# Aufgabe 3: Sudokopie

Team-ID: 00534

Team: Monads

Bearbeiter/-innen dieser Aufgabe:
Leopold Jofer

20. November 2022

## Inhaltsverzeichnis



# Lösungsidee

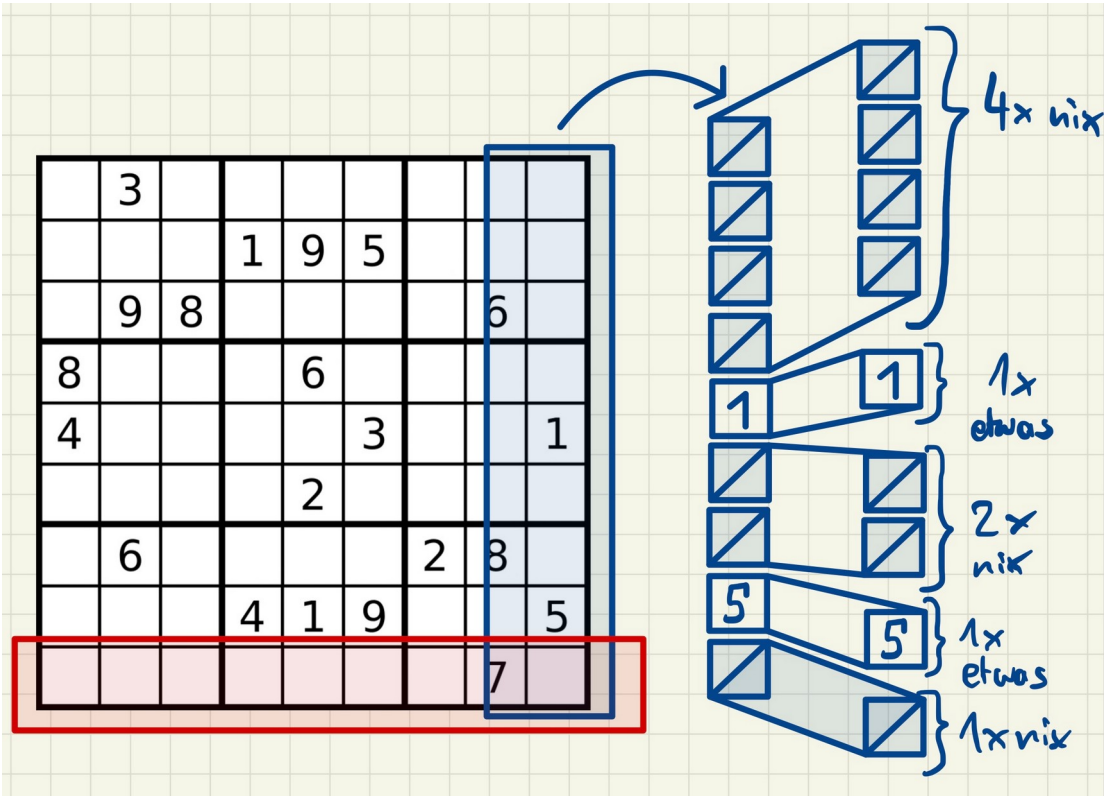
Um diese Aufgabe zu lösen bin ich auf 2 Lösungsansätze gekommen: ein Bruteforce-Ansatz, und ein “schlauer” Ansatz, für den ich mir einen eigenen Algorithmus ausgedacht habe. Beiden liegt das Prinzip zugrunde, dass sie für jede mögliche Rotation des Sudokus probiert werden. Für jede mögliche Rotation kann dann der Algorithmus parallel (also insgesamt 4 mal) ausgeführt werden.

### Der Bruteforce-Ansatz

Der Bruteforce-Ansatz testet nicht blind alle möglichen “Permutationen” durch, sondern baut einen Baum aus den Existierenden. Durch den Baum können wiederholte Berechnungen einigermaßen vermieden werden. Der Ansatz testet in Runden. Jede Runde wird eine neue Ebene mit den neuesten Permutationen zum Baum hinzugefügt. Es wird immer für die unterste Ebene im Baum getestet. Für jeden Knoten in dieser Ebene werden alle Permutationen ermittelt, die sich noch nicht über ihm im Baum befinden, d.h. noch nicht berechnet wurden.
Mit der Anzahl an Ebenen nimmt aber die Laufzeit trotz Optimierungen extrem zu:
$O(n^{Anzahl\ an\ möglichen\ Permutationen})$

### Der “schlaue” Ansatz

Dem schlauen Ansatz verwendet im Endeffekt eine Mustererkennung. Das erkannte “Muster” sind, wie viele und wie welche langen Lücken es in Spalten und Zeilen gibt:




Anschließend werden die Muster des Anfangs- und des Zielsudokus verglichen. Werden identische Spalten oder Zeilen im Ziel erkannt, die vertauscht sind, werden diese ausgetauscht. Dann wird das erste Mal geprüft, ob die beiden Sudokus mittlerweile gleich sind.
Ist dies nicht der Fall, werden die Zahlwerte der Sudokus verglichen. Wenn ein Zahlenwert im Endsudoku nicht der im Startsudoku entspricht, werden die beiden vertauscht.
Im Anschluss werden die Sudokus erneut verglichen. Sind sie nicht gleich, hat der Algorithmus keine Lösung gefunden.

# Umsetzung

## Bruteforce (`src/sudoku/bruteforce.rs`)

Die Bruteforce-Lösung wird in einem Loop bis zur maximalen Ebenentiefe ausgeführt:

```
for i in 0..depth {
     ...
}
```

Die untersten Elemente des Baums werden in der Variable `leaves` gespeichert.
Innerhalb des obigen Loops wird über die untersten Elemente des Baums iteriert:

```
for node in leaves.drain(..) {
     ...
}
```

Anschließend werden die möglichen Permutationen ermittelt:

```
let mut possible_permutations = possible_permutations.clone();
node.borrow_mut().walk(|n| {
     possible_permutations.remove(&n.perm);
});
```

Jede dieser Permutationen wird nun separat auf eine Kopie des Sudokus des Elements aus Loop 2 ausgeführt:

```
let new_sudoku = permutation.apply(sudoku);
```

Stimmen diese überein, wird die bisherige Permutationshistorie für dieses Element als Lösung zurückgegeben:

```
if new_sudoku == *goal {
     let mut solution: Vec<Permutation> = vec![permutation];
     node.borrow_mut().walk(|n| {
          if n.perm != Permutation::None {
               solution.push(n.perm);
          }
     });
     return Some(solution);
}
```

## Schlauer Ansatz (mehrere Dateien)

Zuerst werden die Muster für beide Sudokus ermittelt:

```
let patterns = start.get_patterns();
let goal_patters = goal.get_patters();
```

Daraufhin wird der im ersten Teil beschriebene Vergleich mit den Lücken erstellt und das Ergebnis verglichen:

```
let diff = patterns.diff(&goal_patterns);
start = diff.apply(start, &mut path);
if start == *goal {
     return Some(path);
}
```

Der eigentliche Patternmatcher wird in Teil 1 erklärt, befindet sich in src/sudoku/pattern_finder.rs und ist folgendermaßen aufgebaut:

```
fn get_pattern(&self, iter: impl Iterator<Item = [u8; 9]>) ->
pattern::List {
     let mut result = [EMPTY_PATTERN; 9];
```

```
     for (index, a) in iter.enumerate() {
          let mut pattern = Pattern::new();
          let mut fragment = Fragment {
               length: 0,
               is_zero: true,
          };
          for b in a {
               match fragment == b {
                    true => fragment.length += 1,
                    false => {
                         pattern.push(fragment);
                         fragment = Fragment {
                              length: 1,
                              is_zero: b == 0,
                         }
                    }
               }
          }
          result[index] = pattern;
     }
     result
}
```

Anschließend werden die Zahlenwerte der beiden Sudokus verglichen (falls welche vertauscht werden):

```
let (s, mut numbers) = start.match_numbers(goal);
```

Diese Methode vertauscht nicht übereinstimmende Zahlen:

```
if a != b {
     let perm = Permutation::SwapDigits(a, b);
     sudoku = perm.apply(sudoku);
     path.push(perm);
}
```

Stimmen die Sudokus überein, wird die Lösung zurückgegeben, sonst nichts:

```
if s == *goal {
     Some(path)
} else {
     None
}
```

# Beispiele

Beide Lösungen funktionieren aus ungeklärten Gründen leider nicht mit den Beispielen von der Website. Allerdings funktionieren beide mit einer einfacheren Test-Lösung sudoku5.txt, bei der die Reihen 1 und 2 vertauscht wurden und das Sudoku um 90 Grad gedreht wurde:

```
> Select one file to run using Sudoku sudoku5.txt
Ergebnis für Lösungsweg Smart:
0: Vertauschung der Reihen 2 und 1
1: Drehung um 90 Grad

Ergebnis für Lösungsweg Bruteforce:
0: Vertauschung der Reihen 1 und 2
1: Drehung um 90 Grad
```

# Quellcode

```
// Ausführen:
fn main() -> Result<(), dyn Error> {
	let file = shared::file_picker::to_string("Sudoku")?;

	let (start, goal) = Sudoku::parse_file(&file)?;

	let smart_result = rotate(start.clone(), &goal, |start, goal| {
		smart_find(start, goal, goal.get_patterns())
	});

	show_result(smart_result, "Smart");

	let brute_result = rotate(start, &goal, |start, goal| bruteforce(start,
goal, 5));
	show_result(brute_result, "Bruteforce");
}

// Bruteforce:

struct Node {
	parent: Option<Rc<RefCell<Node>>>,
	sudoku: Option<Box<Sudoku>>,
	perm: Permutation,
}

impl Node {
	fn walk<F: FnMut(&mut Self)>(&mut self, mut for_each: F) {
		for_each(self);
		if let Some(node) = &self.parent {
			node.borrow_mut().walk(for_each);
		}
	}
}

pub fn bruteforce(start: Sudoku, goal: &Sudoku, depth: u8) ->
Option<Vec<Permutation>> {
	let root = Rc::new(RefCell::new(Node {
		parent: None,
		perm: Permutation::None,
		sudoku: Some(Box::new(start.clone())),
	}));

	// The "leaves" are the bottom nodes of the tree
	let mut leaves = vec![Rc::clone(&root)];
```

```
	for _ in 0..depth {
		let possible_permutations = Permutation::get_possible();

		let mut new_leaves: Vec<Rc<RefCell<Node>>> = Vec::new();

		for node in leaves.drain(..) {
			let mut possible_permutations = possible_permutations.clone();
			node.borrow_mut().walk(|n| {
				possible_permutations.remove(&n.perm);
			});

			for permutation in possible_permutations.drain() {
				let sudoku = *node.borrow().sudoku.clone().unwrap();
				let new_sudoku = permutation.apply(sudoku);

				if new_sudoku == *goal {
					let mut solution: Vec<Permutation> = vec!
[permutation];

					node.borrow_mut().walk(|n| {
						if n.perm != Permutation::None {
							solution.push(n.perm);
						}
					});

					return Some(solution);
				}

				let new_node = Rc::new(RefCell::new(Node {
					parent: Some(Rc::clone(&node)),
					perm: permutation,
					sudoku: Some(Box::new(new_sudoku)),
				}));

				new_leaves.push(new_node);
			}

			node.borrow_mut().sudoku = None;
		}
		leaves = new_leaves;
	}
	None
}

// Smart find:
impl Sudoku {
	fn get_pattern(&self, iter: impl Iterator<Item = [u8; 9]>) ->
pattern::List {
		let mut result = [EMPTY_PATTERN; 9];

		for (index, a) in iter.enumerate() {
			let mut pattern = Pattern::new();
			let mut fragment = Fragment {
				length: 0,
				is_zero: true,
			};

			for b in a {
```

```
                        match fragment == b {
                              true => fragment.length += 1,
                              false => {
                                    pattern.push(fragment);
                                    fragment = Fragment {
                                          length: 1,
                                          is_zero: b == 0,
                                    }
                              }
                        }
                  }

                  result[index] = pattern;
            }

            result
      }

      fn match_numbers(self, other: &Self) -> (Self, Vec<Permutation>) {
            let mut path = Vec::new();

            let mut sudoku = self;

            for r in 0..9 {
                  for c in 0..9 {
                        let a = sudoku.0[r][c];
                        let b = other.0[r][c];
                        if a != b {
                              let perm = Permutation::SwapDigits(a, b);
                              sudoku = perm.apply(sudoku);
                              path.push(perm);
                        }
                  }
            }

            (sudoku, path)
      }
}
```

```
pub fn smart_find(
      mut start: Sudoku,
      goal: &Sudoku,
      goal_patterns: Collection<pattern::List>,
) -> Option<Vec<Permutation>> {
      let mut path = Vec::new();

      let patterns = start.get_patterns();
      let diff = patterns.diff(&goal_patterns);

      start = diff.apply(start, &mut path);

      if start == *goal {
            return Some(path);
      }

      let (s, mut numbers) = start.match_numbers(goal);

      path.append(&mut numbers);

      if s == *goal {
            Some(path)
      } else {
            None
      }
}

pub type Pattern = Vec<Fragment>;

pub struct Fragment {
      pub length: u8,
      pub is_zero: bool,
}
```