CS1003 Project 3: Nonograms

Group Report

1. Overview

This project focused on developing a graphical Nonogram puzzle solver and player using Java Swing, reinforcing key programming concepts such as object-oriented design, GUI development, and JSON file handling. Working in teams, we collaboratively designed and implemented a modular application capable of loading, solving, and verifying both black-and-white and colour Nonogram puzzles. The primary requirements were met, including core gameplay functionality, puzzle loading and checking, and a user-friendly interface. We also successfully implemented all secondary requirements, such as handling colour puzzles and recording, saving, loading, and undoing user moves. Furthermore, the project achieved tertiary goals by integrating both deductive and guessing-based solvers capable of resolving complex puzzles requiring backtracking. This project also emphasized effective team collaboration, version control through Git, and iterative design practices to ensure a robust and well-tested application.

2. Team Working

2.1 Approach to Communication:

Our team was productive and maintained excellent communication throughout the entire project. Whenever someone had time to work on the project, they would inform the group via our chat, ensuring that no overlapping edits occurred. This minimized merge conflicts and allowed us to discuss the approach being taken—whether it was the most efficient or if improvements could be made. The chat served as a central hub for updates, questions, and feedback, making our workflow smooth and collaborative. If any issues or uncertainties arose, we addressed them promptly through discussion, ensuring everyone was on the same page.

2.2 Development Techniques:

We divided the tasks effectively among team members so that everyone had clear responsibilities and could contribute meaningfully. 240011448 focused on implementing colour support and the JSON parser; 240021654 handled the model and checker logic; 240021478 and 240016221 worked together on developing the graphical user interface, often using pair programming. During pair programming sessions, one member took the role of driver, writing code, while the other acted as navigator, reviewing the logic and offering suggestions. They frequently switched roles to maintain a balanced understanding across all components. Later, while working on the tertiary requirements, 240021654 and 240016221 collaborated through pair programming, using pen and paper to map out the logic. They employed techniques such as trace tables to follow up recursion, logical reasoning, and pseudocode to develop and complete both the deductive and guessing solvers. To top it off, 240011448 and 240021478 have implemented excellent JUnit tests, and manual tests to make sure our solution is near to perfection.

We made extensive use of Git for version control, which allowed us to track changes, collaborate asynchronously, and easily merge contributions. Commit messages were clear and descriptive, helping everyone understand the purpose of each update. If any team member was unsure about a particular commit, we took time to explain the implementation decisions, fostering a strong team learning environment. This structured yet flexible approach allowed us to complete all aspects of the project on time and to a high standard.

3. Code Design and Implementation

3.1 CellState.java:

This Enum defines the possible states of a Nonogram cell: UNKNOWN, EMPTY, and four colour states (COLOUR_1 to COLOUR_4). Using an Enum ensures type safety and simplifies state management, particularly for colour puzzles.

3.2 BlockConstraint.java:

This class represents a single constraint in the Nonogram, storing the length of a block and its associated colour state. It provides getter methods for accessing these properties, ensuring immutability and encapsulation.

3.3 GUI.java:

The GUI class manages the graphical interface for the Nonogram puzzle application. It is built using Java Swing and serves as the primary interaction point for the user. This class handles loading and saving puzzles, rendering the puzzle grid, managing user inputs (like colouring cells), and validating puzzle completion.

The interface layout consists of a menu bar, status label, a colour key panel, and a dynamic grid panel. Each cell in the grid is represented by a button whose colour reflects its current CellState. The GUI also responds to actions like file operations, undo requests, and solution validation.

3.3.i Startup and Initialization:

The constructor of the class configures the window properties, initializes menus, and sets up the initial layout. The menu bar provides options to load/save puzzles and undo moves, with associated keyboard shortcuts for faster access (Ctrl+O, Ctrl+S, Ctrl+Z).

The bottom panel provides direct buttons for "Check Solution", "Save Moves", "Load Moves", and "Undo" functions. A statusLabel is included at the top to provide feedback messages to the user (e.g., on which file has been loaded).

The GUI remains dormant until a puzzle is loaded. On loading, it dynamically generates the grid, initializes the colour key, and constructs row and column constraints using BlockConstraint labels.

3.3.ii Grid Rendering:

The main puzzle grid is represented as a 2D array of JButton objects. Each button corresponds to a puzzle cell and updates its background colour to match the cell's CellState. The colour for each state is pulled from the puzzle's colour map.

Grid rendering is done by initializeGridGUI(), which builds the full grid layout, including row and column constraint panels and the actual grid panel. This method is also responsible for setting event listeners that modify the cell state when buttons are clicked, using the currently selected colour.

3.3.iii Colour Key Panel

The colour key panel is generated by colourKeyPanel(). It creates a button for each CellState, with its colour representing the actual display colour from the puzzle's colour map. Clicking one of these sets the currentColour, which determines what colour is applied to cells during interaction.

Each cell click applies this selected colour by calling setCellState() in the underlying puzzle. Before any modification, the current state is saved to the puzzle's history to support undo functionality.

3.3.iv Undo and File I/O

Undoing moves is supported through the "Undo" menu and button, invoking the puzzle's undo() method and redrawing the grid accordingly. The undo function is disabled when no puzzle is loaded or when there are no moves to revert.

Puzzle files are loaded using a file chooser and parsed with PuzzleLoader. On successful load, the GUI is refreshed with the new puzzle state. Moves can be saved or loaded using JSON serialization handled internally by the Nonogram class (saveMoves() / loadMoves()).

Puzzle name consistency is checked when loading a move file to ensure that moves are only applied to the correct puzzle.

3.3.v Constraint Display

Constraint panels are built using rowConstraintsLabel() and columnConstraintsLabel(). These helper methods parse each row or column's BlockConstraint array and render it with correct formatting and color-coded numbers.

This design offers intuitive visual guidance to the player while preserving the data integrity and logic encapsulated in the underlying Nonogram object.

3.3.vi Puzzle Validation

The "Check Solution" button allows the user to verify their current solution. It invokes the puzzle's isSolved() method and updates the statusLabel with a green success message or a red failure message. This encourages players to self-verify progress without needing external validation tools.

3.3.vii Error Handling and Feedback

The GUI provides basic error reporting using JOptionPane popups for unexpected events, like failure to load or save files, puzzle name mismatches, or invalid cell operations.

Dynamic status updates are printed to the status label to provide real-time feedback for successful operations like loading or solving a puzzle.

3.4 Nonogram.java

The Nonogram class manages the logic and state for a nonogram puzzle. It stores grid data, row and column constraints, colour mappings, and move history for undo functionality. It is constructed with either a custom or default colour map.

The constructor validates that constraints are not null. It initializes the grid and the move history stack, ensuring a clean state at creation. The overloaded constructor sets default grayscale colours for a classic black-and-white nonogram.

The grid itself is stored as a 2D array of CellState objects. UNKNOWN cells are used initially, ensuring it is default and there are no preset information leaks. The method initialiseGrid() handles this initialization and is also used by resetGrid() to restart the puzzle and clear the move history.

3.4.i Undo Support

The undo mechanism is implemented using a stack (moveHistory) that stores deep copies of the grid using getGridCopy(). Before a change to any cell, saveState() is invoked. This ensures that undoing changes is reliable and maintains previous states. The reason for using a stack is that it follows the LIFO (last in first out) mechanism which allows the last saved state to be called as soon as the undo button is clicked.

3.4.ii Cell Manipulation

setCellState() changes the state of a cell only if it differs from the current state. This is an optimization to avoid saving redundant states to the history stack. It also validates coordinates using validateCoordinates(), which throws an exception for illegal grid positions. getCellState() is a safe getter for external components.

3.4.iii Constraint Access

Methods like getRowConstraints() and getColumnConstraints() allow external modules to read the puzzle structure. The method getName() provides the puzzle identifier.

3.4.iv Puzzle Solving Logic

The isSolved() method checks whether the grid satisfies both the row and column constraints. It uses isLineSolved() for each row and column. This helper method identifies contiguous blocks of the same colour and compares them to the expected block constraints. Colour consistency and block lengths must match precisely.

3.4.v JSON Serialization

saveMoves() serializes the current puzzle state to JSON. It stores the puzzle name, grid, and colour map (if present). The colour map is stored as hex strings and retrieved using standard Colour methods. The method uses a FileWriter and a pretty JSON format.

loadMoves() deserializes a JSON file to restore a previously saved puzzle state. It checks puzzle name consistency before overwriting the current state. The method supports loading both the grid and the colour map. It includes warnings for unrecognized CellState values, ensuring graceful degradation for future state changes or corrupted files.

3.5 PuzzleLoader.java:

The PuzzleLoader class is responsible for reading and parsing a nonogram puzzle from a JSON file and creating a corresponding Nonogram object. It supports both default black-and-white nonograms and those with custom colour mappings.

3.5.i Main Method

This is the entry point for puzzle loading. It takes a JSON file path, reads its contents, and parses it into a Nonogram object. The JSON is expected to define the puzzle name, row constraints, and column constraints. Optionally, it may also define custom cell states using a states object.

Step 1: Read the JSON file into a string.

Step 2: Extract the puzzle name, rows, and columns.

Step 3: Convert these arrays to 2D arrays of BlockConstraint objects using the helper method parseConstraints().

Step 4: Check if custom states are defined. If not, create a default Nonogram with grayscale colours. If custom states are present, use parseStates() to build a colour map and pass it to the constructor.

This modular approach ensures support for both traditional and color nonograms while maintaining backward compatibility with simpler puzzle formats.

3.5.ii Helper Method: parseStates

This method processes the states object in the JSON, which maps CellState enum names to hex colour codes

Iteration: For each entry in the states JSON object:

Convert the key to a CellState enum value using Enum.valueOf().

Decode the hex string using Color.decode() to get a Color object.

Add the pair to a TreeMap<CellState, Color>.

Error Handling: Invalid entries are caught via IllegalArgumentException, and a warning is printed to help with debugging malformed JSON files.

This allows user-defined visual styling of the puzzle grid based on specific colors assigned to cell states.

3.5.iii Helper Method: parseConstraints

This method transforms the rows and columns sections of the JSON into structured BlockConstraint[][] arrays.

Outer Loop: Iterates over rows or columns, each of which is a JSON array of constraints.

Inner Loop: For each block, it reads the count and optional color field.

If a color is present, it converts it to a CellState.

If absent, it defaults to CellState.COLOUR_1, ensuring classic black-and-white puzzles remain valid.

Conversion: Builds a list of BlockConstraint objects and then converts it into a fixed-size array.

This method is robust against varying lengths of constraints per row/column and handles both colour and monochrome puzzles.

3.6 Solver.java

The Solver class encapsulates the logic for automatically solving a Nonogram puzzle using constraint propagation and backtracking. It is a backend utility component that operates independently of the GUI and interacts directly with the Nonogram object to process and update its internal grid state. The class provides a deterministic solver based on logical inference, pattern merging, and recursive generation of valid line permutations that satisfy a row or column's constraints. It stops either when the puzzle is solved, or no further progress can be made.

3.6.i Startup and Invocation

The primary entry point is the solve() method, which orchestrates the full solving process. It runs through iterative passes over all rows and columns, trying to refine the grid based on current information. If no further logical deductions can be made, and the puzzle is not yet solved, the solver can optionally trigger a secondary guessing strategy (e.g., through Guesser.java). Each invocation of solve() operates on the currently loaded puzzle and expects a valid Nonogram object to be passed to the constructor.

3.6.ii Solving Logic and Looping

solve()

* Iterates over the puzzle up to a defined maximum number of iterations (MAX_ITERATIONS).

* On each iteration, processes each row and column to determine if new cell values can be deduced.

* Uses processRows(int) and processColumns(int) methods to handle line-specific logic.

* Exits early if the puzzle is successfully solved.

3.6.iii Line Processing and Deduction

processRows(int rowIndex) / processColumns(int columnIndex)

* Retrieves the row/column constraints and current line values.

* Calls generateLineFills() to find all valid fills (permutations) satisfying the constraints.

* Merges these fills using mergeLineFills() to identify consistent cells across all valid fills.

* If merged result reveals new information, updates the puzzle grid and signals progress.

3.6.iv Backtracking Fill Generation

generateLineFills(BlockConstraint[] constraints, CellState[] line)

* A recursive generator for all line permutations that match the given block constraints.

* Uses a backtracking strategy implemented in backtrackFill() to try placing blocks while respecting:

  * Color requirements.

  * Minimum spacing (especially for blocks of the same color).

  * Alignment with already filled cells (UNKNOWN, EMPTY, or colored).

backtrackFill(...)

* Base case: when all constraints are placed, the remaining cells are filled with EMPTY if valid.

* Recursive step: for each block, tries placing it at every legal position, appending the result if it doesn’t conflict with existing cell states.

3.6.v Merging and Conflict Resolution

mergeLineFills(List<CellState[]> validFills)

* Compares all valid fills cell-by-cell.

* For each position, if all fills have the same value, that value is considered deterministic and applied to the puzzle.

* If fills conflict, the cell remains UNKNOWN.

3.6.vi Utility Functions

* getRow(int), getColumn(int): Extract current puzzle state for a given row or column.

* setRow(int, CellState[]), setColumn(int, CellState[]): Update the grid after deducing new states.

* remainingLength(...): Calculates the minimum required length to place remaining blocks, accounting for spacing.

* matches(CellState actual, CellState expected): Checks compatibility between existing cell value and a proposed value.

3.6.vii Puzzle Completion and Fallback

* If the puzzle is successfully solved (nonogram.isSolved()), the solve() method prints a success message.

* If no progress can be made after all iterations and guessing is allowed, the solver can defer to a brute-force or guessing strategy via the Guesser.

3.6.viii Error Handling

* Gracefully handles puzzles with no valid fills by skipping updates on such rows/columns.

* Contains assertion-like checks to ensure valid puzzle dimensions and constraints during fill generation.

* Does not perform GUI interaction or show popups – backend only.

3.7 Guesser.java

The Guesser class provides a hybrid solving approach for Nonogram puzzles by combining deductive logic with recursive guessing. If pure logic (as handled by the Solver class) fails to produce a complete solution, Guesser will identify the most constrained line and explore all valid fill permutations recursively through backtracking.

3.7.i Main Method: solve()

This is the primary entry point for solving a Nonogram puzzle with fallback guessing.

* Step 1: Attempt a logical (deductive) solution using the Solver class.

* Step 2: If deduction fails, invoke guessAndCheck() to apply recursive guessing.

* Step 3: If any guess leads to a fully solved puzzle, return true.

* Fallback Handling: Enforces a maximum number of guesses (1000) to prevent infinite recursion or unreasonable computational cost.

This method allows robust solving of complex puzzles that cannot be resolved purely through logic.

3.7.ii Recursive Method: guessAndCheck()

This method implements a depth-first recursive backtracking algorithm for puzzle solving:

* Step 1: Exits if guess count exceeds maxGuesses, acting as a safety cap.

* Step 2: Identifies the most constrained line (with the fewest valid fills).

* Step 3: Generates all possible valid fills for that line using current constraints and cell states.

* Step 4: Tries each fill:

    * Saves the current grid state.

    * Applies the fill.

    * Attempts to solve the puzzle again via deduction and further guessing.

    * If unsuccessful, restores the saved state (backtracking).

* Step 5: Returns true if a solution is found, false otherwise.

This ensures minimal guessing and maximal logical propagation after each assumption.

3.7.iii Helper Method: findMostConstrainedLine()

Identifies the row or column that:

* Contains at least one UNKNOWN cell.

* Has the fewest valid fill permutations based on constraints and current state.

This strategy ensures the guessing algorithm branches at the most "informative" point, thereby improving efficiency.

3.8 Implementation

While implementing and designing the code we decided we decided we need follow Schneiderman's 8 Golden Rules of Interface Design to attain a good interface and achieve the best possible UX.

1. Striving for consistency: We made sure the frame size changes puzzle to puzzle but the size of the button stays the same which achieves consistency throughout the game.
2. Enabling frequent users to use shortcuts: We used shortcuts to make the users life easier by just adding shortcuts such as:
    a. Control + s to save
    b. Control + o to open
    c. Control + z to undo
    d. Control + r to reset
    e. Control + l to solve
3. Offer informative feedback: Our game gives informative feedback, it tells the user if the solution is correct or not, it provides feedback if there is no file uploaded and an action gets performed, it informs the user once the file has been successfully uploaded or if the file has failed to upload, it informs the user once the moves have been saved to file or if they have failed to save, it informs the user if there are no more undo's left.
4. Design dialogs to yield closure: there is a "Correct Solution!" "Success" message as soon as the puzzle is solved.
5. Permits easy reversal of actions: We have added the undo feature if the user messed up. Additionally, if the user uploads the wrong file, they can reupload the correct file and the process will take place smoothly.
6. Support internal locus of control: We give the user the access to feel in charge with clear the puzzle, select any puzzle they desire, and they can minimize and close the game whenever they want to take a break.
7. Offer simple error handling: We have added a checker which makes sure that no wrong file gets uploaded and only files with the '.json' extension get uploaded and the file get read to see if the correct file is uploaded like it checks the format etc.
8. Reduce short-term memory load: We added a border around the colour button selected to reduce short-term memory.

4. Testing

For our Testing we decided to manual tests as well as JUnit testing, both of which are documented in the table below.

Test	Expected	Pass/Fail	Evidence
load incorrectJson1.json	Exception is caught. Alert shows file could not be loaded.	Pass	Appendix: Figure 1
load incorrectJson2.json	Exception is caught. Alert shows file could not be loaded.	Pass	Appendix: Figure 2
load nocolumns.json	Exception is caught. Alert shows file could not be loaded.	Pass	Appendix: Figure 3
load norows.json	Exception is caught. Alert shows file could not be loaded.	Pass	Appendix: Figure 4
load noname.json	Exception is caught. Alert shows file could not be loaded.	Pass	Appendix: Figure 5
load noCount.json	Exception is caught. Alert shows file could not be loaded.	Pass	Appendix: Figure 6
help me button	PDF field containing information on nonograms and how to solve puzzle	Pass	Appendix: Figure 7
constraints on cat puzzle	Constraints on cat puzzle display correctly on the grid, all cell states should be unknown	Pass	Appendix: Figure 8
constraints on colour_umbrella	Constraints on colour umbrella puzzle display correctly on the grid, all cell states should be unknown	Pass	Appendix: Figure 9
save moves for house puzzle	Message box shows moves have been successfully stored in json file in the saved puzzles folder	Pass	Appendix: Figure 10 & 11
load moves for house puzzle	Message box shows moves have been loaded and the grid should be filled in correspondence with the json file	Pass	Appendix: Figure 12 & 13
undo button	Last modified cell should be updates to its previous state	Pass	Appendix: Figure 14 & 15
undo button if no moves have been made	Alert shows that there are no moves to undo	Pass	Appendix: Figure 16
reset button on puzzle freshly started	All modified cells should be updated to unknow	Pass	Appendix: Figure 17 & 18
reset button on moves which have been previously saved and loaded	All modified cells since the load should be updated to unknown and therefore puzzle should return to its saved state.	Pass	Appendix: Figure 19 & 20
reset button if no moves have been made	Alert shows that no moves have been made yet	Pass	Appendix: Figure 21

checker on a correct horse solution	Message box shows that it is a correct solution	Pass	Appendix: Figure 22
checker on an incorrect horse solution	Alert shows that it is an incorrect solution	Pass	Appendix: Figure 23
checker on a correct platypus solution	Message box shows that it is a correct solution	Pass	Appendix: Figure 24
checker on an incorrect platypus solution	Alert shows that it is an incorrect solution	Pass	Appendix: Figure 25
checker on the first correct solution to multiChecksPuzzle	Message box shows that it is a correct solution	Pass	Appendix: Figure 26
checker on the second correct solution to multiChecksPuzzle	Message box shows that it is a correct solution	Pass	Appendix Figure 27
solver on smiler puzzle	The cell states of the grid are updated to the correct solution and a message box shows that the puzzle has been solved	Pass	Appendix: Figure 28
solver on platypus puzzle	The cell states of the grid are updated to the correct solution and a message box shows that the puzzle has been solved	Pass	Appendix: Figure 29
solver on colour wink puzzle	The cell states of the grid are updated to the correct solution and a message box shows that the puzzle has been solved	Pass	Appendix: Figure 30
solver on colour cat puzzle	The cell states of the grid are updated to the correct solution and a message box shows that the puzzle has been solved	Pass	Appendix: Figure 31
load saved moves for house puzzle to horse puzzle	Alert shows that the saved moves do not match the current puzzle	Pass	Appendix: Figure 32
load huge.json	Puzzle loads but due to limited screen size, buttons are resized	Pass	Appendix: Figure 33
testInitialisedGrid()	JUnit pass	Pass	Appendix: Figure 34
testGetCellState()	JUnit pass	Pass	Appendix: Figure 34
testSetCellState()	JUnit pass	Pass	Appendix: Figure 34
testResetGrid()	JUnit pass	Pass	Appendix: Figure 34
testGetGridCopy()	JUnit pass	Pass	Appendix: Figure 34
testUndo()	JUnit pass	Pass	Appendix: Figure 34

testEmptyUndo()	JUnit pass	Pass	Appendix: Figure 34
testResetMoves()	JUnit pass	Pass	Appendix: Figure 34
testValidateCoordinatesNormal()	JUnit pass	Pass	Appendix: Figure 34
testValidateCoordinatesExceptional()	JUnit pass	Pass	Appendix: Figure 34
testIsLineSolvedNormal()	JUnit pass	Pass	Appendix: Figure 34
testIsLineSolvedNormal2()	JUnit pass	Pass	Appendix: Figure 34
testIsColumnSolvedNormal()	JUnit pass	Pass	Appendix: Figure 34
testIsColumnSolved2()	JUnit pass	Pass	Appendix: Figure 34
testIsLineSolvedExceptional()	JUnit pass	Pass	Appendix: Figure 34
testIsLineSolvedUnknown()	JUnit pass	Pass	Appendix: Figure 34
testIsSolvedNormal()	JUnit pass	Pass	Appendix: Figure 34
testIsSolvedExceptional	JUnit pass	Pass	Appendix: Figure 34
testSaveMoves()	JUnit pass	Pass	Appendix: Figure 34
testLoadMoves()	JUnit pass	Pass	Appendix: Figure 34
guess solver on player.json	The cell states of the grid are updated to the correct solution and a message box shows that the puzzle has been solved	Pass	Appendix: Figure 35
guess solver on multi_checks.json	The cell states of the grid are updated to the correct solution and a message box shows that the puzzle has been solved	Pass	Appendix: Figure 36
testBacktrackFillCase1()	JUnit pass	Pass	Appendix: Figure 37
testBacktrackFillCase2()	JUnit pass	Pass	Appendix: Figure 37
testBacktrackFillCase3()	JUnit pass	Pass	Appendix: Figure 37
testBacktrackFillCase4()	JUnit Pass	Pass	Appendix: Figure 37
testBacktrackFillCase5()	JUnit Pass	Pass	Appendix: Figure 37
testSolveSolvable()	JUnit Pass	Pass	Appendix: Figure 37

testSolveUnsolvable()	JUnit Pass	Pass	Appendix: Figure 37
testGuessUnsolvable()	JUnit Pass	Pass	Appendix: Figure 37

5. Evaluation

All primary, secondary, and tertiary requirements were successfully implemented in our Nonogram application. We developed an excellent GUI for loading, solving, and checking black-and-white as well as coloured puzzles. The addition of move recording, undo functionality, and a fully functional depth-first search solver with backtracking significantly expanded the functionality of our solution. Our solver can handle complex puzzles, including those with multiple solutions and those requiring guesses. Throughout the project, we enhanced our understanding of object-oriented design, GUI development with Java Swing, and recursive problem-solving techniques. Particular attention was given to maintaining clean separation between UI and core logic and improving testability. Developing graceful error handling for malformed puzzles and unsupported formats improved the program's resilience and user experience.

Version control using git helped streamline collaboration and maintain a clean development history. Regular team meetings and clear task distribution ensured equal participation and kept development on track. Future improvements could involve better performance optimization of the solve. Furthermore, we would like to spend more time adapting the GUI to be compatible with larger puzzles. As documented in the above testing, the huge.josn puzzle produces buttons of differing sizes when it is loaded. This puzzle can still be solved and checked like all other puzzles, however since we prioritised consistency throughout all other puzzles, the GUI suffers in this case, and this is something we would investigate and fix in the future. Overall, we are pleased with our outcome, as the application meets all specifications and reflects significant growth in our understanding of object-oriented preprograming concepts, java swing, permutations and recursions.

6. Conclusion

This project provided a rewarding opportunity to apply and deepen our programming knowledge by building a full-featured Nonogram puzzle application from scratch. Designing and integrating components like the puzzle loader, GUI, checker, and solver helped us understand the complexity of interactive applications.

One of the most enjoyable aspects was implementing the solver, which pushed us to explore and implement advanced algorithmic strategies like backtracking and constraint satisfaction. The challenges of supporting colour puzzles and undo functionality were met through careful planning and research which gave us the conclusion that a stack is the most efficient data structure because of its FIFO (first in first out) mechanism. Group collaboration, including pair programming and frequent communication, played a crucial role in our success.

Testing across a wide range of puzzles, including edge cases, strengthened our confidence in the robustness of our implementation. Given more time, improve performance optimization of the solve. This project has been a major step in our growth as critical thinkers, and we are excited to take forward the skills and insights gained here into future challenges.

7. References

[1] "Introduction to Java Swing." *GeeksforGeeks*, https://www.geeksforgeeks.org/introduction-to-java-swing/. Accessed 9 Apr. 2025.
[2] Lee, Alex. "Java GUI Tutorial - Make a GUI in 13 Minutes #99." *YouTube*, http://www.youtube.com/watch?v=5o3fMLPY7qY, 6 Feb. 2020. Accessed 9 Apr. 2025.
[3] "CS1006_P3_Nonogram.pdf." /cs/studres/2024_2025/CS1006/Lectures/. Accessed 9 Apr. 2025.
[4] Code, Bro. "Java menubar." *YouTube*, http://www.youtube.com/watch?v=7nEal9SJ6oI, 29 Aug. 2020. Accessed 9 Apr. 2025.
[5] "When using JFrame, where does the add method come from?" *Stack Overflow*, https://stackoverflow.com/questions/14860479/when-using-jframe-where-does-the-add-method-come-from. Accessed 9 Apr. 2025.
[6] "BorderLayout (Java Platform SE 7)." *Oracle*, https://docs.oracle.com/javase/7/docs/api/java/awt/BorderLayout.html. Accessed 22 Apr. 2025.
[7] "JLabel in Java Swing." *GeeksforGeeks*, https://www.geeksforgeeks.org/jlabel-java-swing/. Accessed 9 Apr. 2025.
[8] "Swing - JButton." *Tutorialspoint*, https://www.tutorialspoint.com/swing/swing_jbutton.htm. Accessed 19 Apr. 2025.
[9] "How to load a file using JFileChooser." *Stack Overflow*, https://stackoverflow.com/questions/27288636/how-to-load-a-file-using-jfilechooser. Accessed 21 Apr. 2025.
[10] "JOptionPane (Java Platform SE 7)." *Oracle*, https://docs.oracle.com/javase/7/docs/api/javax/swing/JOptionPane.html. Accessed 20 Apr. 2025.
[11] "Align text in JLabel to the right." *Stack Overflow*, https://stackoverflow.com/questions/12589494/align-text-in-jlabel-to-the-right. Accessed 15 Apr. 2025.
[12] "JTextPane add text bottom upwards." *JavaProgrammingForums.com*, https://www.javaprogrammingforums.com/awt-java-swing/33838-jtextpane-add-text-bottom-upwards.html. Accessed 18Apr. 2025.
[13] "Setting JTextPane Font and Color - Java Techniques." *Java Techniques*, https://javatechniques.com/blog/setting-jtextpane-font-and-color/. Accessed 15 Apr. 2025.

[14] "Is it possible to use color hex in JLabel like #02f7fc." *Stack Overflow*, https://stackoverflow.com/questions/39949076/is-it-possible-to-use-color-hex-in-jlabel-like-02f7fc. Accessed 11Apr. 2025.
[15] "How to make JTextArea non-selectable." *Stack Overflow*, https://stackoverflow.com/questions/12546056/how-to-make-jtextarea-non-selectable. Accessed 10Apr. 2025.
[16] "How to save file using JFileChooser in Java." *Stack Overflow*, https://stackoverflow.com/questions/14589386/how-to-save-file-using-jfilechooser-in-java. Accessed 19 Apr. 2025.
[17] "File Chooser (Using the JFileChooser)." *Oracle*, https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html. Accessed 24 Apr. 2025.
[18] "In GBC insets, what do the four numbers inside of the brakets mean in J." *Stack Overflow*, https://stackoverflow.com/questions/35304984/in-gbc-insets-what-do-the-four-numbers-inside-of-the-brakets-mean-in-j. Accessed 22 Apr. 2025.
[19] "GridBagLayout (Java Platform SE 8)." *Oracle*, https://docs.oracle.com/javase/8/docs/api/java/awt/GridBagLayout.html. Accessed 19 Apr. 2025.
[20] "Java AWT | GridBagLayout Class." *GeeksforGeeks*, https://www.geeksforgeeks.org/java-awt-gridbaglayout-class/. Accessed 20 Apr. 2025.

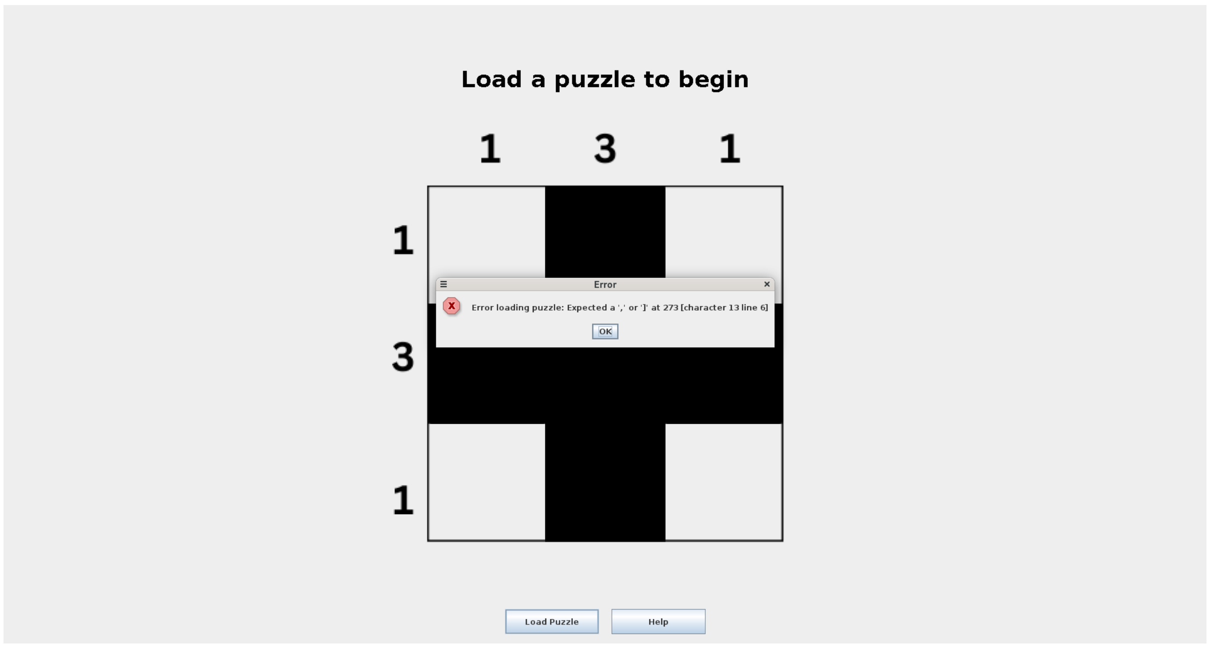
[21] "Puzzle 75579."
Nonograms.org, Web. https://www.nonograms.org/nonograms2/i/75579. Accessed 25 Apr. 2025

[22] Tutorialspoint. "Java System.arraycopy Method." *Tutorialspoint*
https://www.tutorialspoint.com/java/lang/system_arraycopy.htm. Accessed 25 Apr. 2025

[23] Debug assistance [TEH6]

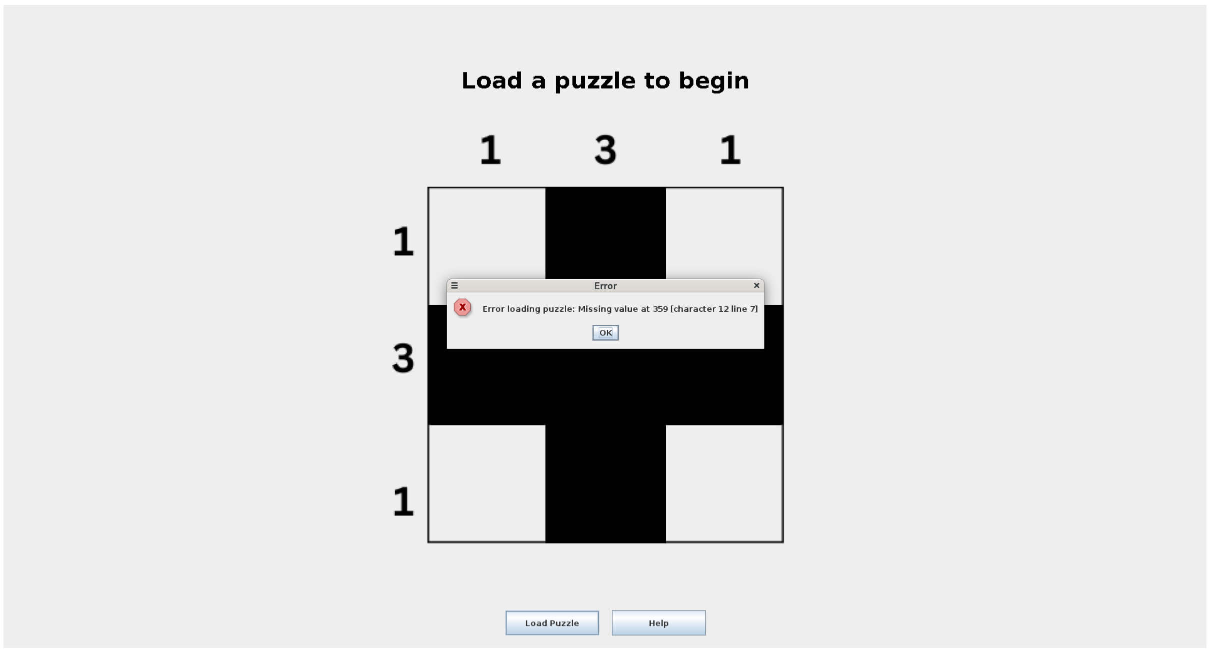
8. Appendix

Figure 1:



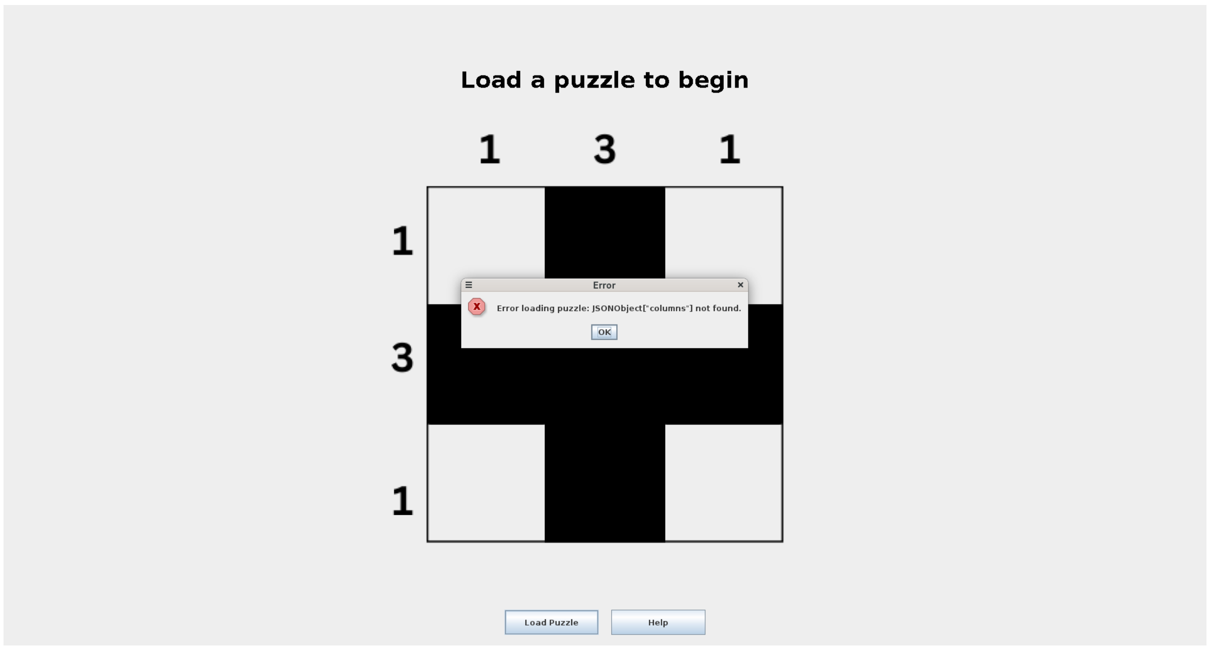
Error loading puzzle: Expected a ',' or ']' at 273 [character 13 line 6]

Figure 2:



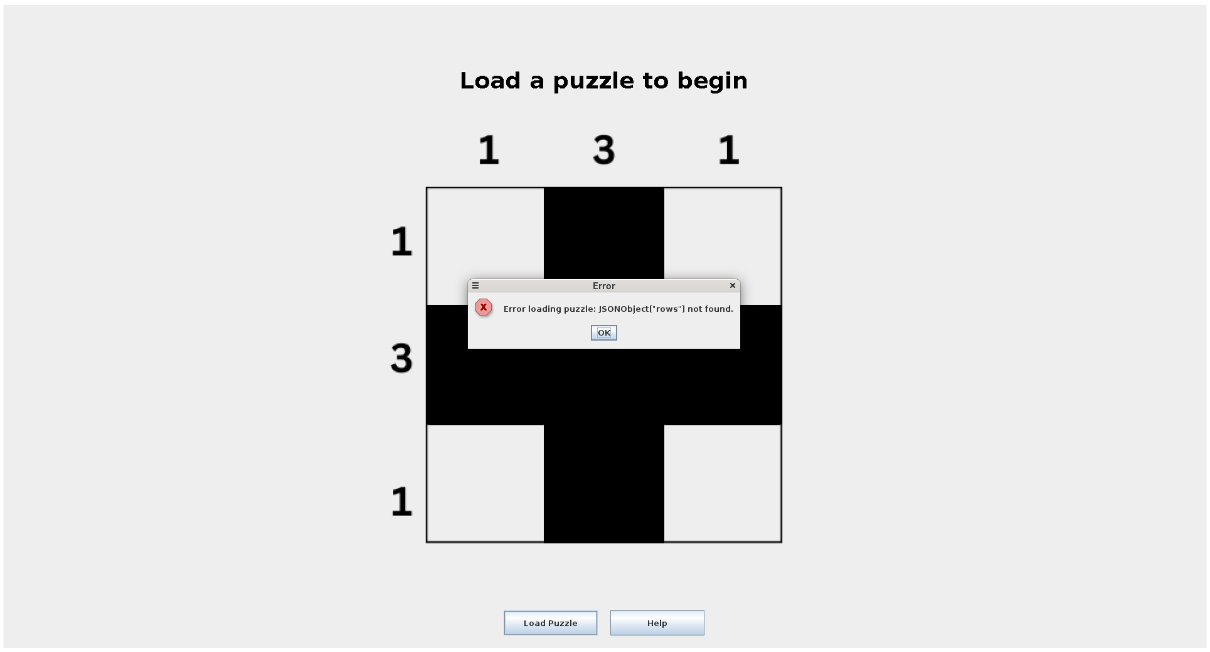
Error loading puzzle: Missing value at 359 [character 12 line 7]

Figure 3:



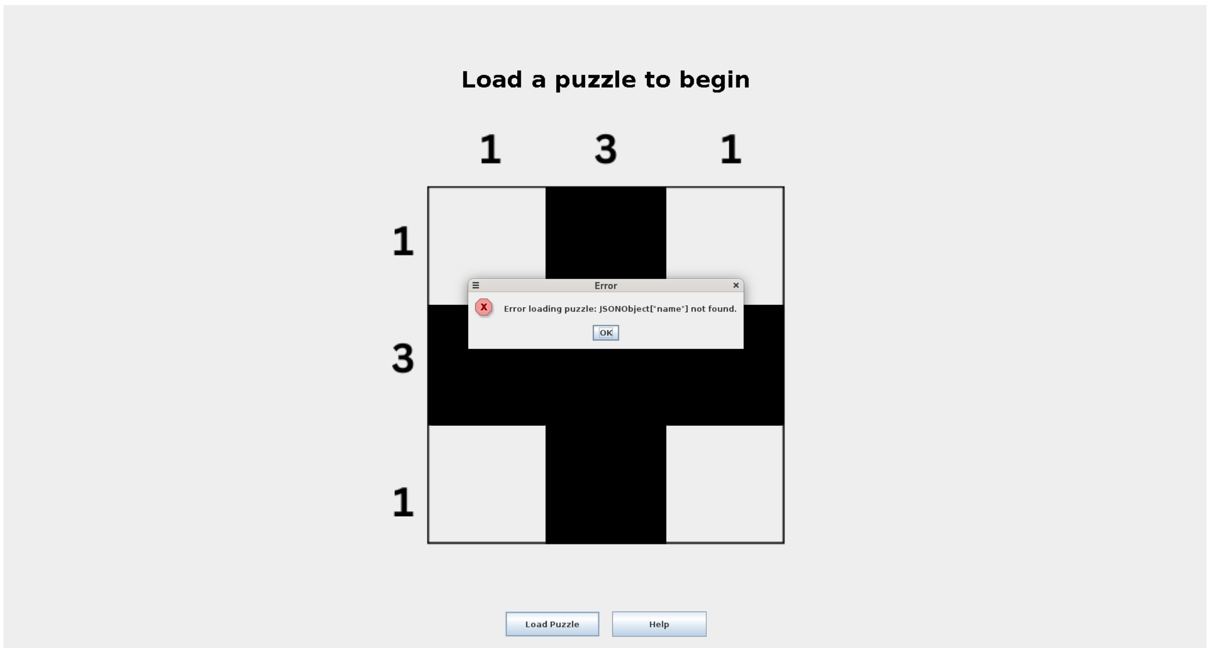
Error loading puzzle: JSONObject["columns"] not found.

Figure 4:



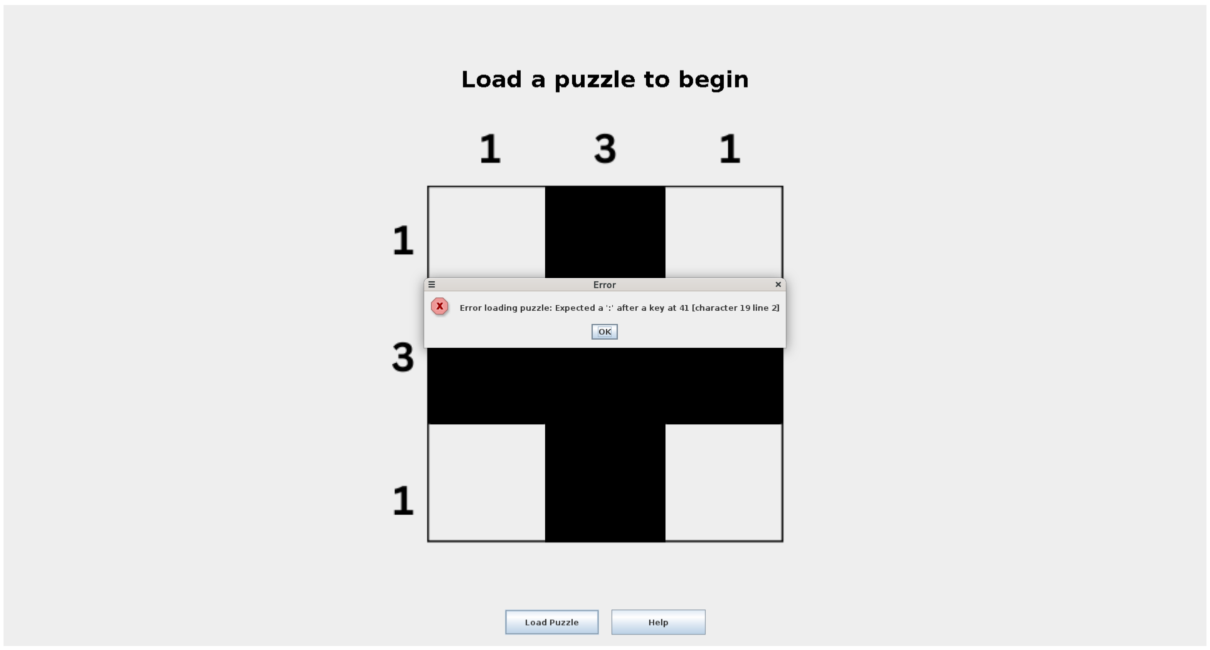
Error loading puzzle: JSONObject["rows"] not found.

Figure 5:



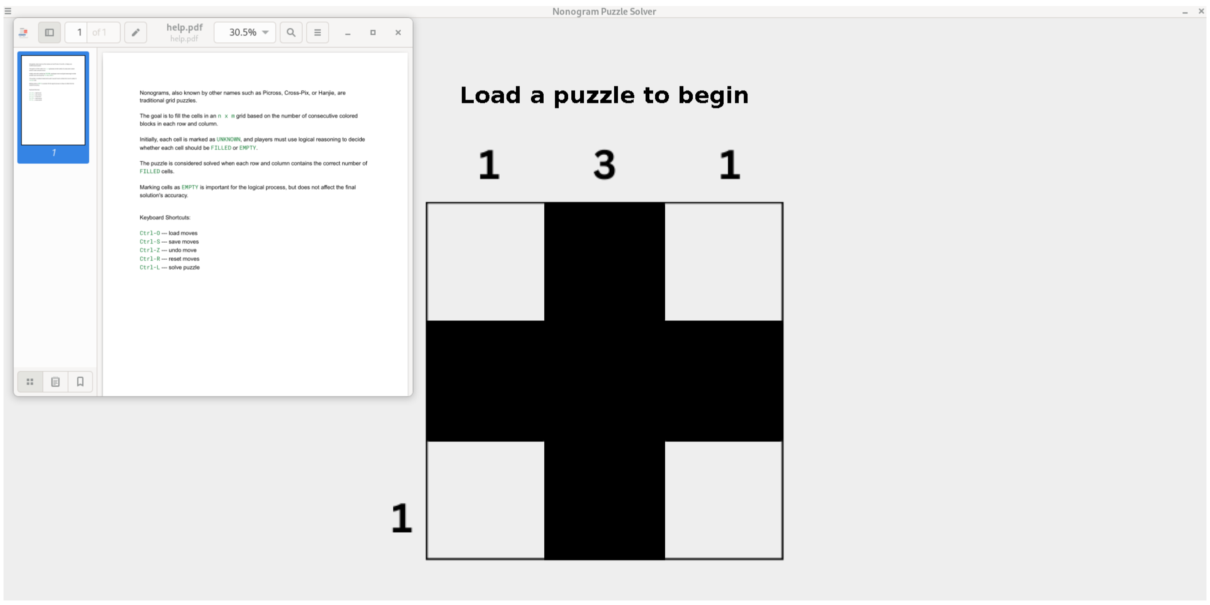
```
Error loading puzzle: JSONObject["name"] not found.
```

Figure 6:



```
Error loading puzzle: Expected a ':' after a key at 41 [character 19 line 2]
```

Figure 7:



Nonograms, also known by other names such as Picross, Cross-Pix, or Hanjie, are traditional grid puzzles.

The goal is to fill the cells in an `n x m` grid based on the number of consecutive colored blocks in each row and column.

Initially, each cell is marked as `UNKNOWN`, and players must use logical reasoning to decide whether each cell should be `FILLED` or `EMPTY`.

The puzzle is considered solved when each row and column contains the correct number of `FILLED` cells.

Marking cells as `EMPTY` is important for the logical process, but does not affect the final solution's accuracy.

Keyboard Shortcuts:

`Ctrl-O` — load moves
`Ctrl-S` — save moves
`Ctrl-Z` — undo move
`Ctrl-R` — reset moves
`Ctrl-L` — solve puzzle

Figure 8:

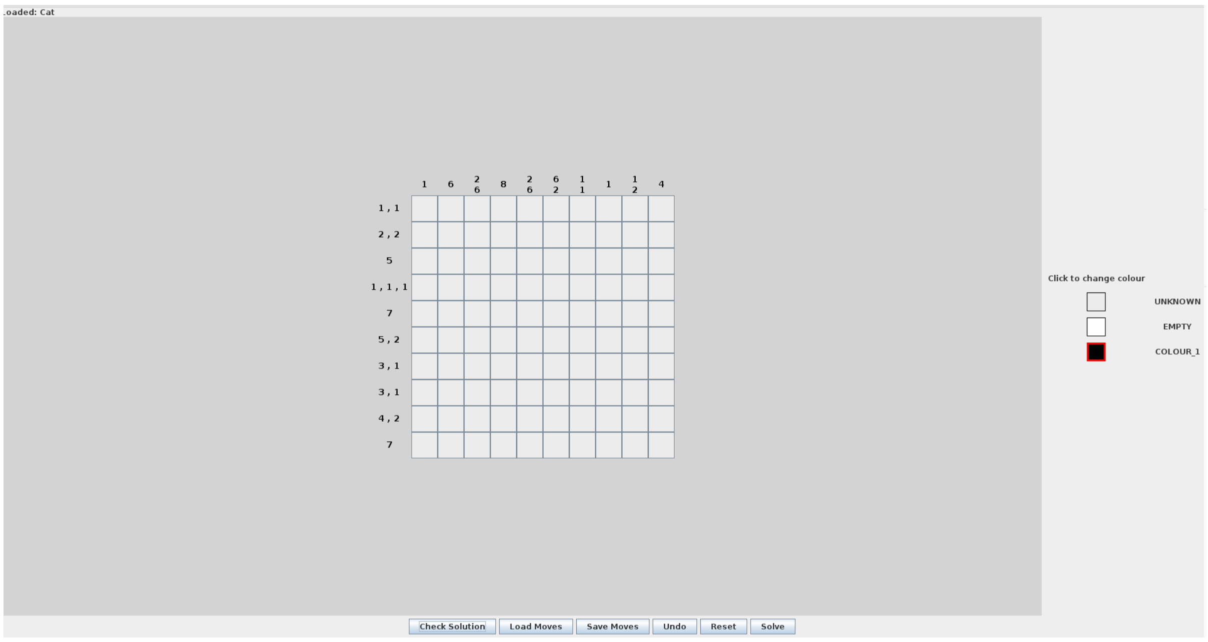


Figure 9:

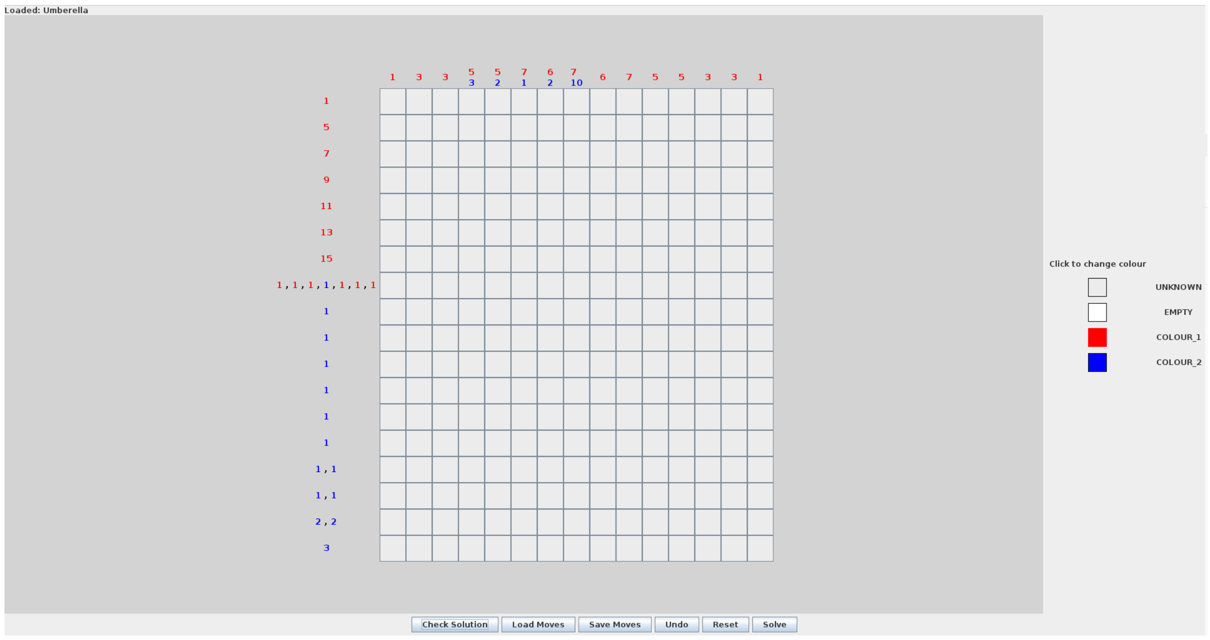
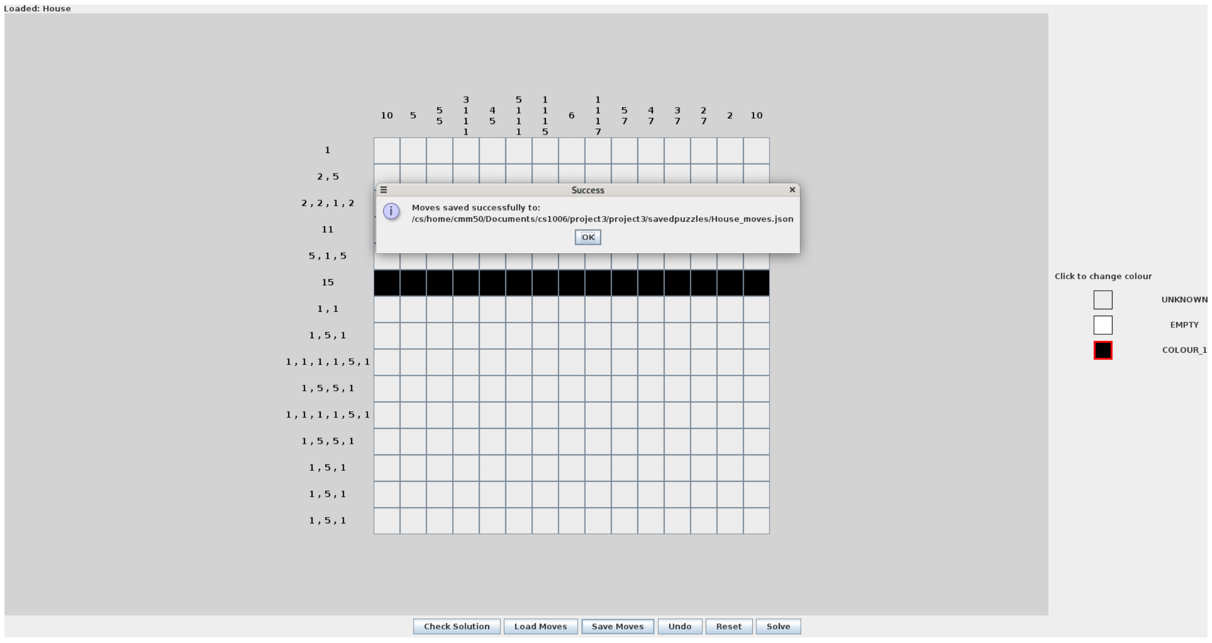
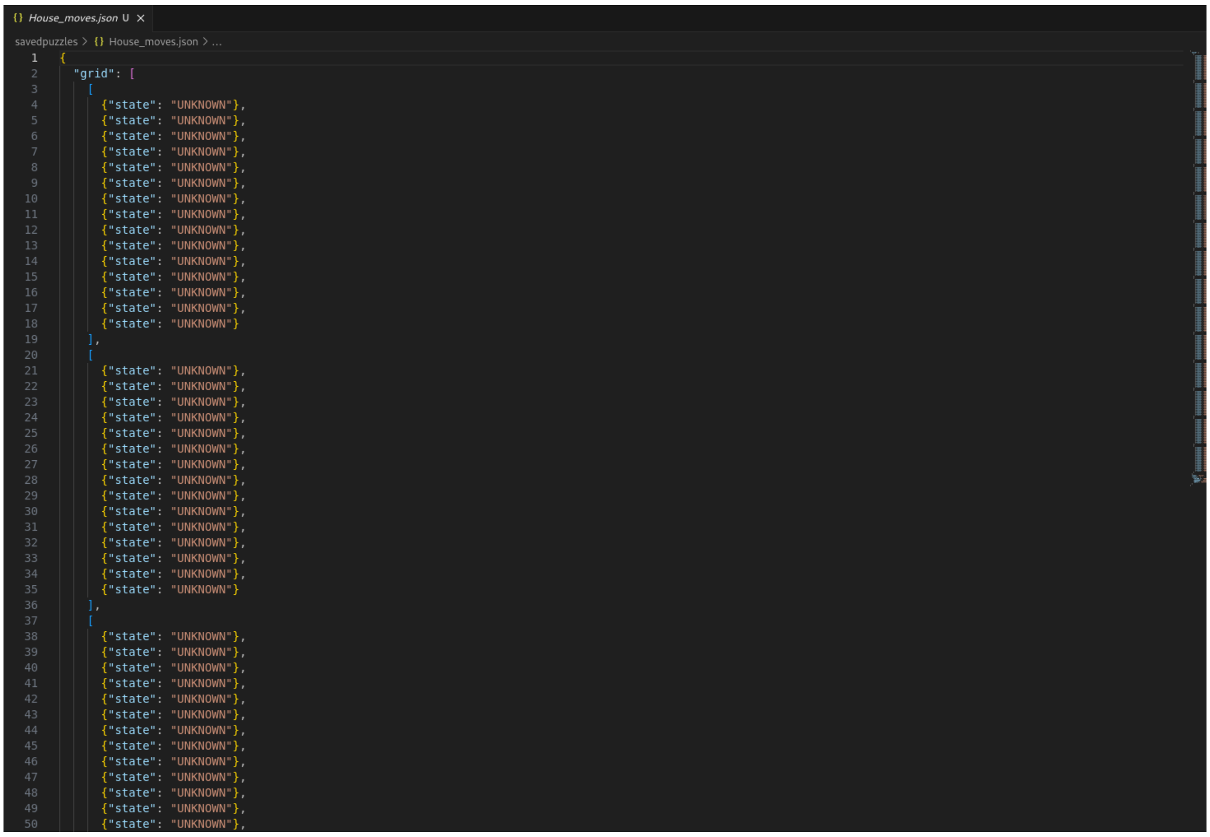


Figure 10:



Moves saved successfully to:
/cs/home/cmm50/Documents/cs1006/project3/project3/savedpuzzles/House_moves.json

Figure 11:



```json
{
  "grid": [
    [
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"}
    ],
    [
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"}
    ],
    [
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
      {"state": "UNKNOWN"},
```

Figure 12:

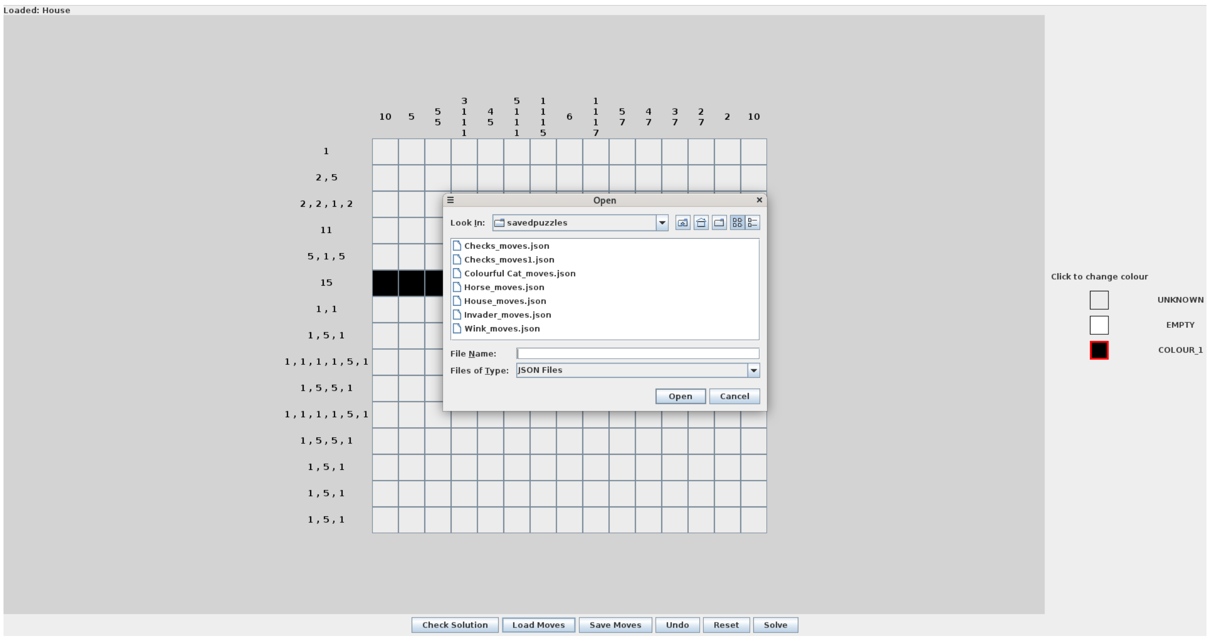
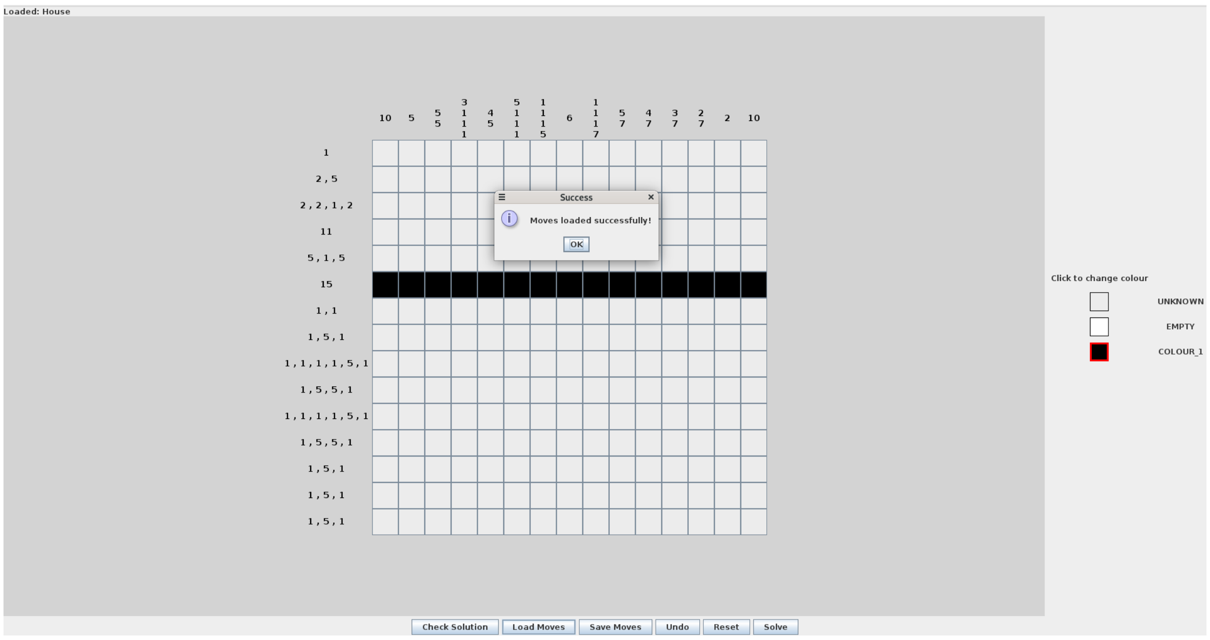


Figure 13:


Loaded: House

Success

Moves loaded successfully!

Figure 14:

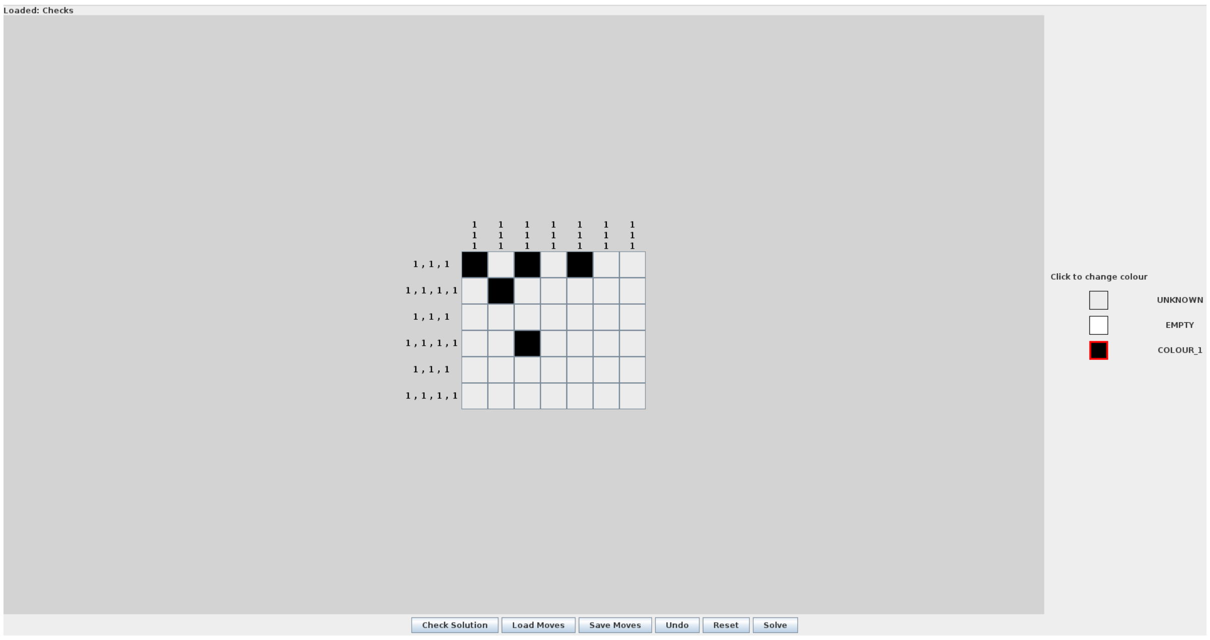

Loaded: Checks

Figure 15:

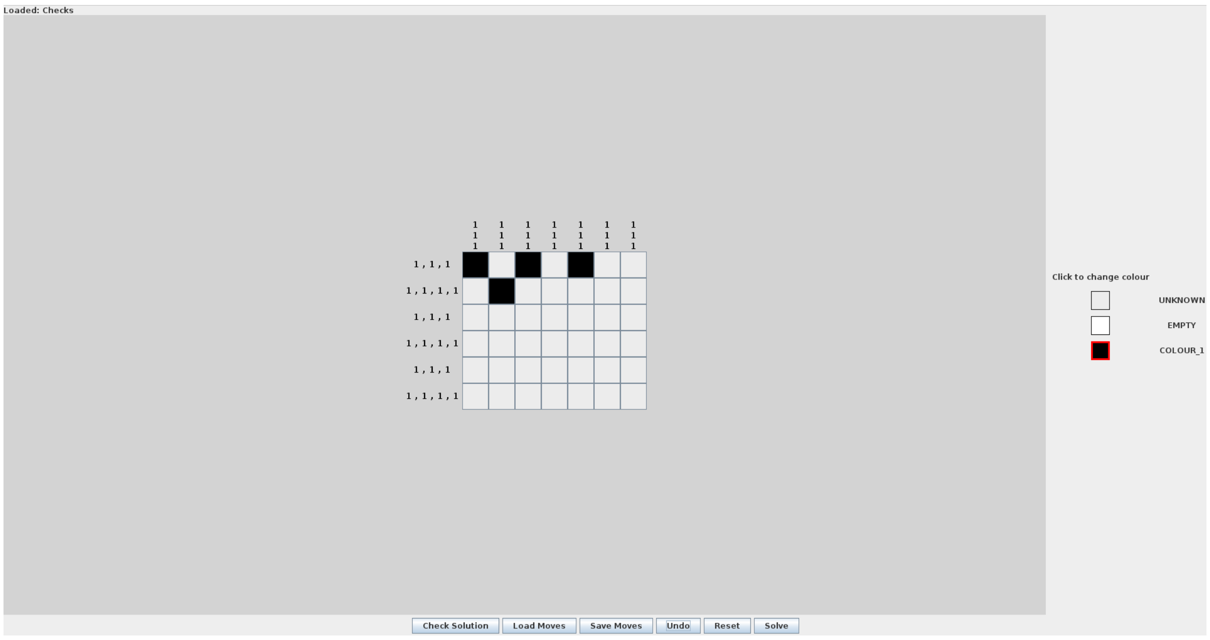
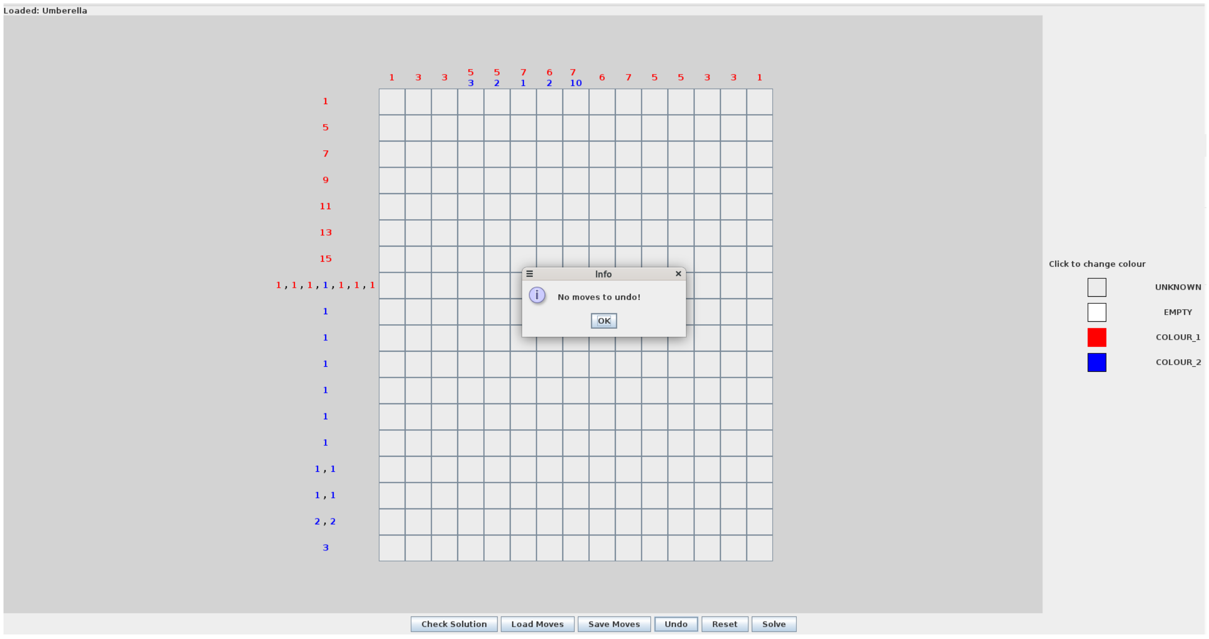

Loaded: Checks

Figure 16:


Loaded: Umberella

Info

No moves to undo!

Figure 17:

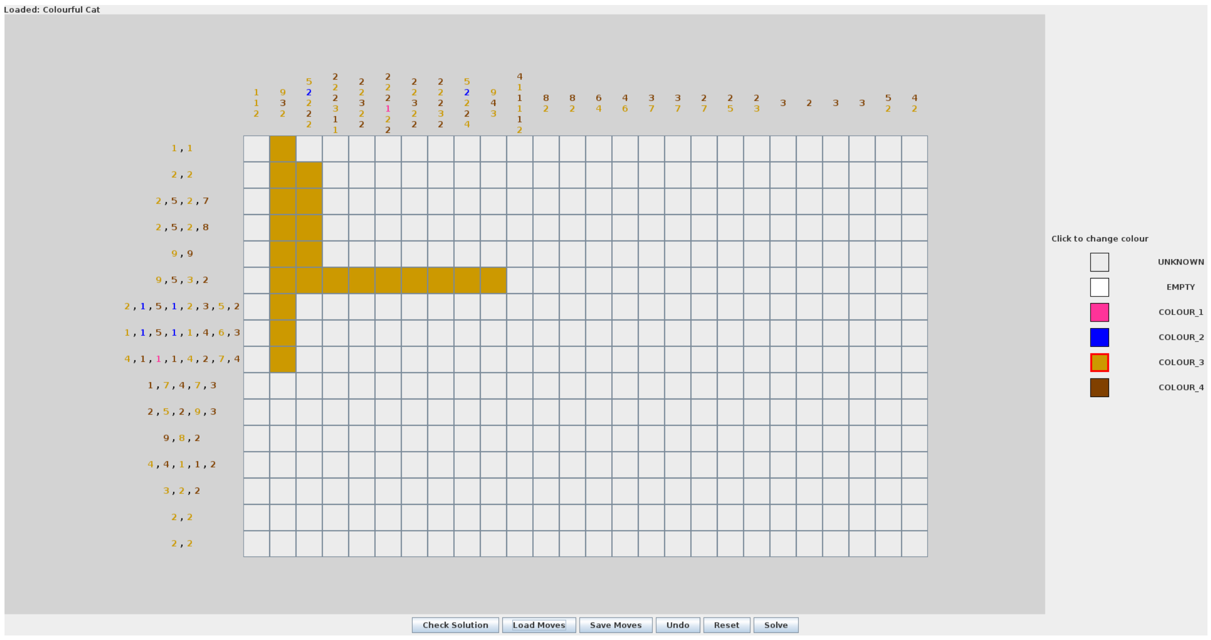


Figure 18:

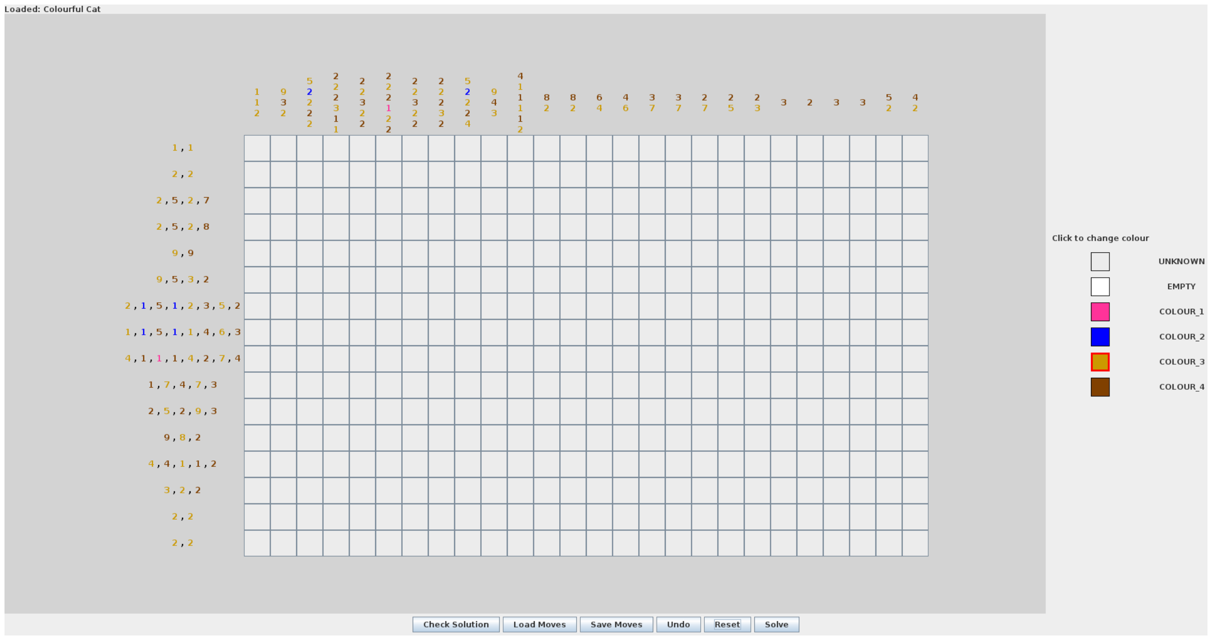


Figure 19:

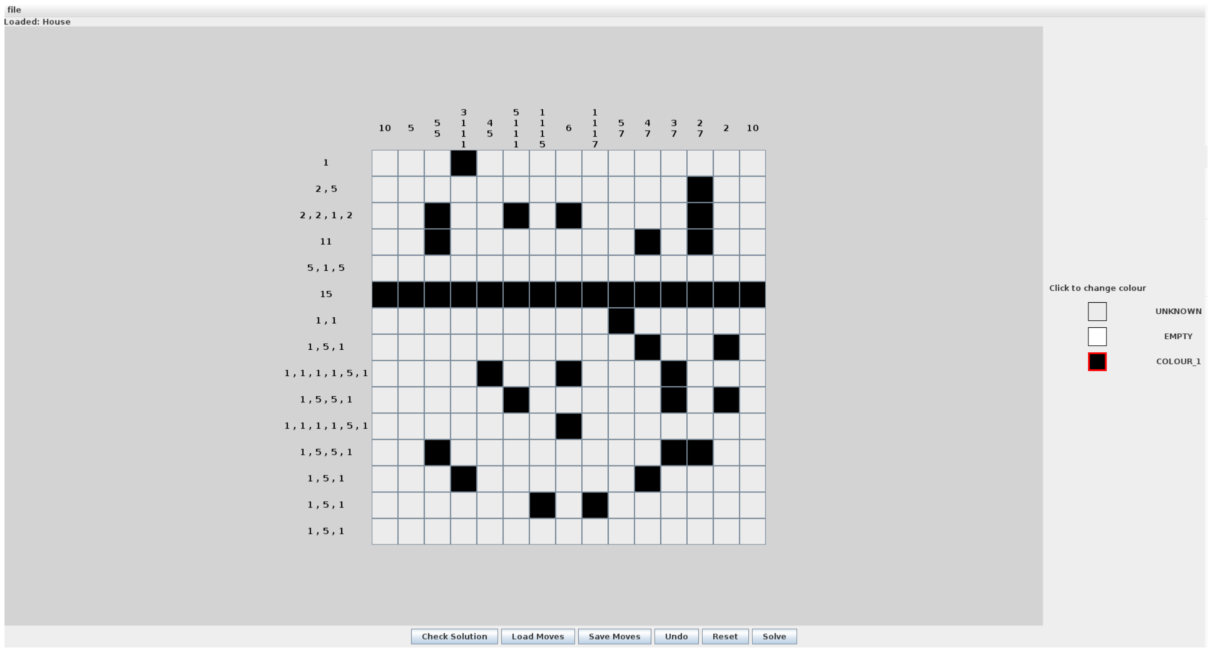


Figure 20:

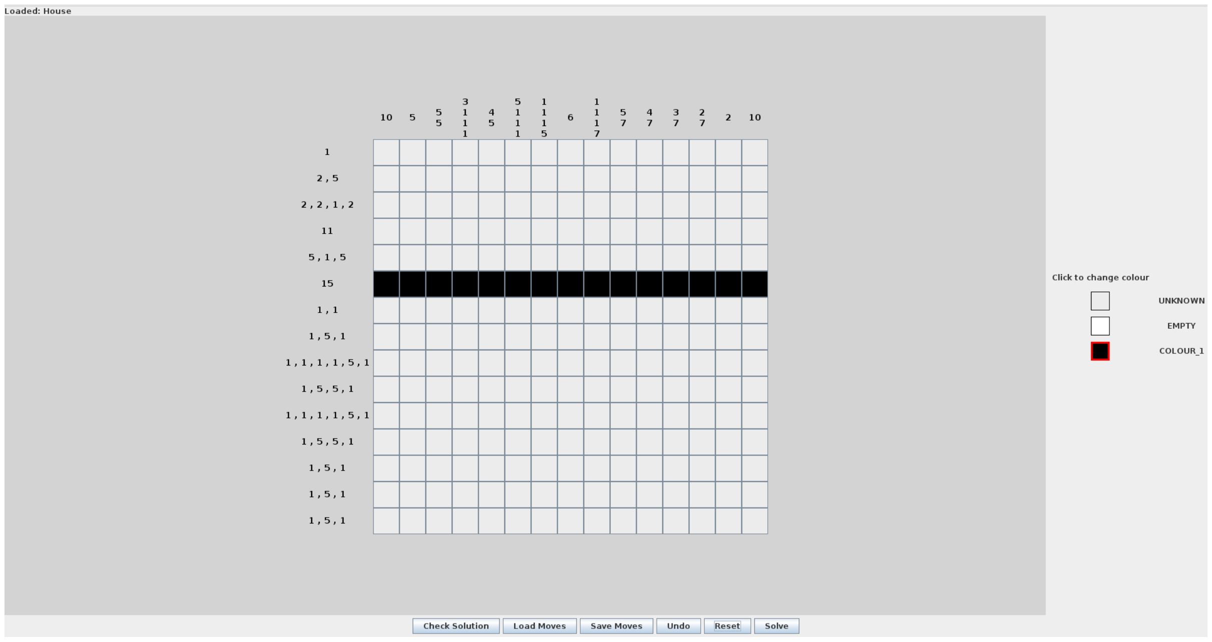
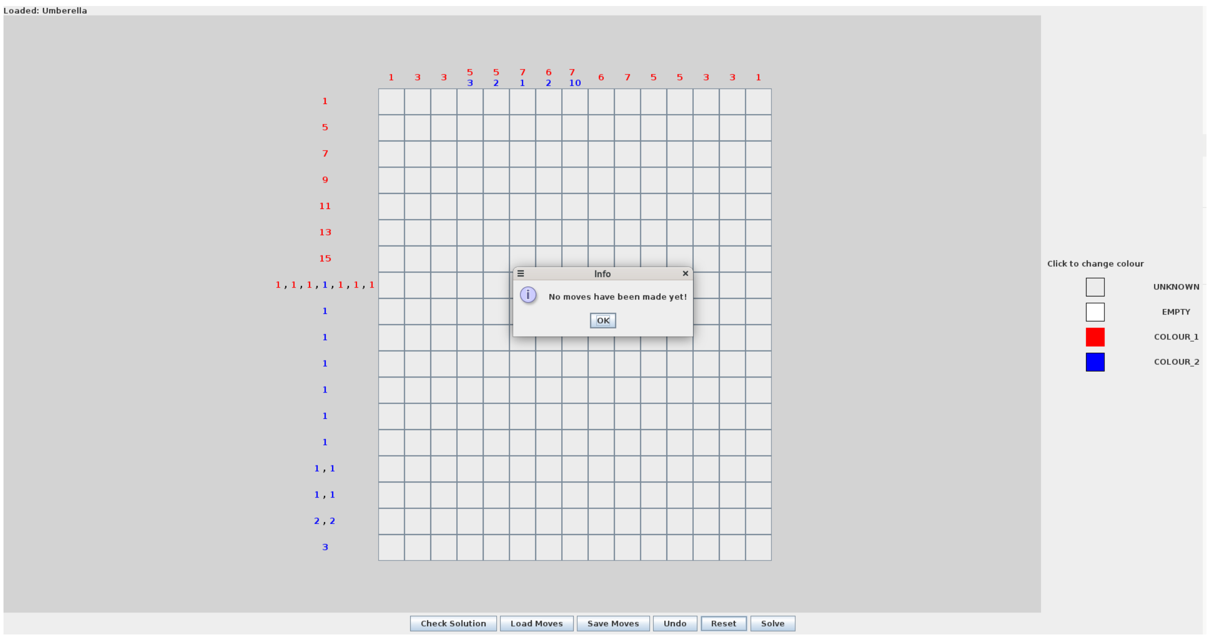


Figure 21:

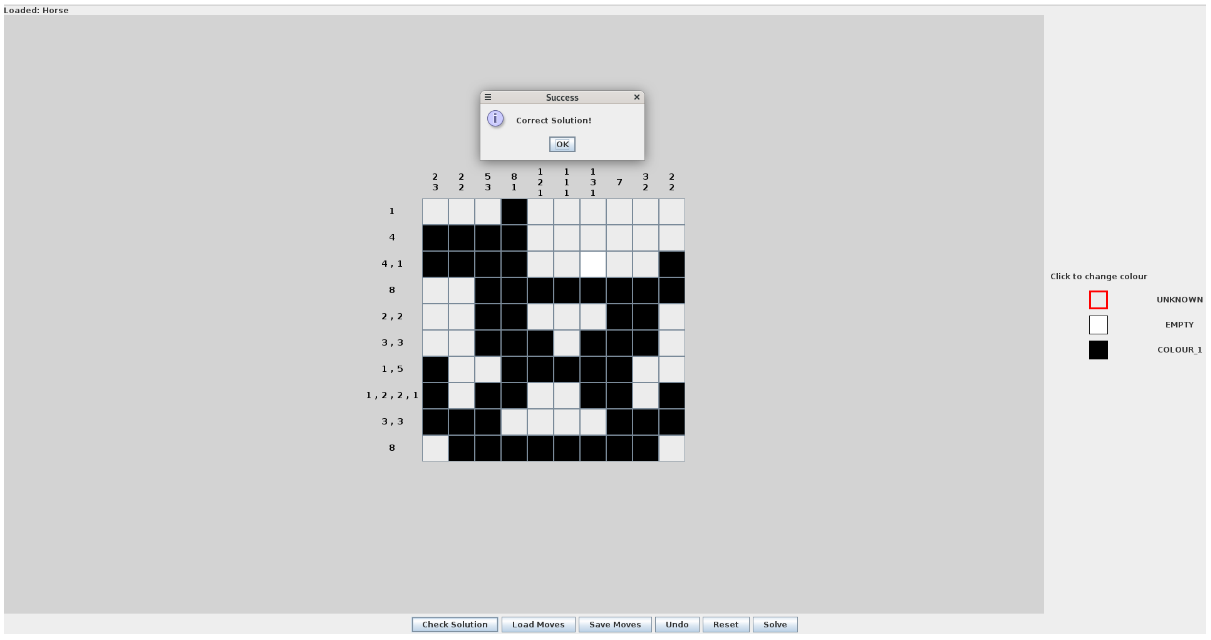


Info

No moves have been made yet!

OK

Figure 22:

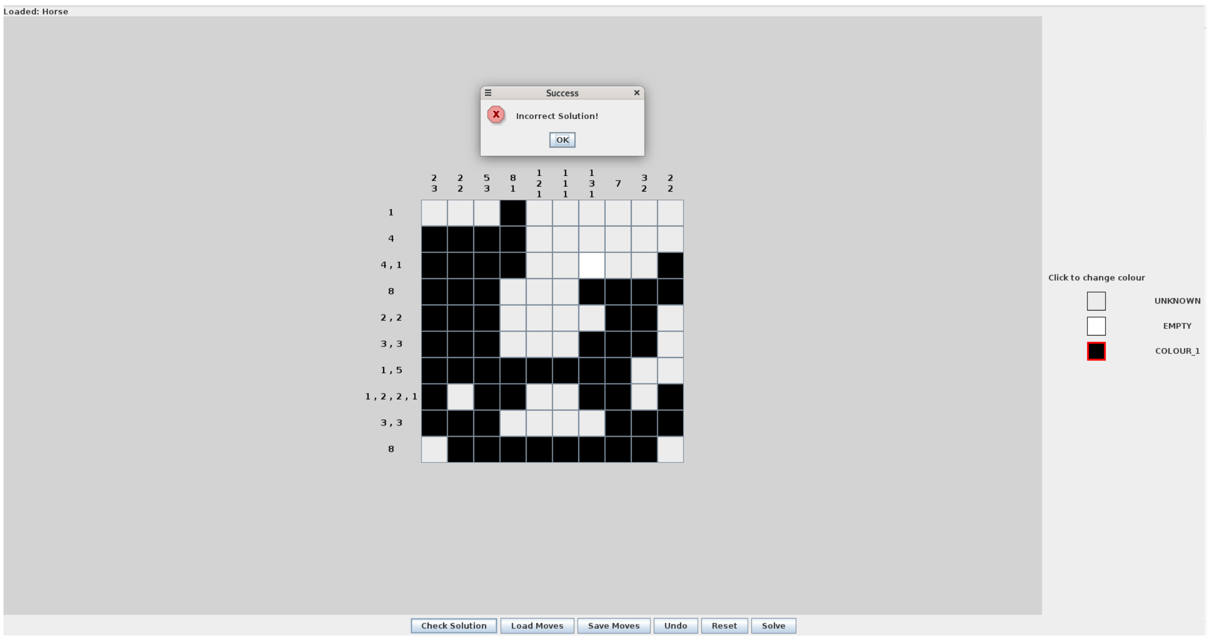


Success

Correct Solution!

OK

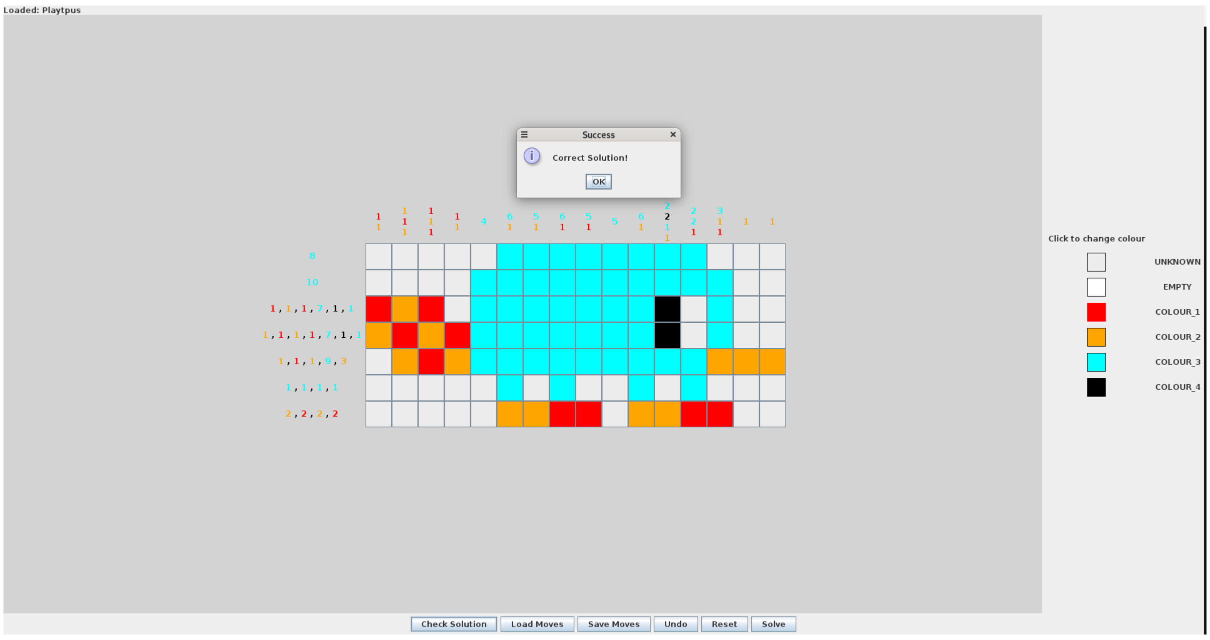
Figure 23:


Loaded: Horse

Success

Incorrect Solution!

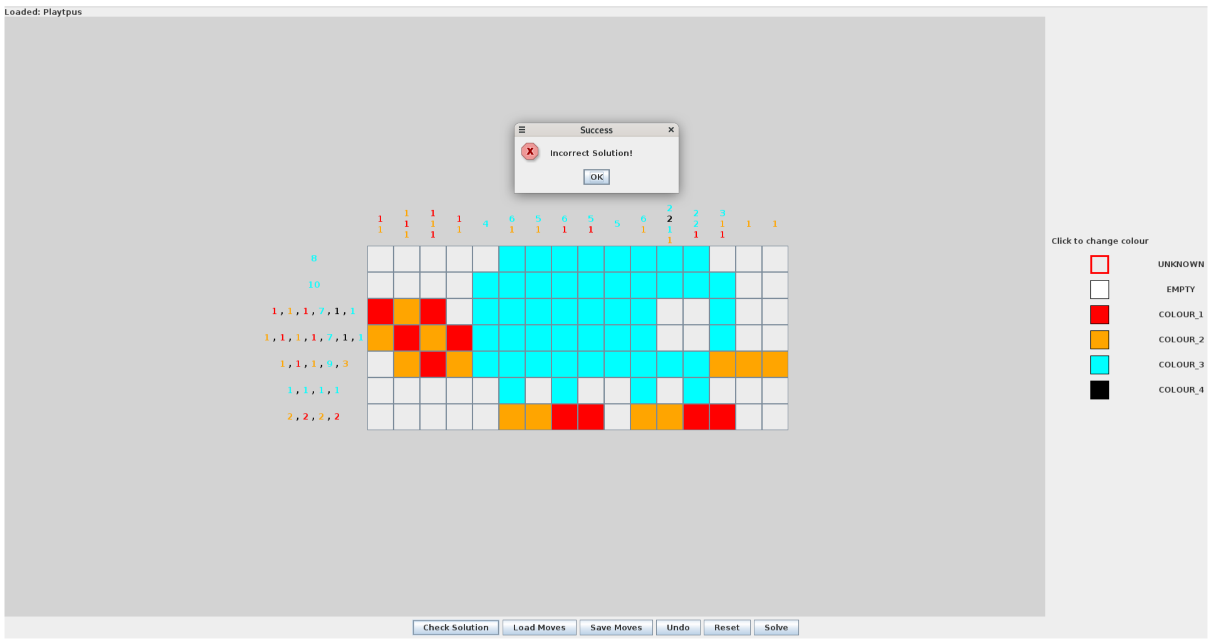
Figure 24:


Loaded: Playtpus

Success

Correct Solution!

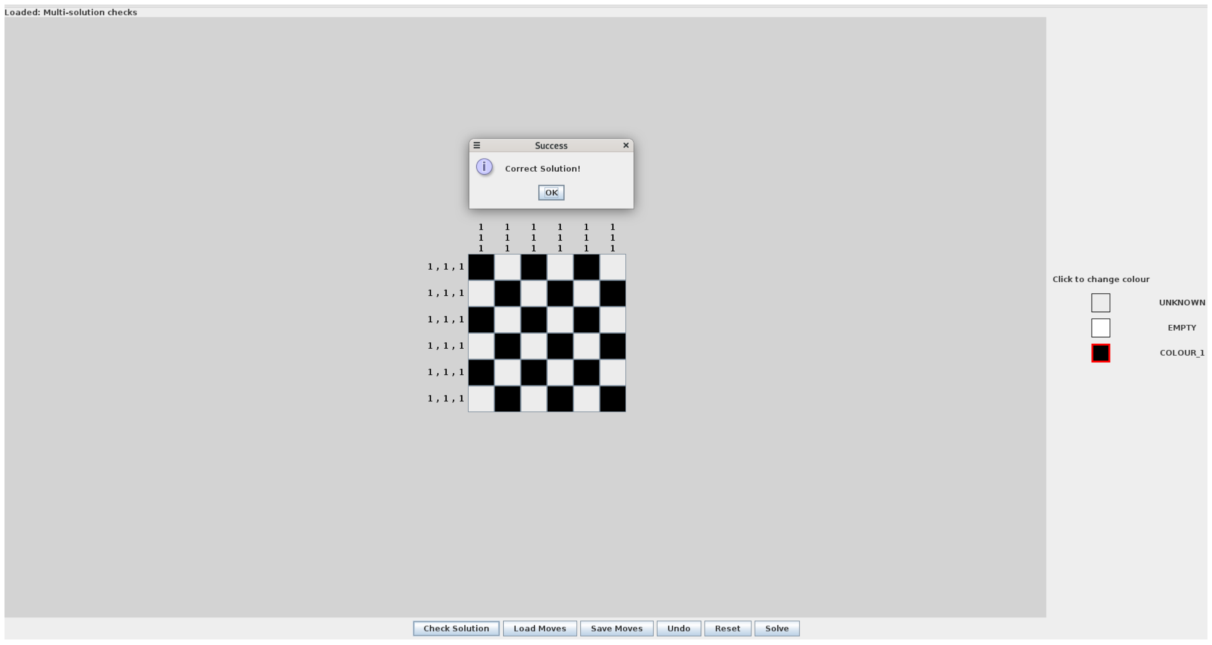
Figure 25:


Loaded: Playtpus

Success

Incorrect Solution!

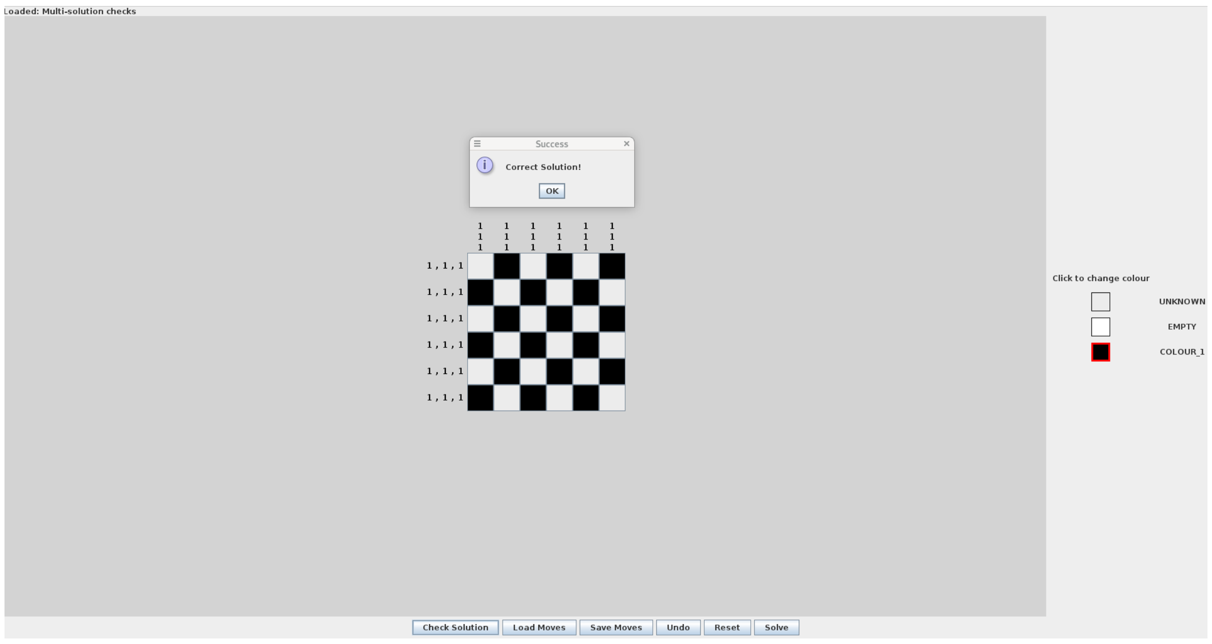
Figure 26:


Loaded: Multi-solution checks

Success

Correct Solution!

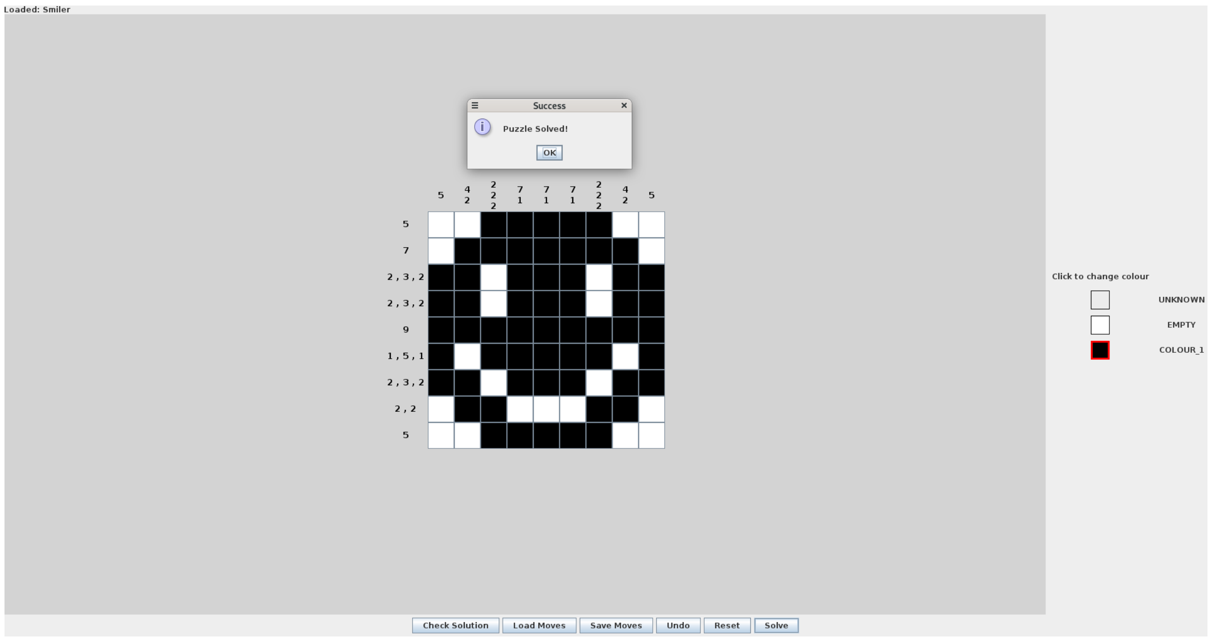
Figure 27:


Loaded: Multi-solution checks

Success

Correct Solution!

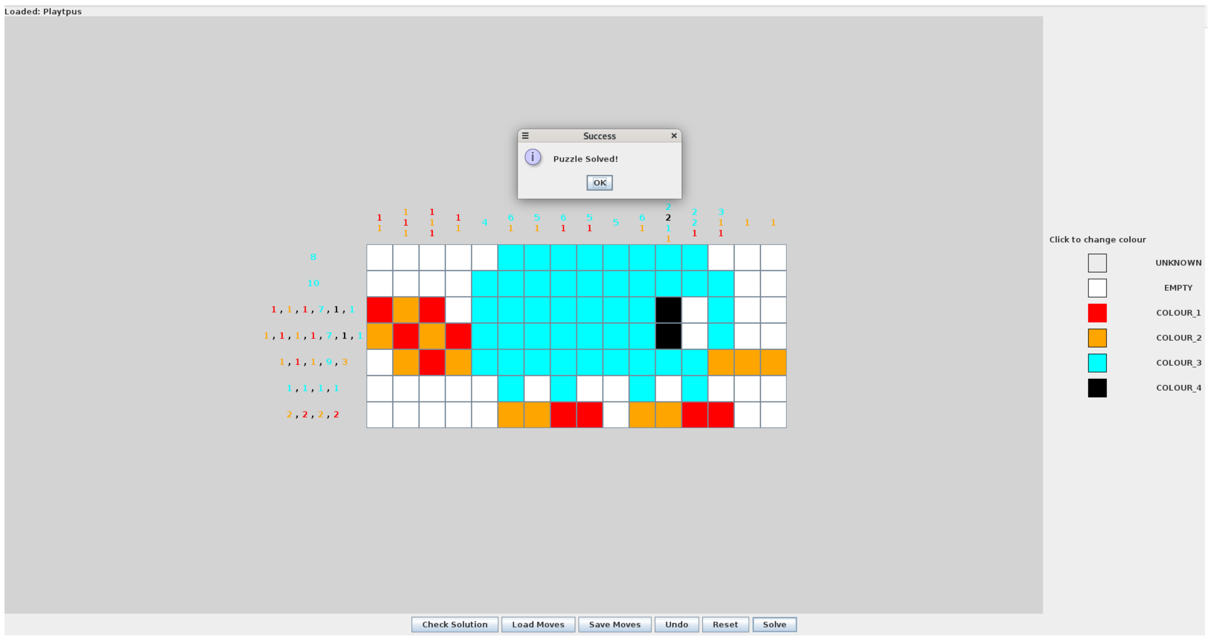
Figure 28:


Loaded: Smiler

Success

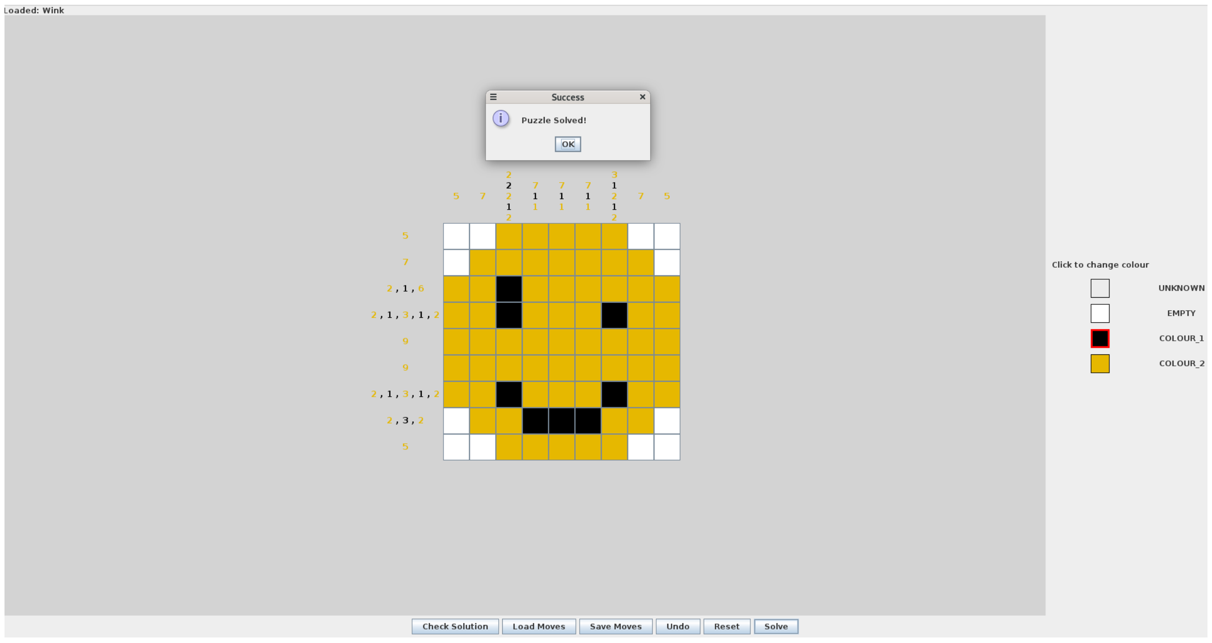
Puzzle Solved!

Figure 29:



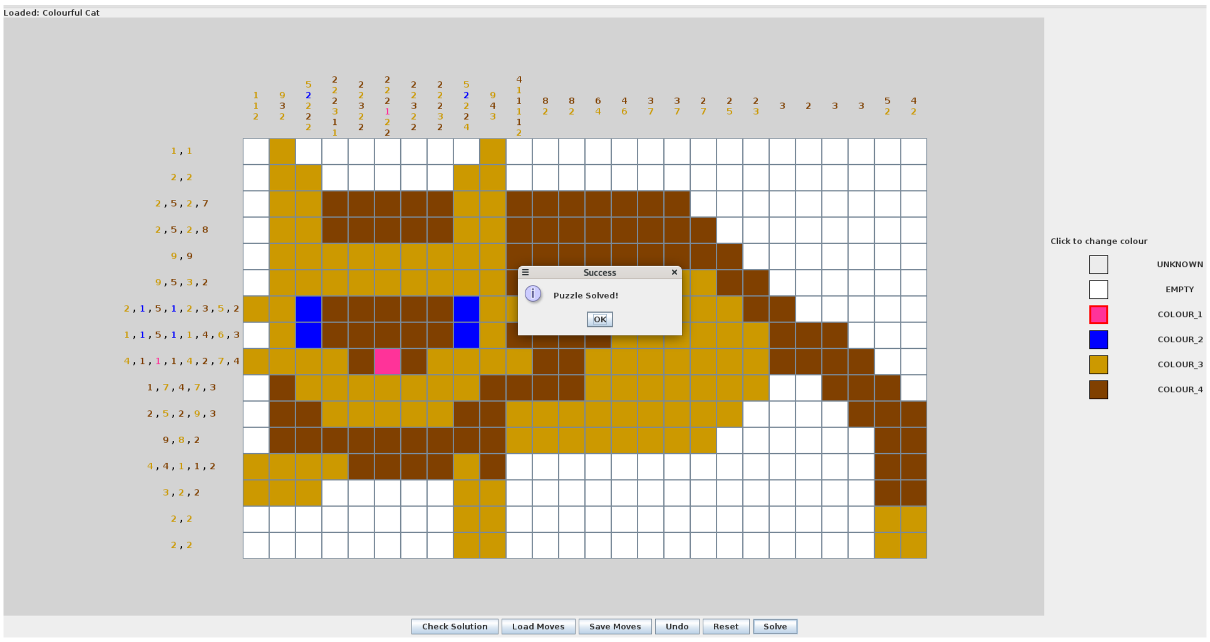
Puzzle Solved!

Figure 30:



Puzzle Solved!

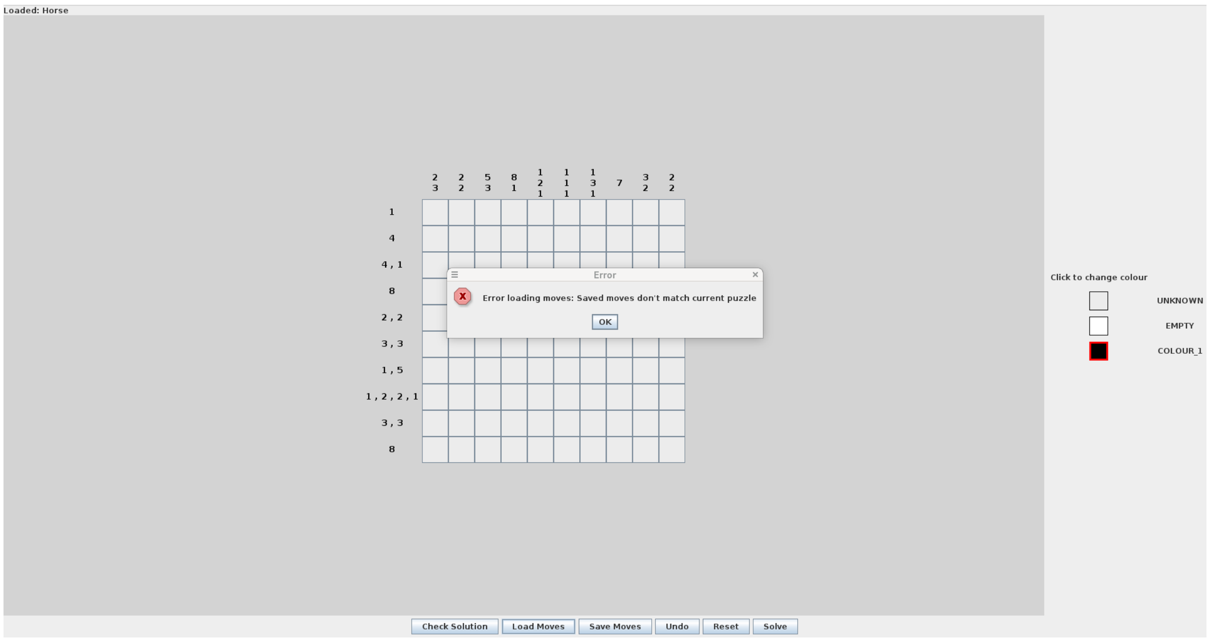
Figure 31:


Loaded: Colourful Cat

Success

Puzzle Solved!

Figure 32:


Loaded: Horse

Error

Error loading moves: Saved moves don't match current puzzle

Figure 33:

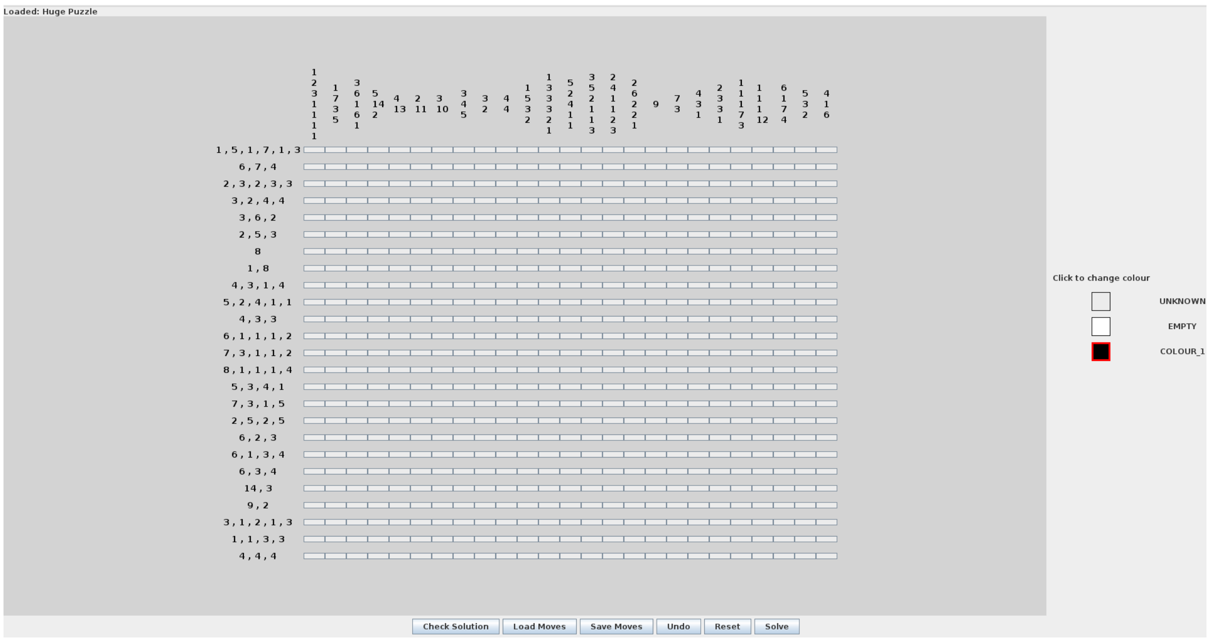


Figure 34:

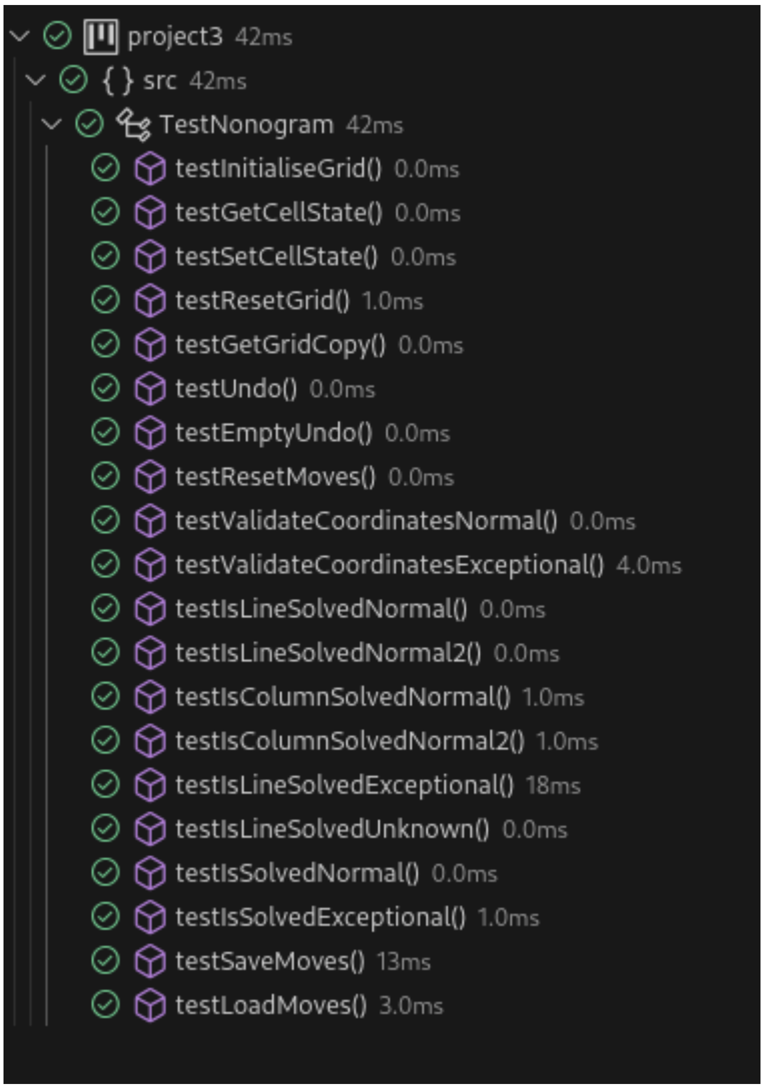
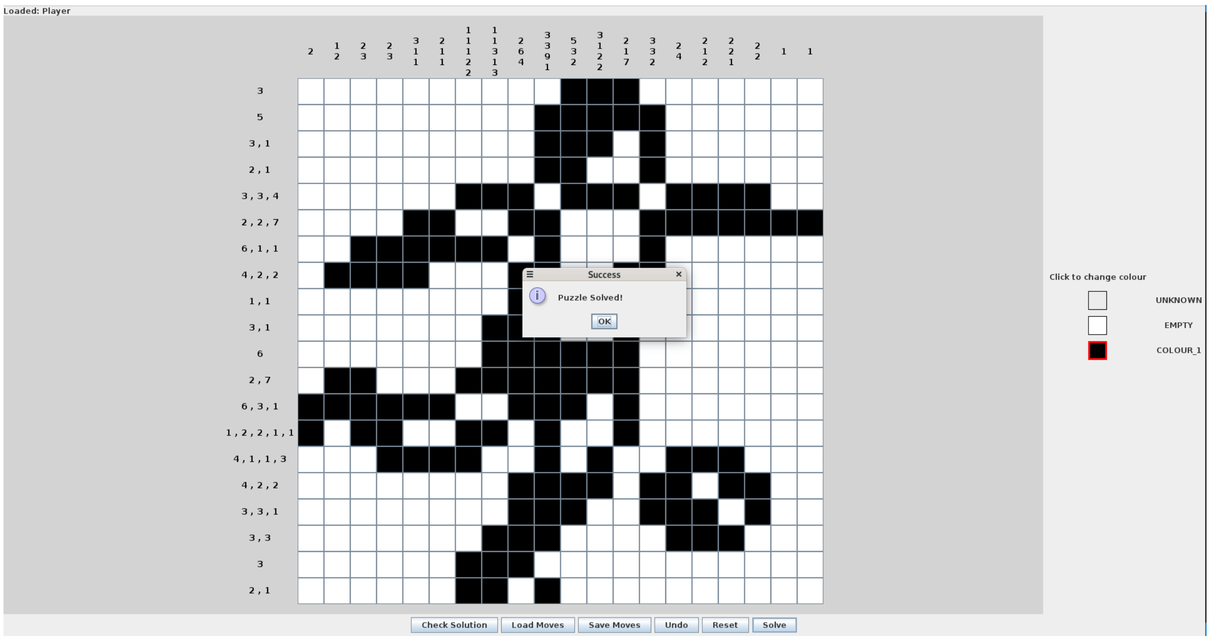

```
project3 42ms
  src 42ms
    TestNonogram 42ms
      testInitialiseGrid() 0.0ms
      testGetCellState() 0.0ms
      testSetCellState() 0.0ms
      testResetGrid() 1.0ms
      testGetGridCopy() 0.0ms
      testUndo() 0.0ms
      testEmptyUndo() 0.0ms
      testResetMoves() 0.0ms
      testValidateCoordinatesNormal() 0.0ms
      testValidateCoordinatesExceptional() 4.0ms
      testIsLineSolvedNormal() 0.0ms
      testIsLineSolvedNormal2() 0.0ms
      testIsColumnSolvedNormal() 1.0ms
      testIsColumnSolvedNormal2() 1.0ms
      testIsLineSolvedExceptional() 18ms
      testIsLineSolvedUnknown() 0.0ms
      testIsSolvedNormal() 0.0ms
      testIsSolvedExceptional() 1.0ms
      testSaveMoves() 13ms
      testLoadMoves() 3.0ms
```

Figure 35:

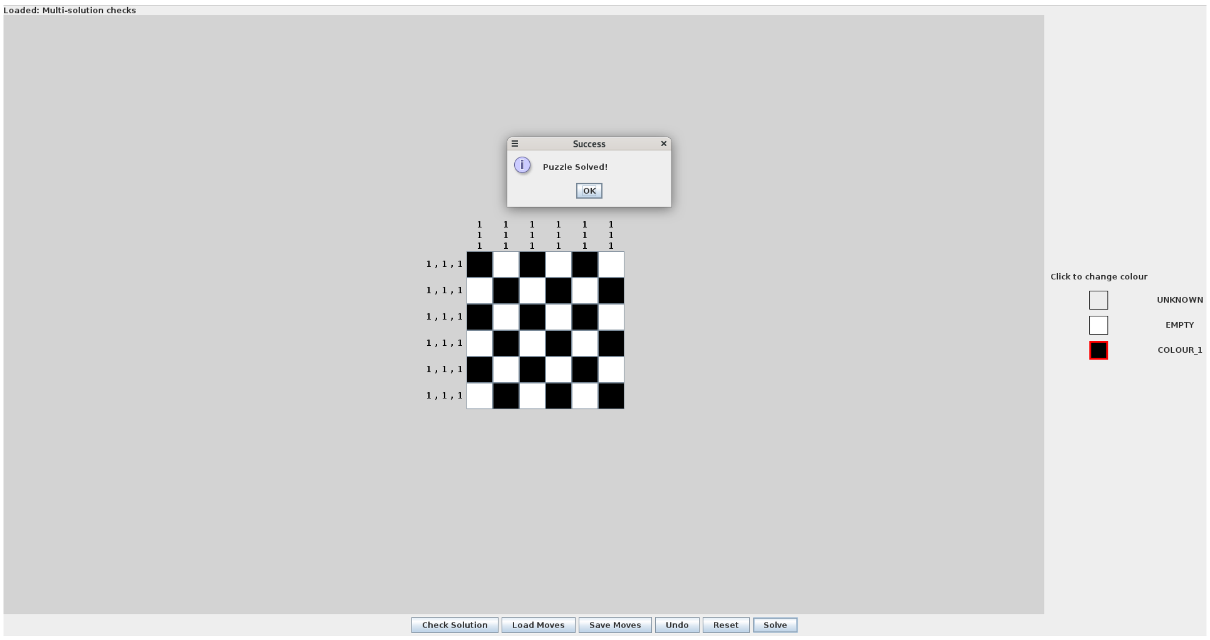


Success

Puzzle Solved!

OK

Figure 36:

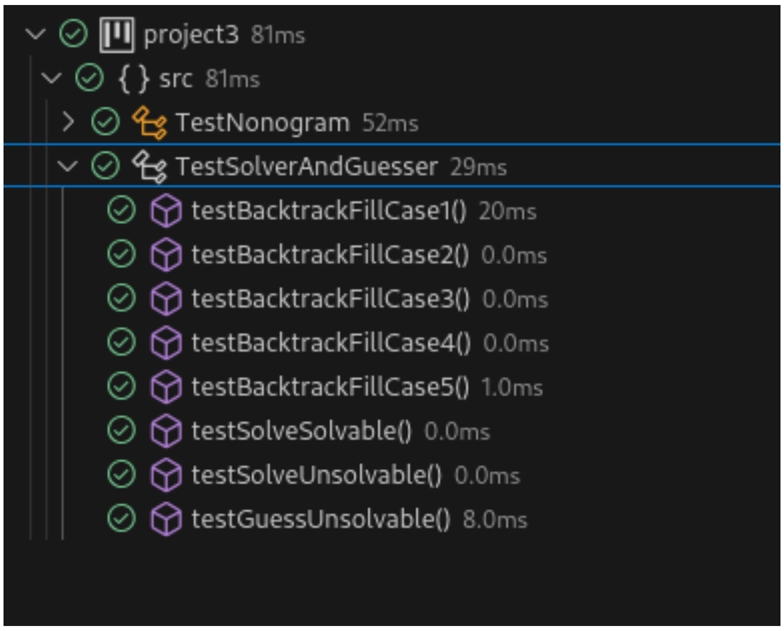


Success

Puzzle Solved!

OK

Figure 37:



```
project3 81ms
  src 81ms
    TestNonogram 52ms
    TestSolverAndGuesser 29ms
      testBacktrackFillCase1() 20ms
      testBacktrackFillCase2() 0.0ms
      testBacktrackFillCase3() 0.0ms
      testBacktrackFillCase4() 0.0ms
      testBacktrackFillCase5() 1.0ms
      testSolveSolvable() 0.0ms
      testSolveUnsolvable() 0.0ms
      testGuessUnsolvable() 8.0ms
```