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

Anwendungsbereich

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

P hat dabei folgende Eigenschaften:

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

Anwendungsbereich

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

P hat dabei folgende Eigenschaften:

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

Anwendungsbereich

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

P hat dabei folgende Eigenschaften:

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

Anwendungsbereich

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

P hat dabei folgende Eigenschaften:

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

Algorithmus

```rust
/// 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;
}
```

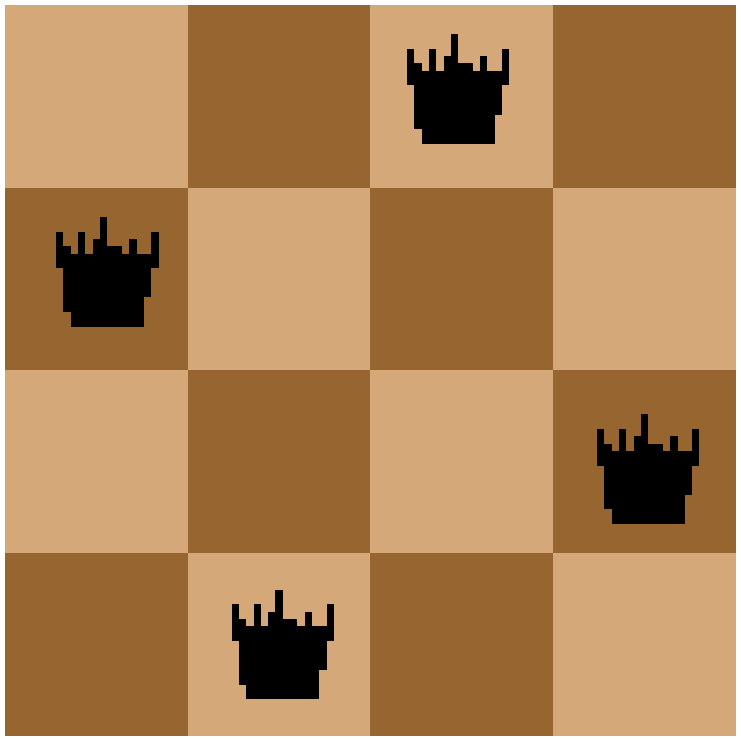
Algorithmus

```rust
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
}
```

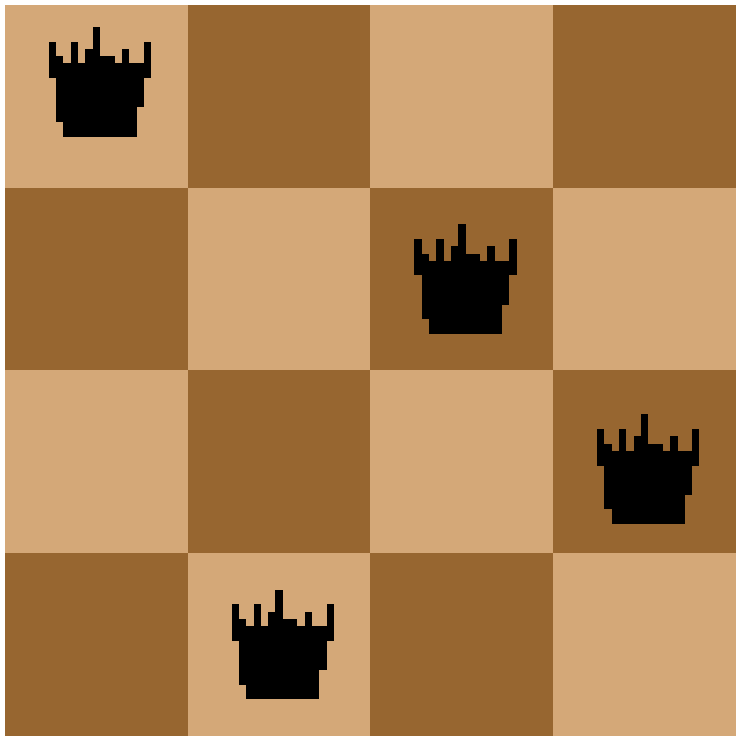
n Queens

Damenproblem

Wie viele Möglichkeiten gibt es n Damen auf einem $n * n$ Schachbrett aufzustellen, dass sich keine zwei Damen schlagen können, also keine 2 Damen in der selben Zeile, Reihe oder Diagonalen stehen.

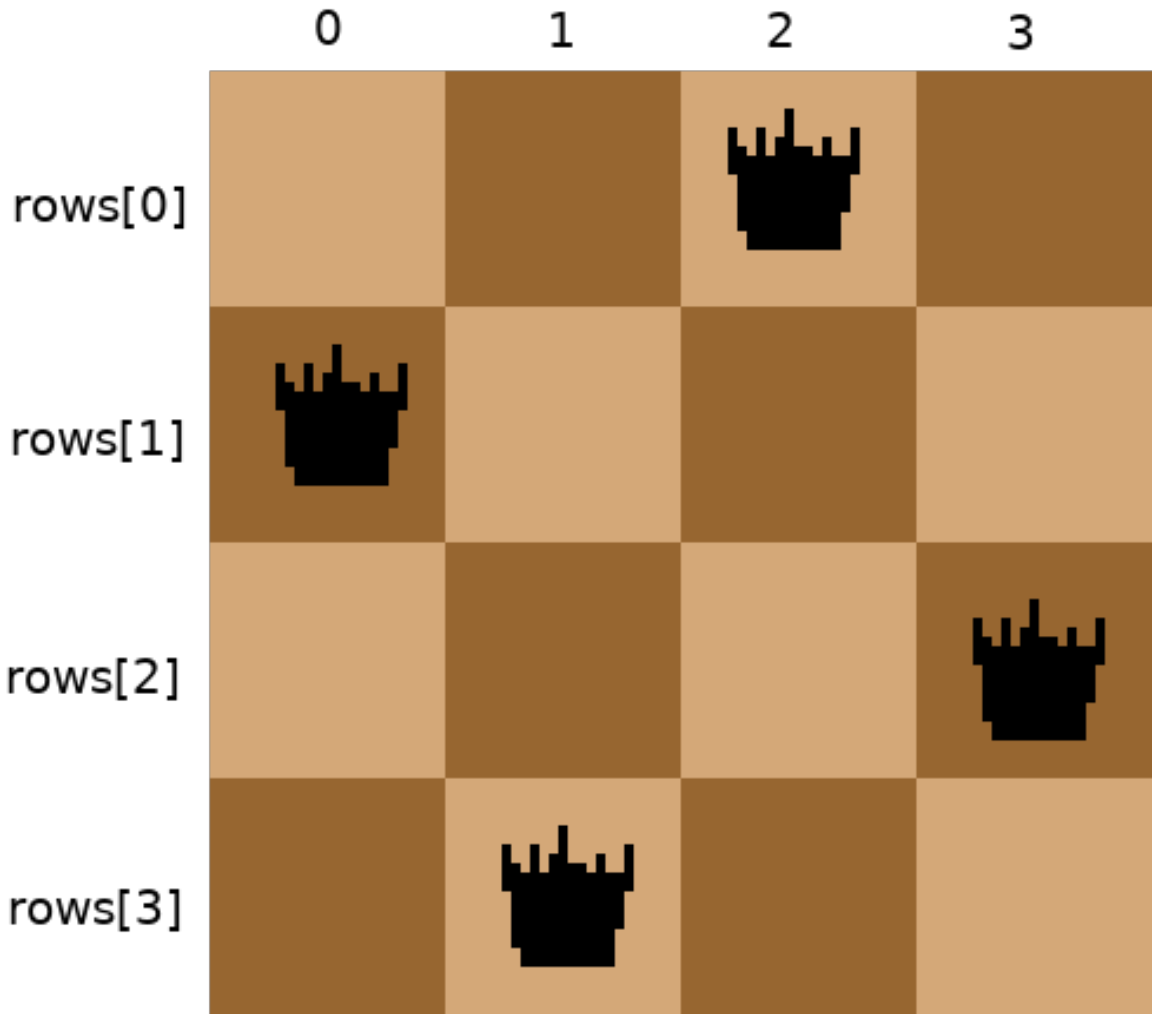


(a) Richtig



(b) Falsch

impl Sequence



impl Sequence

```rust
pub struct Queens {
    n: usize,
    rows: Vec<usize>,
}

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

impl Sequence

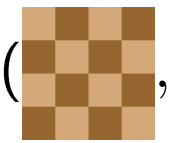
```rust
fn satisfies_condition(&self) -> bool {
    if self.rows.is_empty() {
        return true;
    }

    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

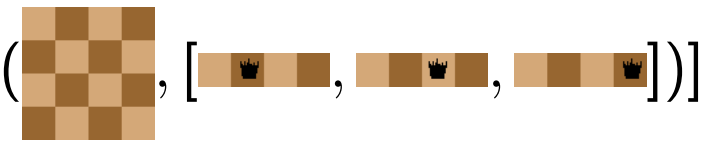
```
fn next_steps(&self) -> Self::Steps {
    0..self.n
}

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

```rust
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 = [ ]

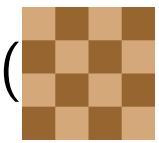
states = [( , [ ,  ,  , ])]

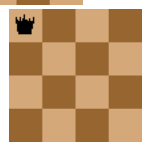
```rust
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        // <- we are here
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();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

$states =$ 

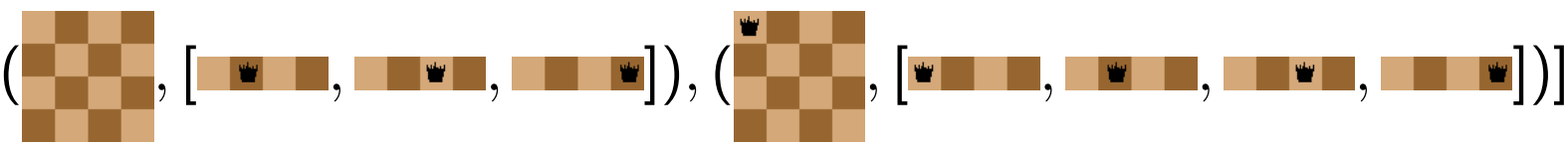
$step =$ 

```
// ...
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();
    }
}
// ...
```

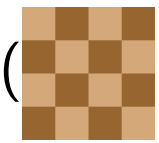
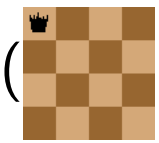
states = [( , [ ,  , ])]

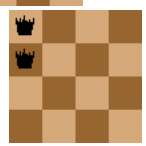
next_state = 

```rust
// ...
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();
    }
}
// ...
```

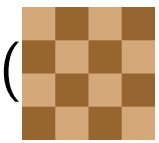
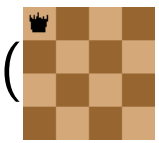
$states =$ 

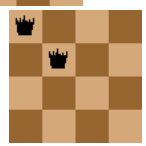
```
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();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

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

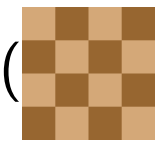
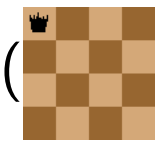
next_state = 

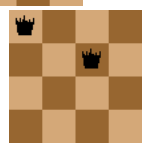
```rust
// ...
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();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

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

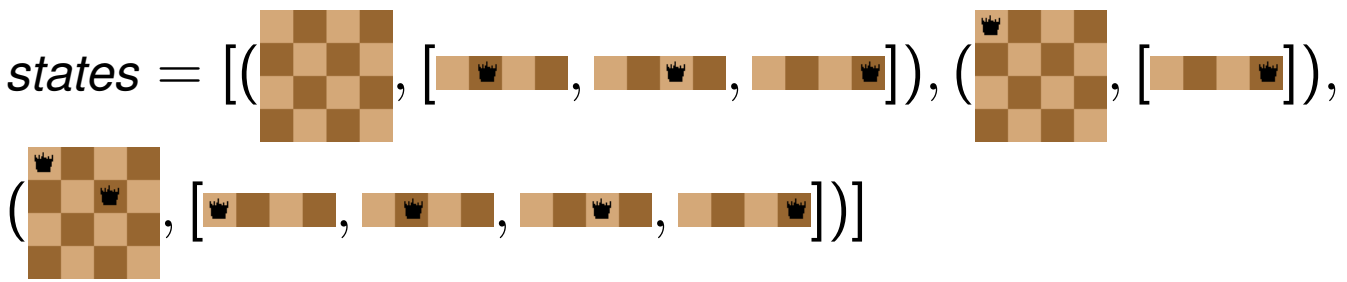
next_state = 

```
// ...
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();
    }
}
// ...
```

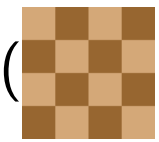
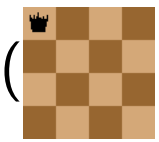
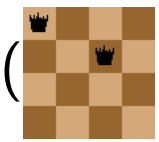
states = [( , [ ,  , ]), ( , [])]

next_state = 

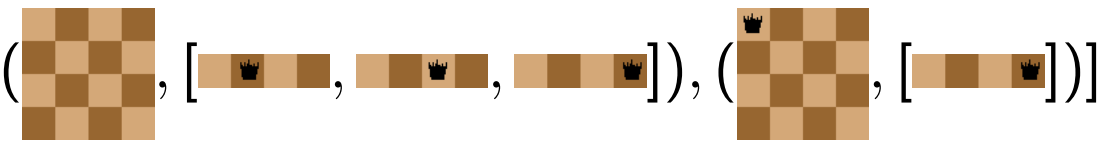
```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
     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

$states =$ 

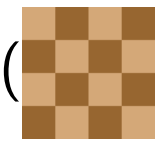
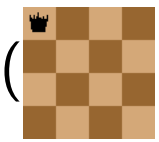
```rust
while let Some((state, steps)) = states.last_mut() {
    // <- we are here after discarding all 4 steps
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps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

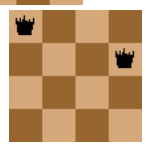
$states =$ [( , [ ,  , ]), ( , []), ( , [ ])]

```
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
    }
}
```

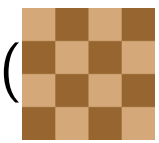
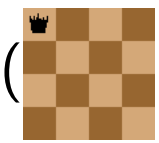
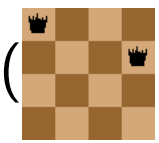
$states =$ 

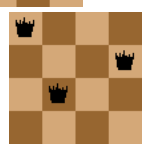
```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        // <- we are here
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();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

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

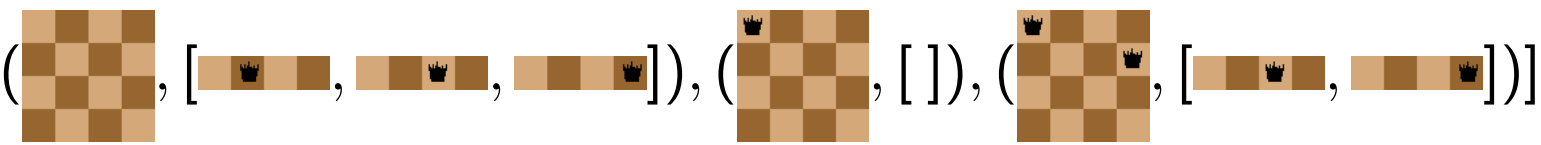
next_state = 

```rust
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();
    }
}
```

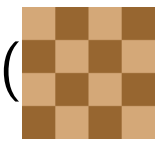
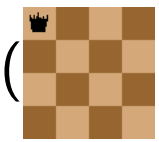
states = [( , [ ,  , ]), ( , [ ]), ( , [ , ])]

next_state = 

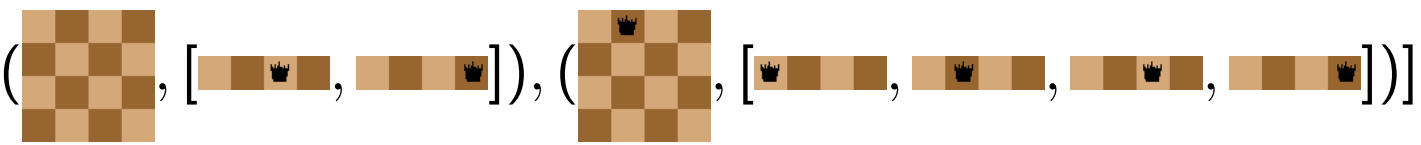
```rust
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
 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

$states =$ 

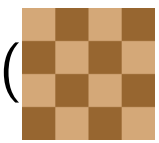
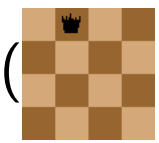
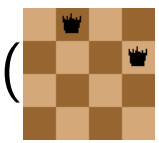
```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
     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

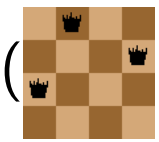
$states =$ [( , [ ,  , ]), ( , [ ]),]

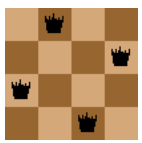
```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
     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

$states =$ 

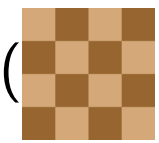
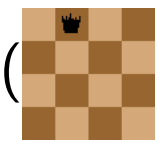
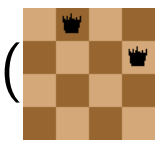
```rust
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            // <- we are here
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();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

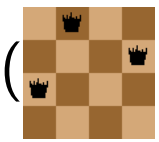
states = [( , [ , ]), ( , [ ]), ( , [ ,  , ]),

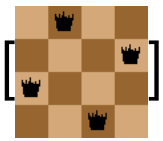
( , [])]

next_state = 

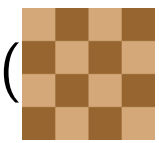
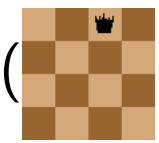
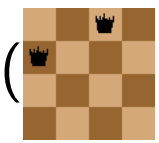
```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
     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

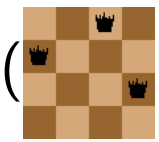
states = [( , [ , ]), ( , [ ]), ( , [ ,  , ]),

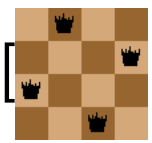
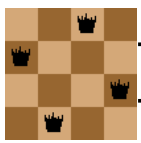
( , [])]

results = []

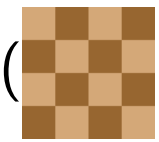
```
while let Some((state, steps)) = states.last_mut() {
    if let Some(step) = steps.next() {
        let next_state = state.apply_step(step);
        if next_state.satisfies_condition() {
            if states.len() < n {
                let next_steps = next_state.next_steps().into_iter();
     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

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

( , [ , ])]

results = [ , ]

```rust
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
    }
}
```

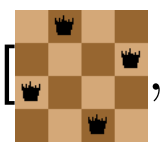
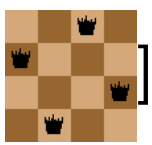
$states = [($  $, [\,])]$

```
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
```

$states = [\,]$

$results = [$  ,  $]$

Listings

Listings

empty

Listings

Listings

empty

Thanks for listening.

Any questions?

References

References I