Eight queens problem

Janik Dehnhardt

December 20, 2024

Contents

1 Introduction

1.1 Grundlegende problemstellung

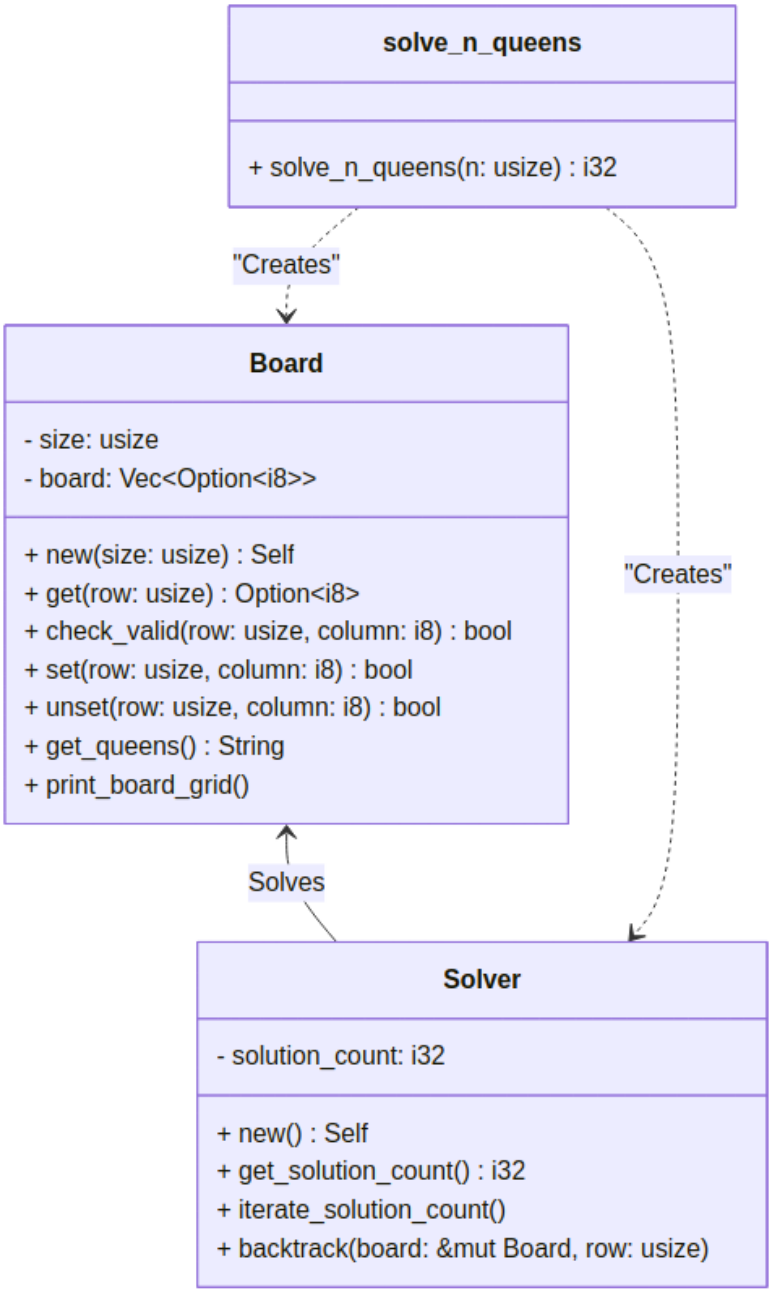
Das 8 Queen problem besteht aus einem 8x8 Schachbrett. Bei den queen Problemen ist die Größe des Boards nxn wenn die Anzahl der zu platzierende Queens n ist. Um das problem zu lösen muss man entweder eine oder alle mögliche lösungen finden. Eine lösung ist valide wenn 8 queens auf Board sind und keine der queens direct eine andere angreifen kann. In diesem fall werden alle mögliche Lösungen gesucht.

1.2 Constraint satisfaction

- Reiheneinschränkung: Jede Dame wird in einer eindeutigen Reihe platziert, was bedeutet, dass keine zwei Damen dieselbe Spalte teilen können.
- Spalteneinschränkung: Jede Dame wird in einer eindeutigen Spalte platziert. Diese Einschränkung wird durch eine Funktion erfüllt, die jede Reihe auf eine passende Spaltennummer überprüft.
- Diagonaleinschränkungen: Keine zwei Damen dürfen sich auf derselben Diagonalen befinden. Das bedeutet, dass für zwei Damen an den Positionen (r_1, c_1) und (r_2, c_2) gilt:
  - Sie dürfen nicht die Bedingung $r_2 - r_1 = c_2 - c_1$ erfüllen (gleiche Diagonale mit "positiver Steigung").
  - Sie dürfen nicht die Bedingung $r_2 - r_1 = -(c_2 - c_1)$ erfüllen (gleiche Diagonale mit "negativer Steigung").

1.3 Wie wird es modelliert

- Die state des Boards wird anhand von der Board class dargestellt.
- Diese stellt den State der Queens auf dem Board anhand von einem Vector da.
  - Beispielsweise: $Q_1 = 1, Q_2 = 3$ bedeutet, dass die erste Dame in der ersten Zeile, erste Spalte, und die zweite Dame in der zweiten Zeile, dritte Spalte steht.
- die Constraints stellen sicher, dass keine zwei Damen dieselbe Spalte oder Diagonale teilen.
- Außerdem wird der state des solvers anhand der solution count dargestellt.



1.4 was gibt das Programm im Erfolgs/Fehlerfall aus

In einem Erfolgsfall werden all mögliche lösung ausgegeben und auch die Anzahl and gefundene lösungen. In einem Fehlerfall (Z.b wenn eine nicht implementierte anzahl von queens dem solver gegeben werden kommt eine Fehlermeldung die dies sagt.

1.5 welche heuristische funktionen werden verwendet?

Es gibt mehrere bekannte heuristische Funktionen die anwendbar sind. Diese Funktionen würden aber nur für die findung einer einzelne lösung nutzlich sein oder für eine gröSSeres n Queen Problem, da diese uns helfen würden die einfachste lösung zu finden indem wir Z.b. nur die Queens mit den wenigsten constraints setzen. Wir wollen aber alle lösung finden also wurden sie hier nicht angewandt.[1] Mögliche Heuristische Funktionen:

- Minimum Remaining Values(MRV):
  - Use the variable with the least amount of values left
- Most constraining variable(MCV)
  - Which variable is used in the most amount of constraints
- Least constraining value(LCV)
  - Prioritize the value selection based on their accumalated effect on all domains
- Simulated Annealing
- Tabu search
- Genetic algorithms

2 Implementation

2.1 Structure

The actual structure of the board consists of the size saved as an int and a board saved as a vector of `N` length. Each Index of the vector represents a row and the value the column that a queen of that row sits on. The vector has a **`value`** or **`None`** at each point representing the column that that queen is in or that no queen has been placed there yet.

```rust
#[derive(Debug)]
struct Board {
    size: usize,
    board: Vec<Option<i8>>, // Each row points to a column (None if no queen is present)
}
```

```rust
impl Board {
    /// Constructor to initialize the board
    fn new(size: usize) -> Self {
        Self {
            size,
            board: vec![None; size],
        }
    }
```

2.2 getter

the getter used to retrieve the column value of a queen after providing the queens row

```rust
/// Getter for column of queen in provided row
fn get(&self, row: usize) -> Option<i8> {
    if row < self.size  {
        self.board[row]
    } else {
        None
    }
}
```

2.3 Constraints

2.3.1 row and column constraint

Two queens on the same row would cause them to attack eachother, we have implemented via the structure the constraint that only one queen can exist per row. As the Vector we use to model the queen placement can only have one number per index.

Jede Reihe auf eine passende Spaltennummer überprüft, wenn eine gefunden wird ist ein conflict vorhanden.

```
/// checks for column conflicts
 fn check_column(&self, column: i8) -> bool {
     !self.board.iter().any(|&col| col == Some(column))
 }
```

2.3.2 Diagonal constraint

It needs to be checked whether queens are diagonal from one another. Given a coord this uses the diagonal function to check that no other queens are on the diagonal.

```
/// returns true if diagonal is fine
fn check_diagonal(&self, row_one: usize, col_one: i8, row_two: usize, col_two: i8) -> bool {
    // Calculate differences
    let col_diff = (col_one - col_two).abs();
    let row_diff = (row_one as i8 - row_two as i8).abs();
    !(row_diff == col_diff) // Return whether the diagonal placement is valid
}

/// returns true if all queen diagonals don't conflict with coord
fn check_all_diagonal(&self, row: usize, col: i8) -> bool {
    for q_row in 0..(self.size-1) {
        if let Some(q_col) = self.get(q_row) {//handles the case where q_col is None
            if !self.check_diagonal(row, col, q_row, q_col) {
                return false;
            }
        }
    }
    return true;
}
```

2.3.3 Check constraints

combines all above checks into one function for ease of use and additionally checks that given column doesn't conflict with any other queens.

```
/// uses all checks to check if coordinate is valid
fn check_valid(&self, row :usize,column: i8) -> bool {
    if !(self.check_column(column)) {
            return false;
    } else if !(row < self.size && column >= 0 && column < self.size as i8) {
            return false;
    } else if !self.check_all_diagonal(row, column) {
            return false;
    }
    return true;
}
```

2.4 setters

Set a queen, checking whether the position is a valid queen placement first

```rust
/// Setter to place a queen at a specific column and row
fn set(&mut self, row: usize, column: i8) -> bool {
    if self.check_valid(row, column) {
        self.board[row] = Some(column);
        return true;
    } else {
        println!("Invalid Queen position: column={}, row={}. Board size is {}.",column, row,
        ↪  self.size);
        return false;
    }
}
```

Unset a queen, only checking whether the position exists on the board

```rust
/// unsetter to remove a queen at a specific column and row
fn unset(&mut self, row: usize, column: i8) -> bool {
    if !(row < self.size && column >= 0 && column < self.size as i8) {
        println!("Invalid position: row={}, column={}. Board size is {}.",row, column,
        ↪  self.size);
        return false;
    } else {
        self.board[row] = None;
        return true
    }
}
```

2.5 printing state

returns string with all queens nicely formatted

```rust
/// Get state of all queens via a string
fn get_queens(&self) -> String {
    let queens: Vec<String> = self
        .board
        .iter()
        .enumerate() // Get (row, Option<column>) for each row
        // Map to formatted string if column exists
        .filter_map(|(row, &col)| col.map(|c| format!("{{{}, {}}}", row, c)))
        .collect();
    format!("Queens: [{}]", queens.join(", "))
}
```

print the queens string

```rust
/// print queens state to terminal
fn print_queens(&self) {
    println!("{}", self.get_queens());
}
```

prints a nice board layout with queens state

```rust
/// nicer printing of current queen state
fn print_board_grid(&self) {
    print!("===");
    for i in 0..self.size {
        print!("{i}==")
    }
    println!();
    for col in 0..self.size {
        print!("{col}|");
        for row in 0..(self.size as i8) {
            if let Some(queen_col) = self.get(col) {
                if queen_col == row {
                    print!(" Q "); // Queen
                } else {
                    print!(" . "); // Empty
                }
            } else {
                print!(" . "); // Empty
            }
        }
        println!("|"); // Newline after each row
    }
    for _i in 0..self.size {
        print!("===")
    }
    println!("===");
}
```

2.6 Implementing the solver

Most of the work is already done as I can already check for validity and the board has already been modelled with the Board struct. This implements the backtracking solution which finds all possible solutions by repeatedly backtracking till all solutions have been found.

> This could be improved in the future possibly by only finding unique solutions. This would reduce the solutions for the 8 queen problem from 92 to 12 by comparing solutions by rotating and mirroring.

```
struct Solver {
    solution_count: i32,
}


impl Solver {
    /// Constructor to initialize the solver
    fn new() -> Self {
        Self {
            solution_count: 0,
        }
    }
    fn get_solution_count(&self) -> i32 {
        return self.solution_count;
    }
    fn iterate_solution_count(&mut self) {
        self.solution_count += 1;
    }

    /// the backtrack solution for queen 8x8 problem
    fn backtrack(&mut self, board: &mut Board, row: usize) {
        if row == board.size {
            board.print_board_grid();
            board.print_queens();
            self.iterate_solution_count();
            return
        }
        for col in 0..board.size as i8  {
            if board.check_valid(row, col) {
                board.set(row, col);
                self.backtrack(board, row + 1);
                board.unset(row, col);
            }
        }
        return
    }
}
```

This function is just implemented for ease of use. It sets up the board and the solver with the right size and lets you decide what size of queen problem you want to solve.

```rust
/// solve 8x8 queens
pub fn solve_n_queens(n: usize) -> i32 {
    let mut board = Board::new(n);
    let mut solver = Solver::new();
    if n <= 10 && n > 0 {
        solver.backtrack(&mut board, 0);
        println!("Solved {n}x{n} Queens problem with {} solutions!",
        ↪  solver.get_solution_count());
        return solver.get_solution_count();
    } else if n < 1{
        println!("Can't solve queen problem for a board that doesn't exist")
    } else {
        println!("Solving for larger than 10 hasn't been implemented yet");
        return 0;
    }
}
```

```rust
fn main() -> Result<(), String> {
    solve_n_queens(8);
    Ok(())
}
```

2.7 tests

Some tests that were implemented that check that functions work the way they are intended to.

```rust
 #[test]
fn create_queen() {
    let mut board =  Board::new(8);
    let create = board.set(0, 0);
    assert_eq!(create, true);
}
```

```rust
#[test]
fn diagonal_found() {
    let mut board =  Board::new(8);
    board.set(0, 0);
    let bad_coord = board.set(1, 1);
    assert_eq!(bad_coord, false);
    assert_eq!(board.check_diagonal(0,0, 1,1), false);
}
```

```rust
#[test]
fn conflicting_column_found() {
    let mut board =  Board::new(8);
    board.set(1, 2);
    assert_eq!(board.check_column(1),true);
    assert_eq!(board.check_column(2),false);
    assert_eq!(board.check_column(3),true);
}
```

```rust
#[test]
fn too_big() {
    let board =  Board::new(8);
    let too_big = board.check_valid(0, 8);
    let other_too_big = board.check_valid(8, 0);
    let way_too_big = board.check_valid(8, 8);
    let okay_size = board.check_valid(0, 0);
    assert_eq!(too_big, false);
    assert_eq!(other_too_big, false);
    assert_eq!(way_too_big, false);
    assert_eq!(okay_size,true);
}
```

```rust
#[test]
fn too_small() {
    let board =  Board::new(8);
    let too_small = board.check_valid(0, -1);
    let way_too_small = board.check_valid(0, -90);
    assert_eq!(too_small, false);
    assert_eq!(way_too_small, false);
}
```

```rust
#[test]
fn output_test() {
    let mut board =  Board::new(8);
    board.set(0, 0);
    let mut queens = board.get_queens();
    assert_eq!(queens,"Queens: [{0, 0}]");
    board.set(1, 4);
    queens = board.get_queens();
    assert_eq!(queens,"Queens: [{0, 0}, {1, 4}]");
    board.set(2, 7);
    board.set(3, 5);
    queens = board.get_queens();
    assert_eq!(queens,"Queens: [{0, 0}, {1, 4}, {2, 7}, {3, 5}]");
    board.set(4, 2);
    board.set(5, 6);
    queens = board.get_queens();
    assert_eq!(queens,"Queens: [{0, 0}, {1, 4}, {2, 7}, {3, 5}, {4, 2}, {5, 6}]");
    board.set(6, 1);
    board.set(7, 3);
    queens = board.get_queens();
    assert_eq!(queens,"Queens: [{0, 0}, {1, 4}, {2, 7}, {3, 5}, {4, 2}, {5, 6}, {6, 1}, {7,
    ↪  3}]");
    println!("Test getter:");
    assert_eq!(board.get(0), Some(0));
    assert_eq!(board.get(1), Some(4));
    assert_eq!(board.get(2), Some(7));
    assert_eq!(board.get(3), Some(5));
    assert_eq!(board.get(4), Some(2));
    assert_eq!(board.get(5), Some(6));
    assert_eq!(board.get(6), Some(1));
    assert_eq!(board.get(7), Some(3));
}
```

References

[1] Ivica Martinjak and M Golub. "Comparison of Heuristic Algorithms for the N-Queen Problem". In: ResearchGate (July 2007). DOI: 10.1109/ITI.2007.4283867. URL: https://www.researchgate.net/publication/4266144_Comparison_of_Heuristic_Algorithms_for_the_N-Queen_Problem (visited on 12/18/2024).

[2] "Constraint-Satisfaction-Problem". In: Wikipedia (May 2022). URL: https://de.wikipedia.org/w/index.php?title=Constraint-Satisfaction-Problem&oldid=222915774 (visited on 12/20/2024).

[3] "Constraint Satisfaction Problem". In: Wikipedia (Nov. 2024). URL: https://en.wikipedia.org/w/index.php?title=Constraint_satisfaction_problem&oldid=1259275985 (visited on 12/20/2024).

[4] Marc Toussaint. "Artificial Intelligence Constraint Satisfaction Problems". In: (2019). URL: https://www.user.tu-berlin.de/mtoussai/teaching/19-ArtificialIntelligence/08-CSP.pdf.

[5] Constraint Satisfaction Problems (CSP) in Artificial Intelligence. Oct. 2024. URL: https://www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/ (visited on 12/20/2024).