(eliminate(values, s2, values[s])
                   for s2 in peers[s])
    return True

def propagate_single_square(values, s, d):
    for u in units[s]:
        dplaces = [s for s in u if d in values[s]]
        if len(dplaces) == 0:
            return False # contradiction!
        elif len(dplaces) == 1:
            if not assign(values, dplaces[0], d):
                return False
    return True
```

Verwalten der möglichen Werte: Suche

UNI FREIBURG

- Geänderte Erfolgsbedingung (alle Var. haben genau einen Wert)

Verwalten der möglichen Werte: Suche

UNI FREIBURG

- Geänderte Erfolgsbedingung (alle Var. haben genau einen Wert)
- Geänderte Variablenauswahl (kleinster Wertebereich)

Verwalten der möglichen Werte: Suche

UNI FREIBURG

- Geänderte Erfolgsbedingung (alle Var. haben genau einen Wert)
- Geänderte Variablenauswahl (kleinster Wertebereich)
- Geänderte Werteselektion (nur mögliche Werte)

Verwalten der möglichen Werte: Suche

UNI FREIBURG

- Geänderte Erfolgsbedingung (alle Var. haben genau einen Wert)
- Geänderte Variablenauswahl (kleinster Wertebereich)
- Geänderte Werteselektion (nur mögliche Werte)

sudokucp.py (6)

```
def search(values):
  "Search for solution"
  if not values: return False # failed earlier
  if all(len(values[s]) == 1 for s in squares):
      return values
  _,s = min((len(values[s]), s) for s in squares
                                if len(values[s]) > 1)
  return some(search(assign(values.copy(), s, d))
              for d in values[s])
```

Testen der Propagierung (1)

UNI FREIBURG

Python-Interpreter

```
>>> display(parse_grid(grid1))

4 8 3 |9 2 1 |6 5 7
9 6 7 |3 4 5 |8 2 1
2 5 1 |8 7 6 |4 9 3
------+------+------
5 4 8 |1 3 2 |9 7 6
7 2 9 |5 6 4 |1 3 8
1 3 6 |7 9 8 |2 4 5
------+------+------
3 7 2 |6 8 9 |5 1 4
8 1 4 |2 5 3 |7 6 9
6 9 5 |4 1 7 |3 8 2
```

Testen der Propagierung (1)

UNI FREIBURG

Python-Interpreter

```
>>> display(parse_grid(grid1))

4 8 3 |9 2 1 |6 5 7
9 6 7 |3 4 5 |8 2 1
2 5 1 |8 7 6 |4 9 3
------+------+------
5 4 8 |1 3 2 |9 7 6
7 2 9 |5 6 4 |1 3 8
1 3 6 |7 9 8 |2 4 5
------+------+------
3 7 2 |6 8 9 |5 1 4
8 1 4 |2 5 3 |7 6 9
6 9 5 |4 1 7 |3 8 2
```

Das Sudoku wurde bereits beim Einlesen gelöst! Tatsächlich ist das bei allen einfachen Sudokus so.

Testen der Propagierung (2)

UNI FREIBURG

Python-Interpreter

```
>>> display(parse_grid(grid2))


  4      1679   12679  |  139    2369    269   |  8     1239   5
26789    3    1256789 | 14589  24569  245689 | 12679  1249  124679
 2689  15689  125689  |   7    234569 245689 | 12369 12349 123469
----------------------+----------------------+----------------------
 3789    2     15789  |  3459  34579   4579  | 13579   6    13789
 3679  15679  15679   |  359     8     25679 |   4    12359 12379
36789    4     56789  |  359     1     25679 | 23579 23589 23789
----------------------+----------------------+----------------------
 289     89     289   |   6     459     3    |  1259   7    12489
  5     6789    3     |   2     479     1    |   69   489    4689
  1     6789    4     |  589    579    5789  | 23569 23589 23689
```

Testen der Propagierung (2)

UNI FREIBURG

Python-Interpreter

```
>>> display(parse_grid(grid2))


  4      1679   12679  |  139    2369    269   |  8     1239   5
26789    3    1256789 | 14589  24569  245689 | 12679  1249  124679
 2689  15689  125689  |   7    234569 245689 | 12369 12349 123469
----------------------+----------------------+----------------------
 3789    2     15789  |  3459  34579   4579  | 13579   6    13789
 3679  15679  15679   |  359     8     25679 |   4    12359 12379
36789    4     56789  |  359     1     25679 | 23579 23589 23789
----------------------+----------------------+----------------------
 289     89     289   |   6     459     3    |  1259   7    12489
  5     6789    3     |   2     479     1    |   69   489    4689
  1     6789    4     |  589    579    5789  | 23569 23589 23689
```

Hier gibt es offensichtlich noch viele offene Möglichkeiten!

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
```

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
>>> timed_search(grid2)
0.013170000000000001
```

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
>>> timed_search(grid2)
0.013170000000000001
>>> timed_search(hard1)
0.009936999999999988
```

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
>>> timed_search(grid2)
0.013170000000000001
>>> timed_search(hard1)
0.009936999999999988
>>> timed_search(hard2)
0.013539999999999996
```

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
>>> timed_search(grid2)
0.013170000000000001
>>> timed_search(hard1)
0.009936999999999988
>>> timed_search(hard2)
0.013539999999999996
>>> timed_search(hard3)
118.054612
```

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
>>> timed_search(grid2)
0.013170000000000001
>>> timed_search(hard1)
0.009936999999999988
>>> timed_search(hard2)
0.013539999999999996
>>> timed_search(hard3)
118.054612
```

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
>>> timed_search(grid2)
0.013170000000000001
>>> timed_search(hard1)
0.009936999999999988
>>> timed_search(hard2)
0.013539999999999996
>>> timed_search(hard3)
118.054612
```

- Praktisch alle Sudokus können so in weniger als einer Sekunde gelöst werden.

Sudoku – Backtracking & Propagierung: Performanz

UNI FREIBURG

Python-Interpreter

```
>>> timed_search(grid1)
0.008320000000000001
>>> timed_search(grid2)
0.013170000000000001
>>> timed_search(hard1)
0.009936999999999988
>>> timed_search(hard2)
0.013539999999999996
>>> timed_search(hard3)
118.054612
```

- Praktisch alle Sudokus können so in weniger als einer Sekunde gelöst werden.
- `hard3` ist eine Ausnahme – allerdings auch kein eindeutig lösbares Sudoku.

UNI FREIBURG

Ausblick

Ausblick

UNI FREIBURG

- Backtracking zusammen mit Constraint-Propagierung ist eine extrem mächtige Technik, um schwierige kombinatorische Probleme zu lösen.

Ausblick

UNI FREIBURG

- Backtracking zusammen mit Constraint-Propagierung ist eine extrem mächtige Technik, um schwierige kombinatorische Probleme zu lösen.
- Wird auch in anderen Kontexten (z.B. SAT-Solving mit Millionen von Variablen) erfolgreich eingesetzt.

Ausblick

UNI FREIBURG

- Backtracking zusammen mit Constraint-Propagierung ist eine extrem mächtige Technik, um schwierige kombinatorische Probleme zu lösen.
- Wird auch in anderen Kontexten (z.B. SAT-Solving mit Millionen von Variablen) erfolgreich eingesetzt.
- Es gibt aber immer wieder Probleminstanzen, die sich als extrem schwierig heraus stellen.

Ausblick

UNI FREIBURG

- Backtracking zusammen mit Constraint-Propagierung ist eine extrem mächtige Technik, um schwierige kombinatorische Probleme zu lösen.
- Wird auch in anderen Kontexten (z.B. SAT-Solving mit Millionen von Variablen) erfolgreich eingesetzt.
- Es gibt aber immer wieder Probleminstanzen, die sich als extrem schwierig heraus stellen.
- Ab einer gewissen Größe (verallgemeinertes Sudoku!) wird es wirklich schwierig, wenn die Probleminstanzen nicht einfach durch Constraint-Propagierung lösbar sind.

Ausblick

UNI FREIBURG

- Backtracking zusammen mit Constraint-Propagierung ist eine extrem mächtige Technik, um schwierige kombinatorische Probleme zu lösen.
- Wird auch in anderen Kontexten (z.B. SAT-Solving mit Millionen von Variablen) erfolgreich eingesetzt.
- Es gibt aber immer wieder Probleminstanzen, die sich als extrem schwierig heraus stellen.
- Ab einer gewissen Größe (verallgemeinertes Sudoku!) wird es wirklich schwierig, wenn die Probleminstanzen nicht einfach durch Constraint-Propagierung lösbar sind.
- Es handelt sich hier um die so genannten NP-vollständigen Probleme.

Ausblick

UNI FREIBURG

- Backtracking zusammen mit Constraint-Propagierung ist eine extrem mächtige Technik, um schwierige kombinatorische Probleme zu lösen.
- Wird auch in anderen Kontexten (z.B. SAT-Solving mit Millionen von Variablen) erfolgreich eingesetzt.
- Es gibt aber immer wieder Probleminstanzen, die sich als extrem schwierig heraus stellen.
- Ab einer gewissen Größe (verallgemeinertes Sudoku!) wird es wirklich schwierig, wenn die Probleminstanzen nicht einfach durch Constraint-Propagierung lösbar sind.
- Es handelt sich hier um die so genannten NP-vollständigen Probleme.
- Und es gibt viel aktive Forschung in der Informatik, diesen Problemen zu Leibe zu rücken.