
	<p>Operating Systems 2 (Fall 2023) Project Discussion</p>	 <p>كلية الحاسبات والذكاء الاصطناعي Faculty of Computers & Artificial Intelligence</p>
---	---	---

Project Number: **Project 4**

Project Name: **N-Queen problem**

Team Members¹:

	Team Member ID	Team member name (in Arabic)	Grade
1	20210243	تقى محمد محمد ابوالفتوح توفيق	
2	20210565	عبدالواحد رجب عبد الواحد حماد	
3	20210531	عبدالرحمن مصطفى حامد محمود	
4	20210261	جومانا محمد السيد طلبه	
5	20210166	الاء حسن عبدالرحمن عبدالرحمن	
6	20210172	الاء ممدوح احمد عثمان	
7			

Evaluation Criteria

General Criteria

Criteria		Grade
Multithreading (5)	No multithreading (2 out of 5)
	Threads in serial (3 out of 5) Correct usage of threads, and synchronization mechanisms	
	Multithreading (4 or 5 out of 5) Correct usage of threads, and synchronization mechanisms	
GUI (2)	No GUI (0 out of 2)
	GUI without thread communication or realtime update (1 out of 2)	
	GUI with correct I/O and Thread communication or realtime update (2 out of 2)	
Documentation (1)		
Understanding (2)		

1.0 Project Description

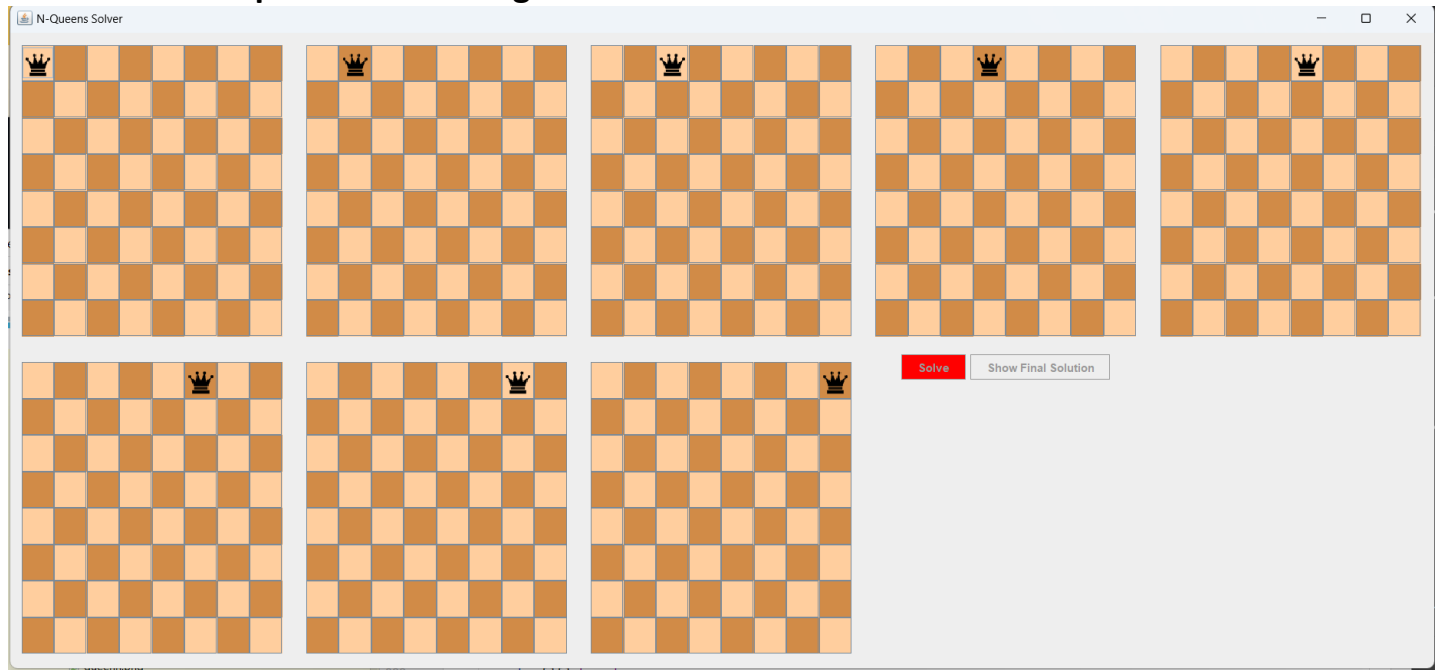
The project is a java application which runs N-Queen problem using N-threads.

All the threads starts from row 0 , with the queen initially placed in the column corresponding to the thread's ID, and all of the threads try to find the solution and check if the position is valid using backtracking algorithm by recursion.

The first solution found by the threads will be stored as the final solution.

And all of these actions are visually represented through real-time updates in a Swing GUI.

Here is an example of the starting state:



Example of ending state:



And between them there is a real-time update GUI

2.0 Solving Steps and used tools

1. At first we had to try to implement the backtracking algorithm for the normal sequential problem.
Using two functions , one of them checks if the position valid , and the other one responsible for the backtracking process .
2. Then we had to solve the problem using threads
We created a class nqeenthread which of type thread ,
and we used
 - a. Cyclic barrier: to ensure that all threads starts at the same point of time
 - b. Mutex: to synchronize the access of the variable “count” that all threads increment
 - c. Atomic variable: to catch the first solution
 - d. Compare and test: allows the threads to change the value of the atomic variable
3. The last step was the GUI , we had to create a GUI that updated at the real-time , using swing .. by implementing functions to show the solution steps performed by each thread , and then show the solution that had been caught first by the threads

3.0 Team members role

Team members name	Role
الاء حسن عبدالرحمن عبدالرحمن الاء ممدوح احمد عثمان عبدالواحد رجب عبدالواحد	GUI
تقى محمد محمد ابو الفتوح توفيق جومانا محمد السيد طلبية عبدالرحمن مصطفى حامد محمود	Multithreaded

Then the multithread has been linked to the GUI and documented the code.

4.0 Code Documentation

1.GUI Class prototypes:

- a. **private void initializeGUI()** : initialize the GUI variables
- b. **private JPanel createChessboardPanel()** : creates chessboard
- c. **private JPanel createControlPanel()**: create the buttons
- d. **public void updateChessboard(char[][] board, int index)**: takes the thread ID

and update the thread chessboard

- e. **private char[][] final_solution()**: checks which thread has found the first solution
- f. **private void showFinalSolution()**: show the final solution in GUI
- g. **private void solveNQueensMultithreaded()**: create n threads each thread

assigned with chessboard

2.Multithreading class prototypes:

- a. **public void run()**: runs when a thread initialized
- b. **public boolean hasSolution()**: checks if the current thread has solution
- c. **private boolean solveNQueens()**: backtracking function
- d. **private boolean isSafe()**: checks if the row, column, diagonal doesn't contain a Queen
- e. **private char[][] get_final_solution()**: returns the solution that had been found first

3.Important functions documentation:

a. run()

Parameters:

None

Description:

Barrier Synchronization:

Attempts to synchronize with a Cyclic Barrier (barrier) for concurrent thread execution. Handles exceptions and returns early if interrupted.

```
try {
    barrier.await();
    System.out.println("process waiting: "+Thread.currentThread().getName());
} catch (InterruptedException | BrokenBarrierException e){
    System.out.println("ERROR IN THRED NUMBER
:"+Thread.currentThread().getName());
    return;
}
```

N-Queens Solution Search:

Invokes the solveNQueens method, initiating a recursive backtracking algorithm to find a valid placement.

```
solveNQueens(0, this.id);
```

Solution Handling:

Updates local state upon finding a solution. If the first thread to find a solution, sets global variables (id_of_thread, final_sol).

```
synchronized (this) {
    if (this.hasSolution == true &&
firstSolutionFound.compareAndSet(false, true))
    {
        System.out.println(Thread.currentThread().getName()
+ "has found solution");
        id_of_thread = this.id;
        nqueen.final_sol = this.board;
        solutionFound = true;
    }
}
```

Solution Counting:

Uses a semaphore (mutex) to ensure mutual exclusion when incrementing a shared count variable. Displays final solution if all threads complete execution.

```
try {  
    mutex.acquire();  
    System.out.println("mutex acquired"+Thread.currentThread().getName());  
    synchronized (count) {  
        if (count.incrementAndGet() == N) {  
            System.out.println(count);  
            char[][] x = final_solution();  
            printBoard(x);  
            showFinalSolution();  
        }  
    }  
} catch (InterruptedException e) {  
    // exception handling code  
}  
finally {  
    mutex.release();  
    System.out.println("mutex released"+Thread.currentThread().getName());  
}
```

Exception Handling:

Handles interruptions during mutex operations.

b. Updatechessboard()

Parameters:

Char[][] board : the chessboard array after the update

Index : thread ID to get the chessboard assigned to the thread

Description:

This function effectively updates the visual representation of the chessboard panel based on the provided board configuration, handling both icon clearing and updating based on the board state.

Panel Retrieval:

The function retrieves the JPanel representing the chessboard at the specified index from the container using `getComponent(index)`.

```
JPanel chessboardPanel = (JPanel) getContentPane().getComponent(index);
```

Icon Clearing:

It iterates through the components of the chessboard panel and sets the icon of each JButton component to null, effectively clearing any existing icons from the squares.

```
Component[] components = chessboardPanel.getComponents();
for (Component component : components) {
    JButton square = (JButton) component;
    square.setIcon(null);
}
```

Board Configuration Iteration:

The function iterates through the 2D `board` array to update the visual representation of each square on the chessboard panel based on the board configuration.

Queen Icon Handling:

When encountering a 'Q' (queen) on the board, it creates an ImageIcon for the queen image, resizes it to 30x30 pixels using `Image.SCALE_SMOOTH`, and sets it as the icon for the corresponding square JButton.

Background Color Setting:

For positions without a queen, it sets the background color of the square based on its position using a light or dark color, determined by the sum of the row and column indices.

c. solveNQueensMultithreaded()

Parameters:

None

Description:

This function initiates a multithreaded solving process for the N-Queens problem by creating separate threads to handle the computation for each chessboard configuration.

Disabling Solve Button:

It disables the solve button to prevent multiple simultaneous executions, and changes its background color to red to indicate that the solving process is ongoing.

Clearing Chessboards:

It clears any existing chessboard configurations to prepare for the new solution.

Resetting Control Variables:

It resets the `stopThreads` flag to false and sets the `solutionsCount` to 0. Additionally, it sets the title of the application window to "N-Queens Solver".

Creating and Starting Threads:

It iterates through the number of chessboards `N` and creates a new thread for each board using the `nqueen` class, passing the board configuration and a barrier object. It then starts each thread and adds it to a list of threads for later reference.

```
for (int i = 0; i < N; i++) {  
    char[][] board = boards[i];  
    int finalI = i;  
    Thread thread = new nqueen(i,board,N,this.barrier);  
    thread.start();  
    threads.add((nqueen) thread);  
}
```


c. solveNQueens()

Parameters:

row: The current row being processed in the N-Queens algorithm.

initial_col: The initial column position where a queen is placed to start

Description:

The function is a recursive backtracking algorithm to solve the N-Queens problem , it's called N times at the first by each thread created (the thread number = initial_col), The algorithm continues this process until all possible combinations have been explored or a solution is found.

Thread Interruption Check:

It checks if the current thread has been interrupted. If so, it returns false.

Base Case - Solution Found:

If the algorithm has successfully placed N queens on the board(current row == N), so the solution is found and it sets "hasSolution" to true and returns true.

Recursive Backtracking and GUI updates:

It iterates through each column in the current row, checking if it's safe(checks row , column and diagonal of current row) to place a queen in that position, and if it's safe it updates the chessboard, visualizes the update, and then recursively calls "solveNQueens" for the next row.

```
for (int col = 0; col < N; col++) {
    if (isSafe(row, col)) {
        if(row==0){
            board[row][intial_col]='Q';
            updateChessboard(this.board,this.id);
        }
        else
            board[row][col]='Q';
            updateChessboard(this.board,this.id);
    }
}
```

Queen Removal (Backtracking):

If placing a queen in the current position does not lead to a solution in the subsequent rows, it backtracks by removing the queen from that position and continues with the next column.

```
if(col==intial_col && row==0){  
    board[row][intial_col]='_';  
    break;  
}  
board[row][col]='_';
```

Thread Visualization Delay:

The visualization and sleep statements are used for displaying the progress of the algorithm at each step.

```
try {  
    this.sleep(274);  
} catch (InterruptedException e) {  
    e.printStackTrace();  
}
```

c. isSafe()

Parameters:

row: The row of the potential queen placement.

col: The column of the potential queen placement.

Description:

The function checks the passed row and column of square if it's safe place or not by checking that there is no other queen is placed in that row , col or diagonal of that square.

Checking N row squares:

Iterate through each column in the specified row (row).

If a queen ('Q') is found in the same column, return false

Checking N column squares:

Iterate through each row in the specified column (col).

If a queen ('Q') is found in the same row, return false

Checking N diagonal squares:

The checking is done through two directions according to the place of passed row and col square

First direction : (Top-Left to Bottom-Right)

Iterate diagonally from the current position (row, col) towards the top-left.

If a queen ('Q') is found in the diagonal path, return false .

Second direction : (Top-Right to Bottom-Left)

Iterate diagonally from the current position (row, col) towards the top-right.

If a queen ('Q') is found in the diagonal path, return false .

```
for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
    if (board[i][j] == 'Q') {
        return false;
    }
}
for (int i = row, j = col; i >= 0 && j < N; i--, j++) {
    if (board[i][j] == 'Q') {
        return false;
    }
}
```