Backtracking

Ausgewählte Kapitel aus "The Art of Computer Programming"

Bastian Kauschke
Hardware-Software-Co-Design, Friedrich-Alexander-Universität Erlangen-Nürnberg
10. 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

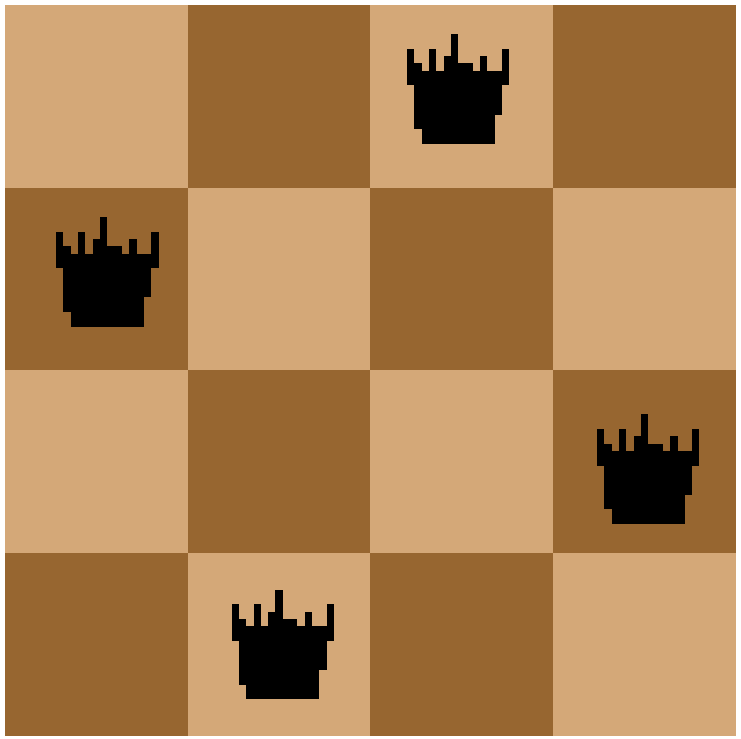
```rust
pub fn b<T: Sequence>(initial: T, n: usize) -> Vec<T> {
    let mut results = Vec::new();
    let mut states = Vec::new();
    let steps = initial.next_steps().into_iter();
    states.push((initial, steps));

    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
}
```

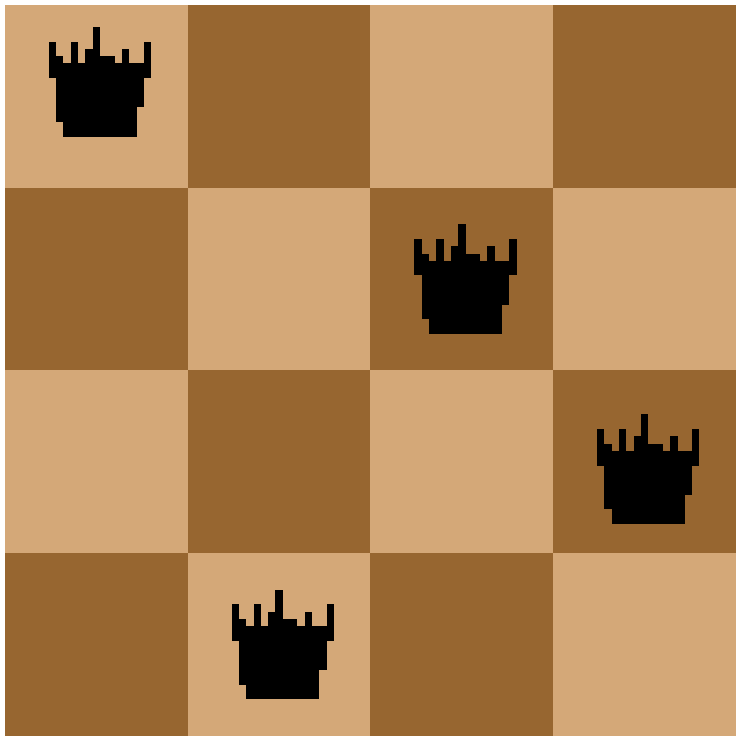
n Queens

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

Listings

Listings

empty

Thanks for listening.

Any questions?

References

References I