HTWK
Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Prozedur Blockfälle

Dokumentation

Christopher Kind, Toni Volker Schoechert

Prof. Dr. Geser
Leipzig, den 18. Juli 2023

Inhaltsverzeichnis

Tabellenverzeichnis

1. Allgemeines

Bevor das Programm detailliert erklärt wird, gibt es erstmal einen kurzen Überblick über die Aufgabe der Prozedur als auch die Aufzählung der Standard Bibliotheken die verwendet werden.

1.1. Übersicht

Der Sudoku-Löser verfolgt das Ziel, ein unvollständiges Sudoku in Form einer Textdatei einzulesen, auf Richtigkeit zu prüfen und anschließend zu lösen. Das Eintragen der Zahlen in den freien Feldern folgt bestimmten Regeln. Jede Zeile soll die Zahlen von 1 bis 9 genau einmal enthalten. Selbiges zählt auch für die Blöcke und Spalten. Durch diese Regelungen ergibt sich, dass jede Zahl im Sudoku genau neunmal vorkommt. Für jedes Sudoku existiert genau eine Lösung.
Das Lösen des Sudokus kann man in mehrere Fälle unterteilen, die auch unterschiedliche Grade an Komplexität aufweisen. Der einfachste Fall ist das Finden von Zahlen, durch das Prüfen, ob in einer Zeile, einer Spalte oder einem Block nur noch genau ein Feld frei ist. Eine weitere einfache Methode ist das Finden einer Zahl, indem man überprüft ob die Zahl bereits achtmal verwendet wurde.
Der nächst schwierigere Fall ist, dass es eine Position gibt, an die genau eine Zahl passt. Die Zahl wird ermittelt, durch Prüfen der Sudoku-Regeln, sodass nach deren Anwendung nur eine Möglichkeit bleibt.
Der letzte Fall ermittelt mithilfe einer vorgegebenen Zahl und eines vorgegebenen Blocks, die Position eben jener Zahl in dem Block. Dies erfolgt indem berücksichtigt wird, ob die Zahl bereits in anderen Blöcken derselben Blockspalte beziehungsweise derselben Blockzeile eingetragen ist und inwieweit der zu untersuchende Block bereits mit Zahlen belegt ist. Bei dieser Dokumentation geht es um den Letzten dieser Fälle, die Prozedur $analyse\_blockfaelle(b, k)$. Die Variable b steht für die Blocknummer und die Variable k für die zu untersuchende Zahl. 1.1 zeigt ein Beispiel für Blockfälle, mit der Zahl 3 und dem Block 1.

x	x	x		3				
x	x	x					3	
	x	9						
	3							

Tabelle 1.1.: Beispiel 1

1.2. Standart Bibliotheken

Für die Prozedur $analyse\_blockfaelle(b, k)$
werden folgende Bibliotheken verwendet: stdio.h[1], stdlib.h[1] und stdbool.h[1]

1.3. Globale Variablen

Damit die Prozedur funktioniert, werden Variablen benötigt um bestimmte Informationen zwischenzuspeichern Weil diese auch in anderen Teilen des Programms verwendet werden können, wurden diese Variablen global in der Testumgebung initialisiert.

Variablen Name	**Datentyp**	**Werte**	**Beschreibung**
zeile_fuer_block[10][10]	int Array	0..8	Array welches die Zeile beinhaltet, in der sich die entsprechende Zahl in dem gewählten Block befindet
spalte_fuer_block[10][10]	int Array	0..8	Array welches die Spalte beinhaltet, in der sich die entsprechende Zahl in dem gewählten Block befindet
zahl_an_position[9][9]	int Array	0..9	Array welches die Zahl beinhaltet, die sich in den entsprechenden Koordinaten des Sudokus befindet.
i	int	0..8	Zeile
j	int	0..8	Spalte
k	int	0..9	Zahl
b	int	1..9	Blocknummer
bz	int	0..2	Blockzeile
bs	int	0..2	Blockspalte
rz	int	0..2	relative Zeile
rs	int	0..2	relative Spalte

Tabelle 1.2.: Globale Variablen

2. Prototypen

Im fertigen Sudoku-Löser ist die Prozedur *analyse_blockfaelle* eingebunden in ein größeres Programm, welches die Prozedur abruft und alle nötigen Werte für die Ausführung der Prozedur vorgibt. Da die anderen Teile des Sudoku$_{L}$ösersgleichzeitigmitunsererProzedurentwickeltwurden, star

2.1. Sudokufeld

Für das Erstellen eines Sudoku Feldes, zum Testen unsere Prozedur, haben wir das Array *zahl_an_position* zu Beginn des Programms mit Werten gefüllt.

```
1  int zahl_an_position[9][9] = {
2          {0, 3, 4, 6, 7, 8, 9, 1, 2},
3          {6, 7, 2, 1, 0, 5, 0, 0, 8},
4          {0, 9, 8, 3, 4, 0, 5, 6, 7},
5          {8, 5, 9, 0, 6, 1, 4, 0, 3},
6          {0, 2, 6, 8, 0, 3, 7, 9, 1},
7          {7, 1, 0, 9, 2, 4, 8, 5, 0},
8          {9, 6, 1, 0, 3, 7, 2, 0, 0},
9          {0, 8, 0, 4, 1, 9, 6, 3, 5},
10         {3, 4, 5, 2, 0, 6, 1, 7, 0}
11 };
```

Das Sudoku in diesem Beispiel ist ein größerer Test zur Überprüfung von $analyse\_blockfaelle(b, k)$, die Nullen stehen im Programm für noch unbekannte Einträge.

2.2. Prozedur: spalte_fuer_block_fuellen

Diese Prozedur rechnet aus, welche Spalten zu einem Block gehören und speichert für jeden Block b die Spalte j ab, in der die Zahl k steht. Die Spalte j wird unter $spalte\_fuer\_block$ abgespeichert. Dabei geht $spalte\_fuer\_block\_fuellen$ die Zahlen 1 bis 9 und Blöcke 1 bis 9 durch.

```
1  int spalte_fuer_block_fuellen (){
2      int ez, es, countz, counts;
3      for (b=1; b<10; b++){
4       for (k=1; k<10; k++){
5           bz = (b-1)/3;
6           bs = (b-1)%3;
7           ez = (bz*3);
8           es = (bs*3);
9          spalte_fuer_block[b][k] = 10;    /
10          for (countz=0; countz < 3; countz++){
11           for (counts=0; counts < 3; counts++){
12            if (zahl_an_position[(ez)+countz][(es)+counts] == k){
13              spalte_fuer_block[b][k] = (es+counts);
14                  }
15              }
16          }
17       }
18     }
19 }
```

In Zeile 2 werden lokale Variablen initialisiert. Diese sind nur innerhalb der Prozeduren nötig und müssen zu Beginn der Prozeduren genullt werden. Deshalb rufen wir sie nur lokal auf. Mit der $for - Schleife$ in Zeile 3 werden alle Blöcke von 1 bis 9 durchlaufen. Die Innere Schleife geht alle Zahlen von 1 bis 9 durch. Damit werden alle Zahlen in jedem Block durchgegangen. In den Zeilen 5 und 6 werden die Blockzeilen(0..2) und Blockspalten(0..2) aus dem Block b berechnet. Diese Ergebnisse werden in Zeile 7 und 8, in die ersten Zeilen(0, 3 oder 6), bzw Spalten(0, 3 oder 6) im Block umgerechnet. Zeile 9 setzt den Wert von *spalte_fuer_block* als Standard auf 10. Der Wert liegt bei einem tatsächlichen Eintrag zwischen 0 und 8. Wenn die Prozedur keine entsprechenden Einträge findet, wird der Wert auf 10 gesetzt. Es ist auch möglich, eine andere Zahl außerhalb des Bereiches 0 bis 8 zu wählen. In Zeile 12 wird überprüft ob die Zahl k schon im Block eingetragen ist. Dies geschieht durch das Überprüfen, ob der Eintrag von $zahl\_an\_position[i(0..2)][j(0..2]$ gleich der Zahl k ist. Durch das durchlaufen der zwei $for - Schleifen$ in Zeile 10 und 11 werden alle Zellen im Block untersucht. Wenn die eingetragene Zahl gleich mit der Zahl k ist, wird die Spalte als spalte_fuer_block[b][k] mit den entsprechenden Werten für b und k gespeichert. Die jeweiligen Umrechnungen sind im Code ersichtlich.

2.3. Prozedur: zeile_fuer_block_fuellen

Diese Prozedur ist nahezu identisch zu der Prozedur *spalte_fuer_block_fuellen*, nur mit Zeilen anstelle von Spalten.

2.4. Prozedur: sudoku_ausgeben

Diese Prozedur gibt das Sudoku Feld, mit allen Einträgen, aus. Das Ausgeben des Feldes wird benötigt, um zu überprüfen ob *analyse_blockfaelle* neue Einträge gemacht hat.

```
1 int sudoku_ausgeben (){
2     for (int i = 0; i < 9; i++) {
3         for (int j = 0; j < 9; j++) {
4             printf("%d", zahl_an_position[i][j]);
5         }
6         printf("\n");
7     }
8 }
```

Mit Zeile 2 werden die Zeilen durchlaufen und mit Zeile 3 die Spalten. Damit werden alle Zellen des Sudokus durchlaufen. In Zeile 4 werden dann alle Zahlen an der momentanen Position durch Print ausgegeben. Der Zeilenumbruch in Zeile 6 sorgt dafür, dass nach dem alle Spalten in einer Zeile durchgegangen wurden, die nächsten Zahlen auf eine neue Zeile geschrieben werden.

2.5. Prozedur: eintragen

Diese Prozedur erfüllt die Aufgabe des Eintragens der Zahl k, falls für sie eine Zeile i und Spalte j gefunden wurde. Die Eintragung erfolgt in das Array *zahl_an_position* an der ermittelten Position (Zeile und Spalte).

```
1 int eintragen (int i, int j, int k) {
2     zahl_an_position[i][j] = k;
3 }
```

2.6. Hauptprogramm für Tests

Im Hauptprogramm werden alle Prozeduren ausgeführt, sodass *analyse_blockfaelle* getestet werden kann.

```
1  int main() {
2      spalte_fuer_block_fuellen();
3      zeile_fuer_block_fuellen();
4      int b_test, k_test;
5      for (b_test=1; b_test < 10; b_test++){
6          for ( k_test = 1; k_test < 10; k_test++){
7              analyse_blockfaelle(b_test, k_test);
8          }
9      }
10     sudoku_ausgeben();
11     return 0;
12 }
```

In den Zeilen 1 und 2 werden die Prozeduren *zeile/spalte_fuer_block_fuellen* ausgeführt, um alle nötigen Variablen für *analyse_blockfaelle* zu ermitteln. Danach werden in Zeile 4, 5 und 6 alle mögliche Kombinationen von b und k durchgegangen und an die Prozedur *analyse_blockfaelle* weitergeben. Damit wird jede Zahl in jedem Block untersucht. Im fertigen Sudoku-Löser sollen nur bestimmte Block und Zahlen Kombinationen untersucht werden. Um das Testen des Sudokus zu vereinfachen, haben wir den Aufruf der Prozedur durch zwei $for - Schleifen$ gelöst. Dies stellt sicher, dass alle Optionen im Sudoku getestet werden. Zeile 10 gibt dann das gelöste (soweit wie mit der Prozedur möglich) Sudoku aus.

3. Prozedur: analyse_blockfaelle

Nachdem die Testumgebung aufgebaut und damit alle nötigen externen Variablen geschaffen sind, kommen wir jetzt zur Hauptaufgabe der Prozedur Blockfaelle. Diese soll mit vorgegebenen Werten von b und k, auf Grundlage der eingetragenen Werte in zahl_an_position und *spalte-* bzw. $zeile\_fuer\_block$ neue Einträge finden. Die Prozedur $analyse\_blockfaelle[b][k]$ erreicht dies durch das Abarbeiten von 9 Schritten.
Zu Beginn der Prozedur wird überprüft ob die gesuchte Zahl k schon im Block b eingetragen ist, um zu verhindern, dass eine Zahl zwei mal im Block eingetragen wird. Wenn das nämlich der Fall ist, kann die Prozedur übersprungen werden. Das wird gemacht durch die Überprüfung von $spalte\_fuer\_block[b][k]$ und $zeile\_fuer\_block[b][k]$.

```
1  if(spalte_fuer_block[b][k]>=0 && spalte_fuer_block[b][k]<9 &&
2     zeile_fuer_block[b][k]>=0 && zeile_fuer_block[b][k]<9) {
3          return -1;
4      }
```

Falls die Zahl schon in dem Block eingetragen ist, sind für beide Arrays Werte zwischen 0 und 8 gespeichert. Falls das der Fall ist, wird die Prozedur verlassen.
Des Weiteren werden noch einige Lokale Variablen eingeführt und gesetzt.

```
1  int b1, x, count, rs0, rz0;
2      bool iflags[3];
3      bool jflags[3];
4
5      bs= (b-1)%3
6      bz= (b-1)/3;
7
8      for(x=0; x<3; x++){
9           jflags[x] = false ;
10          iflags[x] = false ;
11     }
```

Neben den Definitionen der Variablen werden in Zeile 7 und 8 noch die Blockspalte und -zeile von Block b berechnet. Die $for - Schleife$ von Zeile 10 bis 13 setzt i- und jflags auf false, damit keine Zeilen von Anfang an blockiert werden.

Variablen Name	Datentyp	Werte	Beschreibung
b1	int	1..9	geht Blöcke durch, die gleiche Blockspalte /zeile haben wie b (ausgenommen b)
x	int	0..2	Zähler für verschiedene Berechnungen und Schleifen
count	int	0..2	Zähler für Überprüfung von Bedingungen die mehrfach erfüllt seine sollen
rs0	int	0..2	Speichert für einträge potentielle relative Spalten
rz0	int	0..2	Speichert für einträge potentielle relative Zeilen
jflags[3]	bool array	0..2	Speichert blockierte relative Spalten
iflags[3]	bool array	0..2	Speichert blockierte relative Zeilen

Tabelle 3.1.: lokale Variablen

3.1. Schritt 1

Zur Bestimmung der Position von k im Block b, werden durch eine $for-Schleife$, alle Blöcke mit der gleichen Blockspalte auf Einträge von spalte_fuer_block untersucht.

```
1      for (x=0; x<3; x++){
2          b1 = x*3 + bs +1;
3              if (b1 != b){
4                j = spalte_fuer_block[b1][k];
5               if (j>=0 && j<9){
6                     jflags[(j)%3] = true;
7              }
8          }
```

Die $for-Schleife$ wird drei mal durchlaufen und erhöht jedes mal x um eins. In Zeile 2 erfolgt die Umrechnung von x und der Blockspalte in den entsprechend Block b1. Durch die $for-Schleife$ werden damit alle 3 Blöcke in der Blockspalte von b nacheinander untersucht. Es sollen nur die Blöcke untersucht werden, die sich von b unterscheiden. Dies geschieht durch den $if-Befehl$ in Zeile 3. In Zeile 4 wird j mit dem Wert für $spalte\_für\_block$, für den aktuellen Block b1 und die gesuchte Zahl, k belegt. Falls im Block b1 die Zahl k in einer Spalte bekannt ist, ist der Wert zwischen 0 und 8. In Zeile 5 wird überprüft ob j zwischen 0 und 8 liegt. Damit lässt sich erkennen, ob ein Eintragen erfolgte. Dies bedeutet, dass die Zahl in dem Block b1 eingetragen ist und somit eine der 3 relativen Zeilen der Blockspalte von b

für Zahl k blockiert ist. In Zeile 6 wird dann die Spalte j in eine relative Spalte umgerechnet und die entsprechenden jflags auf true gesetzt und damit blockiert.

3.2. Schritt 2

Schritt 2 ist nahezu identisch zu Schritt 1 nur mit den Variablen für Zeilen anstelle der Variablen für Spalten.

3.3. Schritt 3

Dieser Teil der Prozedur überprüft die noch nicht blockierten relativen Spalten des Blocks b auf andere Blockierungen.

```
1  for (rs=0; rs<3; rs++){
2          if (jflags[rs] ==false){
3              count =0;
4              for (rz=0; rz<3; rz++){
5                  if (iflags[rz] == true ||
6                          zahl_an_position[3*(bz)+rz][3*(bs)+rs]!=0){
7                           count++;
8                      }
9              }
10             if (count == 3) {
11                 jflags[rs] = true;
12             }
13         }
14     }
```

Mit der $for-schleife$ in Zeile 1 werden alle 3 relativen Spalten durchgegangen. Der If-Befehl in der 2. Zeile sorgt dafür, dass nur noch nicht blockierte Spalten untersucht werden. Durch die 3. Zeile wird bei beginn der Untersuchung einer neuen relativen Spalte der counter auf Null gesetzt. Der counter wird auf Null gesetzt, damit man ihn mehrfach verwenden kann. Anschliesend werden durch die $for-Schleife$ in Zeile 4 alle 3 relativen Zeilen der Spalte untersucht. Dabei wird der counter in Zeile 7 jedes mal erhöht, wenn mindestens eine der beiden Bedingungen des If-Befehls in Zeile 5 und 6 erfüllt sind. Also Falls die momentane relative Zeile blockiert wird oder falls $zahl\_an\_position[i][j]$ zwischen 1 und 9 liegt. Dabei sind i und j durch die Umrechnung der Blockzeilen und relativen Zeile sowie Blockspalte und relative Spalte gegeben. Durch beide Bedingungen kann die Zelle bestehend aus relativer Spalte und Zeile blockiert werden. Nachdem für die noch nicht blockierte Spalte die $for-$

Schleife von Zeile 4 bis 9 durchlaufen wurde, wird in Zeile 10 überprüft, ob der Wert des Counters 3 ist. Das würde bedeuten, dass alle Zellen der relativen Spalten blockiert sind. Falls ja, kann in diese relative Spalte die Zahl k nicht eingetragen werden. Dann wird in Zeile 11 der *jflags* von der relativen Spalte blockiert.

3.4. Schritt 4

Dieser Schritt verläuft nahezu identisch zu Schritt 3, nur werden hier die Variablen von Zeilen und Spalten vertauscht, mit der Ausnahme von Zeile 6. Damit wird überprüft, ob weitere relative Zeilen blockiert werden.

3.5. Schritt 5

Dieser Schritt stellt fest, ob es genau eine relative Spalte im Block gibt, die nicht blockiert ist. Diese ist dann die relative Spalte in der k stehen muss.

```
1     count = 0;
2     for (rs=0; rs<3; rs++){
3         if (jflags[rs] == true)
4             count++;
5         if (jflags[rs]== false){
6                 rs0 = rs; }
7     }
8     if (count != 2)
9             rs0= -1;
```

Als Erstes wird count wieder auf 0 gesetzt, um wiederverwendbarkeit zu gewährleisten. Mit der $for - Schleife$ in Zeile 2 bis 7 werden nochmal alle drei relativen Spalten von 0 bis 2 durchlaufen. Falls jflags der momentanen reativen Spalte auf true gesetzt ist, bedeutet es, dass die Spalte für die Zahl k blockiert ist und der Counter wird erhöht. Falls jflags auf False gesetzt ist, handelt es sich um eine relative Spalte in die k eingetragen werden kann und wird als rs0 gespeichert. Nach dreimaligen durchlaufen der Schleife, wird in Zeile 8 und 9 überprüft ob count ungleich 2 ist. Der Grund dafür ist, dass nur wenn count den Wert zwei besitzt, es genau eine relative Spalte im Block gibt, in der k eingetragen werden kann. Falls das nicht der Fall ist, muss der Wert von rs0 auf einen Wert gesetzt werden, der nicht zwischen 0 und 2 liegt. Damit wird verhindert, dass eine falsche oder nicht eindeutig bestimmte Spalte zum eintragen verwendet wird.

3.6. Schritt 6

Nach dem 5. Schritt wird rs0 in eine Spalte umgerechnet.

```
1  if (rs0 != -1)
2          spalte_fuer_block[b][k] = 3*(bs)+rs0 ;
```

Mit der 1. Zeile wird überprüft, ob genau eine relative Spalte gefunden wurde. Das passiert, durch die Abfrage ob der Wert von rso ungleich -1 ist. Denn rs0 wird auf -1 gesetzt, wenn es nicht genau eine relative Spalte im Block gibt, die für k infrage kommt. Wenn die Bedingung aus Zeile 1 erfüllt ist, wird dann in Zeile 2 rs0 in eine Spalte umgerechnet und als spalte_fuer_block[b][k] gespeichert.

3.7. Schritt 7

Dieser Schritt ist nahezu identisch zu Schritt 5. nur mit den Variablen für Zeilen statt mit den Variablen für Spalten

3.8. Schritt 8

Dieser Schritt verläuft nahezu identisch zu Schritt 6. nur mit den Variablen für Zeilen statt den Variablen für Spalten.

3.9. Schritt 9

Als letzter Schritt der Prozedur *analyse_blockfaelle* wird jetzt noch die Zahl eingetragen.

```
1  if(rs0 != -1 & rz0 != -1)
2      eintragen(zeile_fuer_block[b][k], spalte_fuer_block[b][k], k);
```

In der ersten Zeile wird überprüft, ob genau eine relative Spalte und genau eine relative Zeile gefunden wurde. Die Variablen rs0 und rz0 sind jeweils nur ungleich -1, wenn Eindeutig eine entsprechende Spalte beziehungsweise Zeile gefunden wurde. Wenn diese Bedingung erfüllt ist, ist die genaue Position der entsprechenden Zahl im Sudoku Feld bekannt. Durch die Prozedur *eintragen* wird dann die Zahl in das Sudoku eingetragen.

A. Anhang

A.1. Test 1

Mit diesem Test wird jede Zahl und jeder Block getestet. Dafür fehlen in dem gegebenen Sudoku in jedem Block 2 Zahlen. Die fehlenden Zahlen sind durch die Prozedur *analyse_blockfaelle* lösbar. Wobei Block 9 eine Ausnahme bildet. In diesem fehlen 3 Zahlen. Dieser Block ist nur lösbar durch das Finden von Lösungen in anderen Blöcken. Input Sudokufeld:

```
1  int zahl_an_position[9][9] = {
2          {0, 3, 4, 6, 7, 8, 9, 1, 2},
3          {6, 7, 2, 1, 0, 5, 0, 0, 8},
4          {0, 9, 8, 3, 4, 0, 5, 6, 7},
5          {8, 5, 9, 0, 6, 1, 4, 0, 3},
6          {0, 2, 6, 8, 0, 3, 7, 9, 1},
7          {7, 1, 0, 9, 2, 4, 8, 5, 0},
8          {9, 6, 1, 0, 3, 7, 2, 0, 0},
9          {0, 8, 0, 4, 1, 9, 6, 3, 5},
10         {3, 4, 5, 2, 0, 6, 1, 7, 0}
11 };
```

Output des Programm:

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

Die Prozedur findet fehlerfrei die komplette Lösung.

A.2. Test 2

In diesem Test wird ein ungelöstes Sudoku vorgegeben und überprüft, ob alle Einträge, die die Prozedur finden müsste, gefunden werden.

```
1  int zahl_an_position[9][9] = {
2          {2, 1, 0, 4, 0, 0, 0, 3, 6},
3          {8, 0, 0, 0, 0, 0, 0, 0, 5},
4          {0, 0, 5, 3, 0, 9, 8, 0, 0},
5          {6, 0, 4, 9, 0, 7, 1, 0, 0},
6          {0, 0, 0, 0, 3, 0, 0, 0, 0},
7          {0, 0, 7, 5, 0, 4, 6, 0, 2},
8          {0, 0, 6, 2, 0, 3, 5, 0, 0},
9          {5, 0, 0, 0, 0, 0, 0, 0, 9},
10         {9, 3, 0, 0, 0, 5, 0, 2, 7}
11 };
```

Output des Programm:

2 1 0 4 5 8 0 3 6
8 0 3 0 0 0 2 0 5
0 0 5 3 0 9 8 0 0
6 0 4 9 0 7 1 0 3
0 0 0 0 3 0 0 0 0
3 0 7 5 0 4 6 0 2
0 0 6 2 9 3 5 0 0
5 0 0 0 0 0 3 6 9
9 3 0 0 0 5 0 2 7

Die Prozedur findet alle möglichen Einträge und generiert auch keine fehlerhaften Einträge.

Literaturverzeichnis

[1] International Organisation for Standardization. ISO/IEC 9899:2018-Information technology-Programming languages, 2018.