EXERCISE REPORT

TEAM 15

BACKTRACKING

Lecturer: Son Nguyen Thanh MCS

Class: CS112.N21.KHTN

Member 1: Quan Vo Minh – 21520093

Member 2: Nhat Nguyen Viet – 21520378

Excercise: Team 7's Excercise

Date: 15/04/2023

EXERCISE 1:

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other

Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' Both indicate a queen and an empty space, respectively.

SOLUTION:

The queens can attack each other if they are in the same row, column or diagonal. So we assign each queen to a row (queen i belongs to row i), then choose their column.

The backtrack function starts with first queen, we iterate through n columns and check if current position valid, then we place first queen there and call backtrack function on second queen and so on.

Algorithm:

1) Start in the topmost row.
2) Make a recursive function which takes state of board and the current row number as its parameter.
3) Fill a queen in a safe place and use this state of board to advance to next recursive call, add 1 to the current row. Revert the state of board after making the call.
    a) Safe function checks the current column, left top diagonal and right top diagonal.
    b) If no queen is present then fill else return false and stop exploring that state and track back to the next possible solution state
4) Keep calling the function until the current row is out of bound.
5) If current row reaches the number of rows in the board then the board is filled.

6) Store the state and return.

Implementation: Link

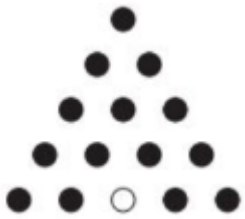
Pseudo code:

```
// Input: n = number of queens
// Output: Possible placements of n queens

//State is an array which State[i] stores column of queen i, so position is (i, State[i])
Backtrack(State, Row)
        // Input: State of the chessboard and current row/queen
        If Row > n then Write(State)
        Else
                for each Column in 1..n do
                        If Safe(State, Row, Column) then
                                State[i] ← Column
                                Backtrack(State, Row + 1)

Safe(State, Row, Column)
        // Input: State of chessboard, Row and Column
        // Output: If the position (Row, Column) is safe in State
        for i in [1..Row − 1] do
                prevcol ← State[i]
                prevrow ← i
                If (prevrow, prevcol) is in same column or diagonal with (Row, Column) then
                        Return False
        Return True
```

EXERCISE 2:

Puzzle Pegs:

This puzzle-like game is played on a board with 15 small holes arranged in an equilateral triangle. In an initial position, all but one of the holes are occupied by pegs, as in the example shown below. A legal move is a jump of a peg over its immediate neighbor into an empty square opposite; the jump removes the jumped-over neighbor from the board.



Design and implement a backtracking algorithm for solving the following versions of this puzzle.

a. Starting with a given location of the empty hole, find a shortest sequence of moves that eliminates 14 pegs with no limitations on the final position of the remaining peg.
b. Starting with a given location of the empty hole, find a shortest sequence of moves that eliminates 14 pegs with the remaining peg at the empty hole of the initial board.

SOLUTION:

Firstly, we need to model the problem in a way that a computer can store and solve. In our solution, we use a matrix 5×9 to present the equilateral triangle. As presented in *Figure 1*, the triangle is modelized with colored cells (i.e. purple, blue, and yellow), with yellow is the free hole (not occupied by the pegs). The uncolored cells are the boundary or the regions that pegs can not jump into.

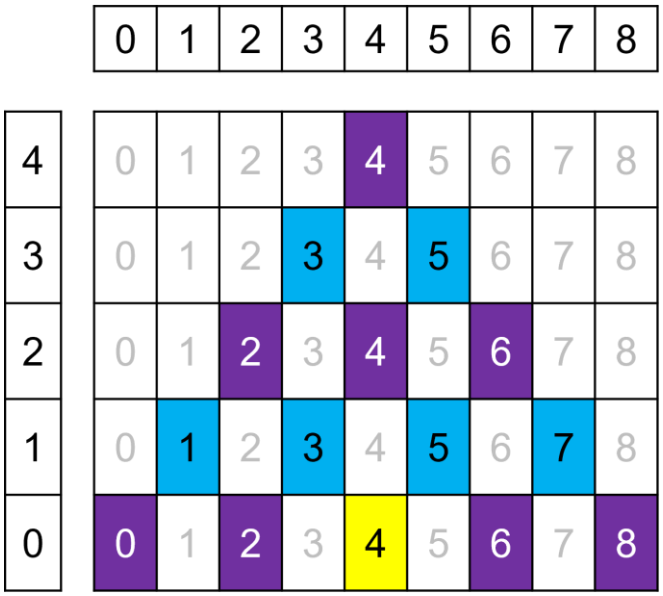


Figure 1: Modeling Problem

"A legal move is a jump of a peg over its immediate neighbor into an empty square opposite; the jump removes the jumped-over neighbor from the board." As described, we can now easily realize that one peg can jump (over its immediate neighbor) anywhere as long as its destination is an empty square and not a forbidden cell (gray). For example, [0, 0] peg can jump to [0, 4] and removes the [0, 2] peg.

Each peg ([x][y]) has 8 available directions to jump, including:

1. Upper diagonal: [x+4, y],
2. Upper right diagonal: [x+2, y+2]
3. Upper left diagonal: [x+2, y-2]
4. Left: [x, y-4]
5. Right: [x, y+4]
6. Lower diagonal: [x-4, y]
7. Lower right diagonal: [x-2, y+2]
8. Lower left diagonal: [x-2, y-2]

Algorithm:

The algorithm for this problem is similar to the n-queens problem. We propose to prune the branches with $len(move\_sequence) + \#remain\_pegs \geq len(shortest\_path)$.

Implement: puzzle_pegs.py

Pseudo-code:

```
initialize_board()

// Create an empty list to store the sequence of moves
move_sequence = []

// Perform a depth-first search to find the shortest sequence of moves to
solve the game
def backtrack(board, depth, move_sequence):
    // Base case: the game is solved
    if is_game_over(board):
        return move_sequence

    // Recursive case: try all possible moves
    shortest_sequence = None
    for peg_to_move in get_all_pegs(board):
        for direction in get_all_move_directions():
            if is_valid_move(board, peg_to_move, direction):
                // Make the move
                new_board = make_move(board, peg_to_move, direction)
                new_move_sequence = move_sequence + [(peg_to_move, direction)]

                // Recursively search for the shortest sequence of moves
                new_sequence = backtrack(new_board, depth + 1,
                                         new_move_sequence)

                // Update the shortest sequence if a shorter one is found
                if new_sequence is not None and (shortest_sequence is None or
                   len(new_sequence) < len(shortest_sequence)):
                    shortest_sequence = new_sequence

    return shortest_sequence

// Call the depth-first search function to find the shortest sequence of moves
shortest_sequence = backtrack(board, 0, [])

// Display the shortest sequence of moves
display_move_sequence(shortest_sequence)
```