Aufgabe 3: Sudokopie

Team-ID: 00428

Team: MingXu

Bearbeiter dieser Aufgabe:
Max Obreiter

1. October 2022

Inhaltsverzeichnis

Lösungsidee

Es werden alle möglichen Permutationen des alten Sudokus auf Übereinstimmung mit dem neuen Sudoku überprüft.

Umsetzung

Das alte (oben in der Datei) Sudoku wird bei allen Permutationen durch Zeilen-/Spalten-Vertauschen (inkl. Blöcke) auf das “angeblich” neue Sudoku/90 Grad gegen den Uhrzeigersinn gedrehtes neue Sudoku überprüft. Statt das alte Sudoku um 90 Grad im Uhrzeigersinn zu drehen, wird hier das neue Sudoku um 90 Grad gegen den Uhrzeigersinn rotiert.

Für jede mögliche Permutation werden zuerst die Positionen der existierenden Zahlen (also keine „0“) mit den neuen Sudokus überprüft. Sind die Positionen deckunggleich, so wird überprüft, ob alle Ziffern x beim alten Sudoku dieselbe Position wie alle y bei einem der neuen Sudokus hat. Ist dies der Fall, so wurde x zu y umbenannt.

Diese Aufgabe wird auf mehreren Prozessoren parallel ausgeführt, wodurch eine Aufgabe in unter 2 Sekunden auf einem „AMD FX(tm)-8300 Eight-Core Processor 3.30 Ghz“.

Beispiele

Sudoku0:

Zeilen 1-3: 2 1 3

Zeilen 7-9: 1 3 2

Spalten 1-3: 3 1 2

Spalten 4-6: 2 3 1

Spalten 7-9: 3 2 1

Sudoku1:

Zeilenblöcke: 3 1 2

Spaltenblöcke: 2 3 1

90 Grad im Uhrzeigersinn rotiert.

Sudoku2:

Zeilen 4-6: 1 3 2

Spaltenblöcke: 3 2 1

Umbenennung: 2 3 4 5 6 7 8 9 1

Sudoku3:

Unterschiedliche Sudokus!

Sudoku4:

Zeilenblöcke: 3 1 2

Zeilen 1-3: 2 3 1

Zeilen 4-6: 2 3 1

Zeilen 7-9: 3 2 1

Spaltenblöcke: 1 3 2

Spalten 1-3: 3 2 1

Spalten 4-6: 3 2 1

Spalten 7-9: 3 2 1

Umbenennung: 4 8 1 9 2 5 7 3 6

90 Grad im Uhrzeigersinn rotiert.

Quellcode

```rust
pub fn a3(sudokus: String) {
  let (original, neu) = Sudoku::parse(&sudokus);
  let mut neu_rotiert = neu; // neu wird kopiert
  neu_rotiert.rotiere();
  println!(
    "{}",
    // alle Kombinationen von Möglichkeiten, die Zeilen-/Spalten-(Blöcke) zu permutieren, 6 ^ 8
    // dies ist ein paralleler Iterator von `rayon`, welche die Parallelisierung übernimmt
    get_moeglichkeiten()
      // suche nach der ersten Permutation, durch die das alte Sudoku "das neue wird"
      .find_map_any(|umformungen| {
        // das alte Sudoku wird kopiert und danach umgeformt
        let neue_moeglichkeit = original.kopieren_und_umformen(umformungen);
        // ist das neu-erzeugte Sudoku einem der neuen Sudoku "ähnlich"
        // (=alles identisch ausser eine Umbenennung der Ziffern)
        let passend = neue_moeglichkeit.aehnlich(&neu);
        let nach_rotation_passend = neue_moeglichkeit.aehnlich(&neu_rotiert);
        // ist eines der beiden "ähnlich", so wird das Ergebnis formattiert und zurückgegeben
        passend.or(nach_rotation_passend).map(|umbenennung| {
          formattiere_ergebnis(nach_rotation_passend.is_some(), umformungen, umbenennung)
        })
      })
      .unwrap_or_else(|| "Unterschiedliche Sudokus!".to_string())
  );
}

fn aehnlich(&self, other: &Self) -> Option<[u8; 9]> {
  // die Felder der beiden Sudokus werden in einem Iterator über 81 Elementen transformiert,
  // bei dem jedes Element angibt, ob dieses Feld eine Ziffer enthält oder nicht (keine 0 == true)
```

```
  let [s, o] = [self, other].map(|s|
s.feld.iter().flatten().map(Option::is_some));
  // sind beide Iteratoren identisch (=an den selben Stellen gibt es
Ziffern/keine Ziffern)
  // so wird überprüft, ob es möglich ist, durch eine Umbenennung beide
Sudokus "gleich zu setzen"
  // itertools hat viele nützliche Funktionen auf Iteratoren
  if itertools::equal(s, o) {
    // für beide Sudokus wird eine HashMap erstellt, die die Ziffer als
Schlüssel besitzt
    // und als Wert die Positionen (0-indexed) in einem HashSet hat
    let [s, o] = [self, other].map(Self::positionen_nach_zahl);
    // sind beide HashMaps gleich lang (also beide haben dieselbe Anzahl
an verschiedenen Ziffern),
    // so werden deren HashSets auf Gleichheit überprüft und die
Umbenennung gespeichert
    if s.len() == o.len() {
      let mut umbenennung = [0; 9];
      s.into_iter()
        // hier wird für jede Ziffer x im 1. Sudoku eine Ziffer y im 2.
Sudoku gesucht,
        // deren Positionen identisch sind
        .map(|(n, e)| o.iter().find(|(_, f)| &&e == f).map(|(m, _)| (n,
*m)))
        .try_for_each(|o| {
          // gibt es eine solche Ziffer x und y, so ist o = Some(x, y). x
und y werden in umbenennung gespeichert.
          // Ist dies nicht der Fall, so ist o = None und "try_for_each"
gibt None zurück
          umbenennung[o?.0 as usize - 1] = o?.1;
          Some(())
        })
        // ist das Ergebnis von "try_for_each" nicht None (also es gibt
eine Umbenennung),
        // so wird die Umbenennung zurückgegeben (also statt Some(()) =>
Some(umbenennung))
        .map(|_| umbenennung)
    } else {
      None
    }
  } else {
```

```
    None
  }
}

fn positionen_nach_zahl(&self) -> HashMap<u8, HashSet<(usize, usize)>> {
  let mut h = HashMap::<_, HashSet<_>>::new();
  self.feld
    .iter()
    .enumerate()
    .flat_map(|(y, zeile)| {
      zeile
        .iter()
        .enumerate()
        .filter_map(move |(x, zahl)| zahl.map(|z| (x, y, z)))
    })
    .for_each(|(x, y, zahl)| {
      h.entry(zahl).or_default().insert((x, y));
    });
  h
}
```

Dies sind die möglichen Umformungen (hierbei beschreiben die Ziffern die Neue Position):

```
umformung! {
  Umformung {
    Keine: "1 2 3",
    LinksRotieren: "2 3 1",
    RechtsRotieren: "3 1 2",
    Vertausche1_2: "2 1 3",
    Vertausche1_3: "3 2 1",
    Vertausche2_3: "1 3 2"
  }
}
```