Helwan University – Faculty of Computing & ArtificialIntelligence

Module: Operating System2 –"Semester1" 2025-2026

Project #5 : Make a Square

Project Overview

TetrisSquareSolver is a multi-threaded JavaFX application that solves a 4×4 Tetris-like puzzle using backtracking.
The goal is to place all given pieces on a square board without overlap until the board is completely filled.

The solution explores different piece orderings in parallel using multiple threads to improve performance and visualize the solving process in real tim

1. Project Description

The goal of this project is to build a 4×4 square using 4 or 5 puzzle pieces. Each piece is defined by a binary matrix where 1 represents a solid block and 0 represents empty space. All pieces must be used, and pieces may be rotated (no flipping is allowed according to the problem modeling section).

The program searches for all possible valid solutions that fill the 4×4 board completely without overlap. If no solution exists, the program reports:

No solution possible

The project is implemented using Java, multithreading, and a GUI with real-time updates, following the general guidelines of the Operating Systems 2 course.

2. Input Specification

- The first line contains the number of pieces.
- For each piece:
    - One line with two integers: number of rows and columns.

- Followed by rows of `0` and `1` characters describing the shape.

Example Input:

1
2 3
111
101

Multiple input files (text files) are used to test different cases.

3. Output Specification

- A 4×4 square printed using numbers:
  - Solid cells of piece #1 → 1
  - Solid cells of piece #2 → 2, etc.

Example Output:

1112
1412
3422
3442

- If no arrangement can form a complete square:

  No solution possible

4. Problem Modeling

- The board is modeled as a **fixed 4×4 matrix**.
- Each puzzle piece:
  - Is modeled as a 2D binary matrix.
  - Has multiple possible **rotations** (0°, 90°, 180°, 270°).
- **Flipping is not allowed**.

Multithreading Model

- The **main thread** selects one piece.
- For each possible rotation of that piece, a **worker thread** is created.
- Each thread:
    - Places the selected piece on the board.
    - Performs a **backtracking search** to place remaining pieces.
- The number of active threads is limited based on the number of available CPU cores.

---

5. Algorithm Overview

1. Read input pieces from file.
2. Generate all valid rotations for each piece.
3. Main thread selects an initial piece.
4. For each rotation:
    - Spawn a thread.
    - Try placing the piece at all valid board positions.
5. Use recursive backtracking to place remaining pieces.
6. If the board is completely filled:
    - Save and display the solution.
7. If all threads finish with no solution → print *No solution possible*.

---

6. GUI Description

Input

- Select number of pieces of each type (standard Tetris-like pieces).
- Load input file.

Output

- Multiple 4×4 boards displayed:
    - Each board represents a thread's search path.
    - Real-time updates show placements and backtracking steps.
- Final solution boards are highlighted.

7. What We Actually Did?

- Implemented piece rotation logic.
- Designed thread-safe board placement using synchronization.
- Used Java Threads to explore solution space concurrently.
- Built a Swing-based GUI with real-time visualization.
- Tested the program using multiple input text files.

8. Team Members Roles

- **Karim Mohamed(Member 1):** Input file parsing, piece data structures, and validation of input format.
- **Mohamed Sab3(Member 2):** Piece rotation logic (0°, 90°, 180°, 270°) and generation of all valid orientations.
- **Mohamed khaled (Member 3):** Board representation (4×4 grid), placement checking, and collision detection.
- **Rania Ahmed (Member 4):** Backtracking algorithm for placing remaining pieces and detecting complete solutions.
- **Fatema Hany(Member 5):** Multithreading design (main thread + worker threads), thread synchronization, and performance optimization.
- **Amira Hassan (Member 6):** GUI implementation (Java fx), real-time visualization of threads, and displaying final solutions.

9. Code Documention

Input Parsing (parseInput)

Purpose:

Convert user text input into structured Piece objects.

Key points:

Reads piece dimensions (rows, columns).

Builds a binary matrix (int[][]) for each piece.

Supports multiple pieces separated by blank lines.

Performs validation (dimensions, missing rows, invalid format).

Why important:

Ensures correct and safe input before starting the solving process.

Piece Representation (Piece class)

Purpose:

Represents a Tetris piece and all its possible orientations.

Responsibilities:

Stores piece ID and shape matrix.

Generates all unique rotations (90°, 180°, 270°).

Optionally generates mirrored (flipped) versions.

Avoids duplicate rotations using hashing (Set<String>).

Why important:

Backtracking depends on trying every possible orientation of each piece.

Board Representation (`Board` class)

Purpose:
Represents the 4×4 board and manages piece placement.

Main methods:

- `canPlace(...)`: checks if a piece fits at a given position.
- `place(...)`: places a piece on the board.
- `remove(...)`: removes a piece (undo step).
- `isFull()`: checks if the board is completely filled.

- `copyGrid()`: creates a deep copy for visualization and storage.

Why important:
This class is the core state of the backtracking algorithm.

Backtracking Solver (`Solver` class)

Purpose:
Searches for valid solutions using recursive backtracking.

Algorithm steps:

1. **Take the next piece.**
2. **Try all its rotations.**
3. **Try all valid board positions.**
4. **Place the piece if possible.**
5. **Recurse to the next piece.**
6. **Undo placement if no solution is found.**

Concurrency support:

- **Checks `AtomicBoolean stopRequested` to allow safe stopping.**
- **Uses shared `solutions` and `AtomicInteger solutionsCount`.**

Why important:
Implements the actual solving logic.

Multithreading (`ThreadWorker` class)

Purpose:
Runs the solver independently in a separate thread.

Responsibilities:

- **Receives a unique ordering of pieces.**
- **Owns a dedicated UI panel (`BoardPane`).**
- **Executes the solver logic.**
- **Logs progress and results.**

Design choice:
Implements `Runnable` instead of extending `Thread` to separate task logic from execution control

Thread Management (Main Application)

Approach:

- **One thread per different piece ordering.**
- **Threads are created and started externally.**
- **A monitoring thread waits for all solver threads to finish.**

Why important:
Allows parallel exploration of the search space.

JavaFX UI (`BoardPane`)

Purpose:
Provides real-time visualization of each solver thread.

Features:

- **Canvas-based board rendering.**
- **Color-coded pieces.**
- **Per-thread logs.**
- **Thread-safe UI updates using `Platform.runLater.`**

Thread-Safe Shared Data

Used structures:

- **`AtomicBoolean` → stopping all threads safely.**
- **`AtomicInteger` → counting solutions correctly.**
- **`CopyOnWriteArrayList` → storing solutions safely.**

Why important:
Prevents race conditions in a concurrent environment.