

Backtracking

Ausgewählte Kapitel aus "The Art of Computer Programming"

Bastian Kauschke

Hardware-Software-Co-Design, Friedrich-Alexander-Universität Erlangen-Nürnberg

11. August 2019

Outline

Einführung

n Queens

Langford Pairs

Laufzeit Einschätzung

Einführung

Anwendungsbereich

Sequenzen $x_1, x_2, x_3 \dots x_n$ für welche die Bedingung $P_n(x_1, x_2, x_3 \dots x_n)$ gilt.

P hat dabei folgende Eigenschaften:

- $P_I(x_1, x_2, x_3 \dots x_I)$ gilt nur, wenn $P_{I-1}(x_1, x_2, x_3 \dots x_{I-1})$ gilt
- wenn $P_I(x_1, x_2, x_3 \dots x_I)$ gilt, ist $P_{I+1}(x_1, x_2, x_3 \dots x_{I+1})$ einfach zu testen
- $P_0()$ gilt immer

Anwendungsbereich

Sequenzen $x_1, x_2, x_3 \dots x_n$ für welche die Bedingung $P_n(x_1, x_2, x_3 \dots x_n)$ gilt.

P hat dabei folgende Eigenschaften:

- $P_l(x_1, x_2, x_3 \dots x_l)$ gilt nur, wenn $P_{l-1}(x_1, x_2, x_3 \dots x_{l-1})$ gilt
- wenn $P_l(x_1, x_2, x_3 \dots x_l)$ gilt, ist $P_{l+1}(x_1, x_2, x_3 \dots x_{l+1})$ einfach zu testen
- $P_0()$ gilt immer

Anwendungsbereich

Sequenzen $x_1, x_2, x_3 \dots x_n$ für welche die Bedingung $P_n(x_1, x_2, x_3 \dots x_n)$ gilt.

P hat dabei folgende Eigenschaften:

- $P_l(x_1, x_2, x_3 \dots x_l)$ gilt nur, wenn $P_{l-1}(x_1, x_2, x_3 \dots x_{l-1})$ gilt
- wenn $P_l(x_1, x_2, x_3 \dots x_l)$ gilt, ist $P_{l+1}(x_1, x_2, x_3 \dots x_{l+1})$ einfach zu testen
- $P_0()$ gilt immer

Anwendungsbereich

Sequenzen $x_1, x_2, x_3 \dots x_n$ für welche die Bedingung $P_n(x_1, x_2, x_3 \dots x_n)$ gilt.

P hat dabei folgende Eigenschaften:

- $P_l(x_1, x_2, x_3 \dots x_l)$ gilt nur, wenn $P_{l-1}(x_1, x_2, x_3 \dots x_{l-1})$ gilt
- wenn $P_l(x_1, x_2, x_3 \dots x_l)$ gilt, ist $P_{l+1}(x_1, x_2, x_3 \dots x_{l+1})$ einfach zu testen
- $P_0()$ gilt immer

Algorithmus

```

/// A required set of methods needed for the generic backtracking algorithms.
pub trait Sequence {
    type Step;
    type Steps: IntoIterator<Item = Self::Step>;

    /// Checks if this sequence satisfy its condition.
    ///
    /// This function can assume that the parent of `self` satisfied this condition.
    fn satisfies_condition(&self) -> bool;

    /// generates all possible next steps at this current state.
    fn next_steps(&self) -> Self::Steps;

    /// applies a `step` to `self`, returning the resulting sequence.
    ///
    /// this function will only be called if `self.satisfies_condition() == true`.
    fn apply_step(&self, step: Self::Step) -> Self;
}

```


Algorithmus

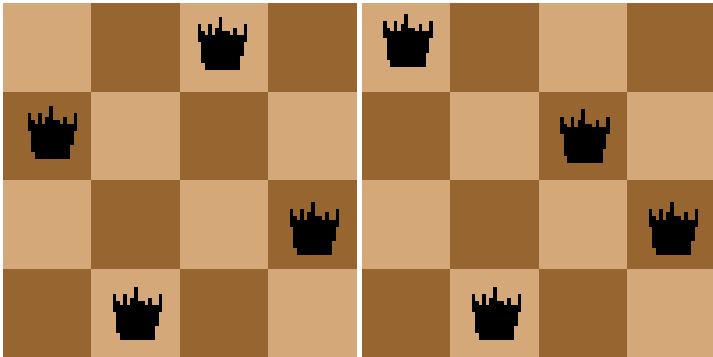
```
pub fn b<T: Sequence>(initial: T, n: usize) -> Vec<T> {
    let mut results = Vec::new();
    let mut states = Vec::new();
    let steps = initial.next_steps().into_iter();
    states.push((initial, steps));

    while let Some((state, steps)) = states.last_mut() {
        if let Some(step) = steps.next() {
            let next_state = state.apply_step(step);
            if next_state.satisfies_condition() {
                if states.len() < n {
                    let next_steps = next_state.next_steps().into_iter();
                    states.push((next_state, next_steps));
                } else {
                    results.push(next_state);
                }
            }
        } else {
            states.pop();
        }
    }
    results
}
```

n Queens

Damenproblem

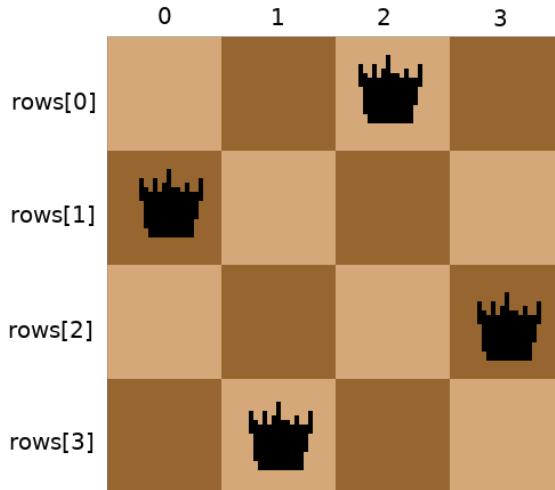
Wie kann man n Damen auf einem $n * n$ Schachbrett aufstellen, ohne dass sich zwei Damen schlagen können, also keine 2 Damen in der selben Zeile, Reihe oder Diagonalen stehen.



(a) Richtig

(b) Falsch

impl Sequence



impl Sequence

```
pub struct Queens {  
    n: usize,  
    rows: Vec<usize>,  
}  
  
impl Queens {  
    pub fn new(n: usize) -> Self {  
        Self {  
            n,  
            rows: Vec::new(),  
        }  
    }  
}
```

impl Sequence

```
fn satisfies_condition(&self) -> bool {  
    if self.rows.is_empty() {  
        return true;  
    }  
  
    let k = self.rows.len() - 1;  
  
    for j in 0..k {  
        let k_col = self.rows[k] as isize;  
        let j_col = self.rows[j] as isize;  
  
        if k_col == j_col || (j_col - k_col).abs() as usize == k - j {  
            return false;  
        }  
    }  
    true  
}
```

impl Sequence

```
fn next_steps(&self) -> Self::Steps {  
    0..self.n  
}  
  
fn apply_step(&self, step: Self::Step) -> Self {  
    let mut rows = self.rows.clone();  
    rows.push(step);  
    Self { n: self.n, rows }  
}
```

```
let results = b(Queens::new(4), 4);





pub fn b<T: Sequence>(initial: T, n: usize) -> Vec<T> {
    let mut results = Vec::new();
    let mut states = Vec::new();


    let steps = initial.next_steps().into_iter();
    states.push((initial, steps));
    // <- we are here
    // ...
}
```

results = []





states = [(, [, , , ])]



```
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        // <- we are here
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
// ...
```

states = [(, [, , )])]




step = 

```
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
// ...
```









states = [(, [, , ])]


next_state = 

```
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // our previous position
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
                // <- we are here
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
// ...
```








states = [(, [, , ]), (, [, , , ])]


```
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
// ...
```

states = [(, [, , ]), (, [, , ])]







next_state = 


```
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
// ...
```

states = [(, [, , ]), (, [, ])]







next_state = 







```
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
// ...
```

states = [(, [, , ]), (, [])]

next_state = 

```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
                // <- we are here
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
```

states = [(, [, , ]), (, []),

(, [, , , , ])])]







```
while let Some((state, steps)) = states.last_mut() {
  // <- we are here after discarding all 4 steps
  if let Some(step) = steps.next() {
    let next_state = state.apply_step(step);
    if next_state.satisfies_condition() {
      if states.len() < n {
        let next_steps = next_state.next_steps().into_iter();
        states.push((next_state, next_steps));
      } else {
        results.push(next_state);
      }
    } else {
      states.pop();
    }
  }
}
```


states = [(, [, , ]), (, []), (, [])]


```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
        // <- we are here
    }
}
```









states = [(, [, , ]), (, [])]


```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        } else {
            states.pop();
        }
    }
}
```

states = [(, [, , , ]), (, [])]









next_state = 

```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        } else {
            states.pop();
        }
    }
}
```

states = [(, [, , ]), (, []), (, [, ])]

next_state = 









```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
        // <- we are here
    }
}
```

states = [(, [, , ]), (, []), (, [, ])]











```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
        // <- we are here
    }
}
```


states = [(, [, , ]), (, []),]

```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
                // <- we are here
            } else {
                results.push(next_state);
            }
        } else {
            states.pop();
        }
    }
}
```











states = [(, [, ]), (, [, , , , ])]


```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            // <- we are here
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
    }
}
```

states = [(, [, ]), (, []), (, [, , ]), (, [])]


next_state = 


```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
                // <- we are here
            }
        }
    } else {
        states.pop();
    }
}
```


states = [(, [, ]), (, []), (, [, , ]),
(, [])]

results = [



```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
                // <- we are here
            }
        }
    } else {
        states.pop();
    }
}
```







```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        } else {
            states.pop();
            // <- we are here
        }
    }
}
```

states = [(, [])]

```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        }
    } else {
        states.pop();
        // <- we are here
    }
}
```

```
states = []
```

```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
                states.push((next_state, next_steps));
            } else {
                results.push(next_state);
            }
        } else {
            states.pop();
        }
    }
}
// <- we are here
results
```

states = []

results = 

B*

satisfies_condition vergleicht momentan die neue Reihe mit allen anderen Reihen.

Kann mit drei Bitsets ersetzt werden:



(a) Spalten

(b) Gegendiagonale

(c) Hauptdiagonale

Algorithm

```
pub fn b_star(n: usize) -> Vec<Queens> {
    // setup
    let mut results = Vec::new();

    let mut columns = BitVec::from_elem(n, false);
    let mut left_diagonals = BitVec::from_elem(2 * n - 1, false);
    let mut right_diagonals = BitVec::from_elem(2 * n - 1, false);

    let mut rows = Vec::new();
    let mut column = 0;

    loop {
        // ...
    }
}
```

```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        columns.set(column, true);
        left_diagonals.set(column + rows.len(), true);
        right_diagonals.set(column + n - 1 - rows.len(), true);
        rows.push(column);
        column = 0;
      } else {
        let mut q = rows.clone();
        q.push(column);
        results.push(Queens { n, rows: q });
        column += 1;
      }
    } else {
      column += 1;
    }
  }
}

if let Some(prev) = rows.pop() {
  right_diagonals.set(prev + n - 1 - rows.len(), false);
  left_diagonals.set(prev + rows.len(), false);
  columns.set(prev, false);
  column = prev + 1;
} else {
  return results;
}
}

```

```
let results = b_star(4);




pub fn b_star(n: usize) -> Vec<Queens> {
    let mut results = Vec::new();

    let mut columns = BitVec::from_elem(n, false);
    let mut left_diagonals = BitVec::from_elem(2 * n - 1, false);
    let mut right_diagonals = BitVec::from_elem(2 * n - 1, false);

    let mut rows = Vec::new();
    let mut column = 0;

    // ...
}
```

results = []

columns = , *left_diagonals* = , *right_diagonals* = 





rows = 

column = 0


```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        // <- we are here
        columns.set(column, true);
        left_diagonals.set(column + rows.len(), true);
        right_diagonals.set(column + n - 1 - rows.len(), true);
        rows.push(column);
        column = 0;
      } else {
        // add to results, removed for clarity
      }
    } else {
      column += 1;
    }
  }
  // backtracking, removed for clarity
}





```

columns = ,
 left_diagonals = ,
 right_diagonals = 
rows = 
column = 0

```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        columns.set(column, true);
        left_diagonals.set(column + rows.len(), true);
        right_diagonals.set(column + n - 1 - rows.len(), true);
        rows.push(column);
        // <- we are here
        column = 0;
      } else {
        // add to results, removed for clarity
      }
    } else {
      column += 1;
    }
  }
  // backtracking, removed for clarity
}





```

$columns =$ ,
 $left_diagonals =$ ,
 $right_diagonals =$ 
 $rows =$ 
 $column = 0$

```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        columns.set(column, true);
        left_diagonals.set(column + rows.len(), true);
        right_diagonals.set(column + n - 1 - rows.len(), true);
        rows.push(column);
        column = 0;
      } else {
        // add to results, removed for clarity
      }
    } else {
      column += 1;
      // <- we are here
    }
  }
  // backtracking, removed for clarity
}





```

$columns =$ ,
 $left_diagonals =$ ,
 $right_diagonals =$ 
 $rows =$ 
 $column = 1$

```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        columns.set(column, true);
        left_diagonals.set(column + rows.len(), true);
        right_diagonals.set(column + n - 1 - rows.len(), true);
        rows.push(column);
        column = 0;
      } else {
        // add to results, removed for clarity
      }
    } else {
      column += 1;
      // <- we are here
    }
  }
  // backtracking, removed for clarity
}




```


$columns =$ ,
 $left_diagonals =$ ,
 $right_diagonals =$ 
 $rows =$ 
 $column = 2$

```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        columns.set(column, true);
        left_diagonals.set(column + rows.len(), true);
        right_diagonals.set(column + n - 1 - rows.len(), true);
        rows.push(column);
        column = 0;
        // <- we are here
      } else {
        // add to results, removed for clarity
      }
    } else {
      column += 1;
    }
  }
  // backtracking, removed for clarity
}

```





$columns =$ ,
 $left_diagonals =$ ,
 $right_diagonals =$ 

$rows =$ 

$column = 0$

```
loop {
  // while ...

  // <- we are here
  if let Some(prev) = rows.pop() {
    right_diagonals.set(prev + n - 1 - rows.len(), false);
    left_diagonals.set(prev + rows.len(), false);
    columns.set(prev, false);
    column = prev + 1;
  } else {
    break;
  }
}
```

$columns =$ ,
 $left_diagonals =$ ,
 $right_diagonals =$ 
 $rows =$ 
 $column = 4$




```

loop {
  // while ...

  if let Some(prev) = rows.pop() {
    right_diagonals.set(prev + n - 1 - rows.len(), false);
    left_diagonals.set(prev + rows.len(), false);
    columns.set(prev, false);
    column = prev + 1;
    // <- we are here
  } else {
    return results;
  }
}

```

prev = 2

columns = ,
 left_diagonals = ,
 right_diagonals = 




rows = 


column = 3

```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        // update rows etc, removed for clarity
      } else {
        // <- we are here
        let mut q = rows.clone();
        q.push(column);
        results.push(Queens { n, rows: q });
        column += 1;
      }
    } else {
      column += 1;
    }
  }
  // backtracking, removed for clarity
}

```

$columns =$ ,
 $left_diagonals =$ ,
 $right_diagonals =$ 

$rows =$ 

$column = 2$


```

loop {
  while column < n {
    if !(columns[column]
      || left_diagonals[column + rows.len()]
      || right_diagonals[column + n - 1 - rows.len()])
    {
      if rows.len() + 1 < n {
        // update rows etc, removed for clarity
      } else {
        let mut q = rows.clone();
        q.push(column);
        results.push(Queens { n, rows: q });
        column += 1;
        // <- we are here
      }
    } else {
      column += 1;
    }
  }
  // backtracking, removed for clarity
}

```





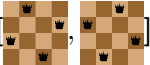

results = 

rows = 

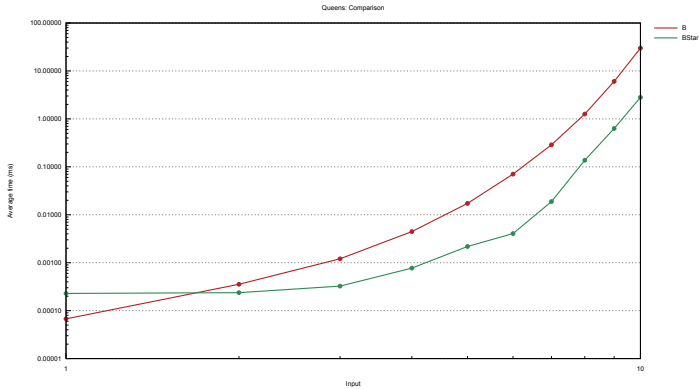
column = 3

```
loop {
  // while ...

  // <- we are here
  if let Some(prev) = rows.pop() {
    right_diagonals.set(prev + n - 1 - rows.len(), false);
    left_diagonals.set(prev + rows.len(), false);
    columns.set(prev, false);
    column = prev + 1;
  } else {
    return results;
  }
}
```

$columns =$ , $left_diagonals =$ , $right_diagonals =$ 
 $rows =$ 
 $column = 4 results = [$ ,  $]$

Performance



Langford Pairs

Langford Paare

Finde alle Permutationen der Menge $M = 1, -1, 2, -2, \dots, n, -n$ für die gilt $x_I = p \Rightarrow x_{I+p+1} = -p$.

Für $n = 4$ sind $[2, 3, 4, -2, 1, -3, -1, -4]$ und $[4, 1, 3, -1, 2, -4, -3, -2]$ die einzigen Lösungen.

Implementation

```
pub fn l(n: usize) -> Vec<Vec<isize>> {
    let mut results = Vec::new();

    let mut sequence = vec![0; n * 2];
    let mut position = 0;

    let mut unused_values = (1..=n).collect::<Vec<_>>();
    unused_values.push(0);

    let mut undo = vec![0; n * 2];
    let mut ptr = 0;

    loop {
        // ...
    }
}
```

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      sequence[position] = unused_values[ptr] as isize;
      sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
      undo[position] = ptr;
      unused_values[ptr] = unused_values[unused_values[ptr]];

      ptr = 0;
      position += 1;
      if unused_values[ptr] == 0 {
        results.push(sequence.clone());
      } else {
        while sequence[position] < 0 {
          position += 1;
        }
      }
    } else {
      ptr = unused_values[ptr];
    }
  }
  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;
    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
  } else {
    return results;
  }
}

```

```
pub fn l(n: usize) -> Vec<Vec<isize>> {
    let mut results = Vec::new();

    let mut sequence = vec![0; n * 2];
    let mut position = 0;

    let mut unused_values = (1..=n).collect::<Vec<_>>();
    unused_values.push(0);

    let mut undo = vec![0; n * 2];
    let mut ptr = 0;
    // <- we are here

    loop {
        // ...
    }
}

results = []
sequence = [0, 0, 0, 0, 0, 0], position = 0
unused_values = [1, 2, 3, 0], ptr = 0, unused_values[ptr] = 1
undo = [0, 0, 0, 0, 0, 0]
```



```

loop {
    while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
        if sequence[position + unused_values[ptr] + 1] == 0 {
            // <- we are here
            sequence[position] = unused_values[ptr] as isize;
            sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);

            undo[position] = ptr;
            unused_values[ptr] = unused_values[unused_values[ptr]];

            // update position and reset ptr, removed for clarity
        } else {
            ptr = unused_values[ptr];
        }
    }
    // backtracking, removed for clarity
}

```

sequence = [0, 0, 0, 0, 0, 0], *position* = 0
unused_values = [1, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 1
undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
      sequence[position] = unused_values[ptr] as isize;
      // <- we are here
      undo[position] = ptr;
      unused_values[ptr] = unused_values[unused_values[ptr]];

      // update position and reset ptr, removed for clarity
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 0
unused_values = [1, 2, 3, 0], *ptr* = 0, *unused_values[ptr]* = 1
undo = [0, 0, 0, 0, 0, 0]

```

loop {
    while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
        if sequence[position + unused_values[ptr] + 1] == 0 {
            sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
            sequence[position] = unused_values[ptr] as isize;

            undo[position] = ptr;
            unused_values[ptr] = unused_values[unused_values[ptr]];
            // <- we are here
            // update position and reset ptr, removed for clarity
        } else {
            ptr = unused_values[ptr];
        }
    }
    // backtracking, removed for clarity
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 0
unused_values = [2, 2, 3, 0], *ptr* = 0, *unused_values[ptr]* = 2
undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      // update sequence and undo, removed for clarity
      unused_values[ptr] = unused_values[unused_values[ptr]];

      ptr = 0;
      position += 1;
      // <- we are here
      if unused_values[ptr] == 0 {
        results.push(sequence.clone());
      } else {
        while sequence[position] < 0 {
          position += 1;
        }
      }
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 1

unused_values = [2, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 2

undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      // update sequence and undo, removed for clarity
      unused_values[ptr] = unused_values[unused_values[ptr]];

      ptr = 0;
      position += 1;
      if unused_values[ptr] == 0 {
        results.push(sequence.clone());
      } else {
        while sequence[position] < 0 {
          position += 1;
        }
        // <- we are here
      }
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 1

unused_values = [2, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 2

undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
      sequence[position] = unused_values[ptr] as isize;

      undo[position] = ptr;
      unused_values[ptr] = unused_values[unused_values[ptr]];
      // <- we are here
      // update position and reset ptr, removed for clarity
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [1, 2, -1, 0, -2, 0], *position* = 1
unused_values = [3, 2, 3, 0], *ptr* = 0, *unused_values[ptr]* = 2
undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      // update sequence and undo, removed for clarity
      unused_values[ptr] = unused_values[unused_values[ptr]];

      ptr = 0;
      position += 1;
      if unused_values[ptr] == 0 {
        results.push(sequence.clone());
      } else {
        while sequence[position] < 0 {
          position += 1;
        }
        // <- we are here
      }
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [1, 2, -1, 0, -2, 0], *position* = 3

unused_values = [3, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 3

undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    // compute and execute next steps, removed for clarity
  }
  // <- we are here
  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;
    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
  } else {
    return results;
  }
}

```

sequence = [1, 2, -1, 0, -2, 0], *position* = 3
unused_values = [3, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 3
undo = [0, 0, 0, 0, 0, 0]


```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    // compute and execute next steps, removed for clarity
  }

  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }
    // <- we are here

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;
    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
  } else {
    return results;
  }
}

```

sequence = [1, 2, -1, 0, -2, 0], *position* = 1
unused_values = [3, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 3
undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    // compute and execute next steps, removed for clarity
  }

  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;
    // <- we are here
    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
  } else {
    return results;
  }
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 1
unused_values = [3, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 3
undo = [0, 0, 0, 0, 0, 0], *removed_value* = 2

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    // compute and execute next steps, removed for clarity
  }

  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;

    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
    // <- we are here
  } else {
    return results;
  }
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 1
unused_values = [2, 2, 3, 0], *ptr* = 2, *unused_values*[*ptr*] = 3
undo = [0, 0, 0, 0, 0, 0], *removed_value* = 2

```

loop {
    while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
        if sequence[position + unused_values[ptr] + 1] == 0 {
            // <- we are here
            sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
            sequence[position] = unused_values[ptr] as isize;

            undo[position] = ptr;
            unused_values[ptr] = unused_values[unused_values[ptr]];
            // update position and reset ptr, removed for clarity
        } else {
            ptr = unused_values[ptr];
        }
    }
    // backtracking, removed for clarity
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 1
unused_values = [2, 2, 3, 0], *ptr* = 2, *unused_values*[*ptr*] = 3
undo = [0, 0, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
      sequence[position] = unused_values[ptr] as isize;

      undo[position] = ptr;
      unused_values[ptr] = unused_values[unused_values[ptr]];
      // <- we are here
      // update position and reset ptr, removed for clarity
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [1, 3, -1, 0, 0, -3], *position* = 1
unused_values = [2, 2, 0, 0], *ptr* = 2, *unused_values[ptr]* = 0
undo = [0, 2, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      // update sequence and undo, removed for clarity
      unused_values[ptr] = unused_values[unused_values[ptr]];

      ptr = 0;
      position += 1;
      if unused_values[ptr] == 0 {
        results.push(sequence.clone());
      } else {
        while sequence[position] < 0 {
          position += 1;
        }
        // <- we are here
      }
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [1, 3, -1, 0, 0, -3], *position* = 3

unused_values = [2, 2, 0, 0], *ptr* = 0, *unused_values*[*ptr*] = 2

undo = [0, 2, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    // compute and execute next steps, removed for clarity
  }

  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;
    // <- we are here
    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
  } else {
    return results;
  }
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 1
unused_values = [2, 2, 0, 0], *ptr* = 0, *unused_values*[*ptr*] = 2
undo = [0, 2, 0, 0, 0, 0], *removed_value* = 3

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    // compute and execute next steps, removed for clarity
  }

  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;

    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
    // <- we are here
  } else {
    return results;
  }
}

```

sequence = [1, 0, -1, 0, 0, 0], *position* = 1
unused_values = [2, 2, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 2
undo = [0, 2, 0, 0, 0, 0], *removed_value* = 3


```

loop {
    while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
        if sequence[position + unused_values[ptr] + 1] == 0 {
            // <- we are here
            sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
            sequence[position] = unused_values[ptr] as isize;

            undo[position] = ptr;
            unused_values[ptr] = unused_values[unused_values[ptr]];
            // update position and reset ptr, removed for clarity
        } else {
            ptr = unused_values[ptr];
        }
    }
    // backtracking, removed for clarity
}

```

sequence = [2, 3, 0, -2, 0, -3], *position* = 2
unused_values = [1, 0, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 1
undo = [1, 1, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    if sequence[position + unused_values[ptr] + 1] == 0 {
      sequence[position + unused_values[ptr] + 1] = -(unused_values[ptr] as isize);
      sequence[position] = unused_values[ptr] as isize;

      undo[position] = ptr;
      unused_values[ptr] = unused_values[unused_values[ptr]];
      // <- we are here
      // update position and reset ptr, removed for clarity
    } else {
      ptr = unused_values[ptr];
    }
  }
  // backtracking, removed for clarity
}

```

sequence = [2, 3, 1, -2, -1, -3], *position* = 2
unused_values = [0, 0, 3, 0], *ptr* = 0, *unused_values[ptr]* = 1
undo = [1, 1, 0, 0, 0, 0]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len()
    if sequence[position + unused_values[ptr] + 1] == 0 {
      // update sequence and undo, removed for clarity
      unused_values[ptr] = unused_values[unused_values[ptr]];

      ptr = 0;
      position += 1;
      if unused_values[ptr] == 0 {
        results.push(sequence.clone());
        // <- we are here
      } else {
        while sequence[position] < 0 {
          position += 1;
        }
      } else {
        ptr = unused_values[ptr];
      }
    }
  // backtracking, removed for clarity
}

```

sequence = [2, 3, 1, -2, -1, -3], *position* = 3
unused_values = [0, 0, 3, 0], *ptr* = 0, *unused_values*[*ptr*] = 1
undo = [1, 1, 0, 0, 0, 0], *results* = [[2, 3, 1, -2, -1, -3]]

```

loop {
  while unused_values[ptr] != 0 && position + unused_values[ptr] + 1 < sequence.len() {
    // compute and execute next steps, removed for clarity
  }
  // <- we are here
  if position != 0 {
    position -= 1;
    while sequence[position] < 0 {
      position -= 1;
    }

    let removed_value = sequence[position] as usize;
    sequence[position] = 0;
    sequence[position + removed_value + 1] = 0;

    unused_values[undo[position]] = removed_value;
    ptr = removed_value;
  } else {
    return results;
  }
}

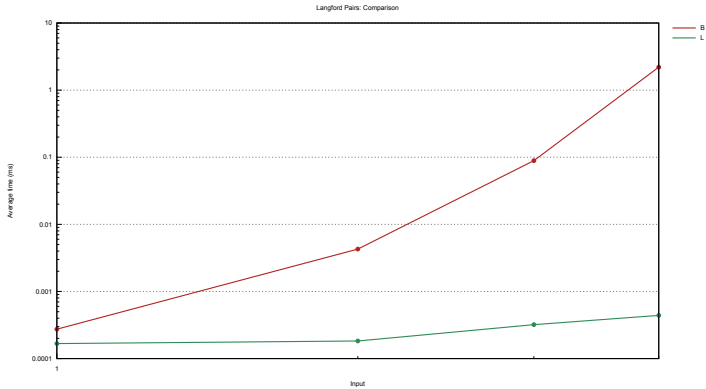
```

sequence = [0, 0, 0, 0, 0, 0], *position* = 0

unused_values = [1, 2, 3, 0], *ptr* = 3, *unused_values*[*ptr*] = 0

undo = [2, 0, 0, 0, 0, 0], *results* = [[2, 3, 1, -2, -1, -3], [3, 1, 2, -1, -3, -2]]

Performance



Laufzeit Einschätzung

```
pub fn e<T: Sequence>(initial: T, n: usize) -> Duration {
    let mut current = initial;

    let mut estimated_steps = 1;
    let mut total_length = Duration::from_secs(0);

    for _ in 0..n - 1 {
        let now = Instant::now();

        let next_states: Vec<_> = current
            .next_steps()
            .into_iter()
            .map(|step| current.apply_step(step))
            .filter(|state| state.satisfies_condition())
            .collect();

        total_length += now.elapsed() * estimated_steps;
        estimated_steps *= next_states.len() as u32;
        if let Some(next) = next_states.into_iter().choose(&mut thread_rng()) {
            current = next;
        } else {
            break;
        }
    }
    total_length
}
```

```
pub fn repeated_estimate<T: Sequence + Clone>(  
    initial: T,  
    n: usize,  
    repetitions: u32,  
) -> Duration {  
    let mut total_duration = Duration::from_secs(0);  
  
    for _ in 0..repetitions {  
        total_duration += e(initial.clone(), n);  
    }  
  
    total_duration / repetitions  
}
```


Vielen Dank für die Aufmerksamkeit.
Gibt es noch Fragen?