CS5011-P1: Intelligent Mosaic Game Solver

Student ID: 220034672

Submission Date: 17th February 2023

1 Introduction

1.1 Assignment Details

This document outlines the development and evaluation of an intelligent agent designed to solve Mosaic puzzles, a logic game played on an N x N grid. The project was submitted on the 17th of February, 2023, by student 220034672.

1.2 Implementation Checklist

- Part A – Fully implemented and fully working.
- Part B – Fully implemented and fully working.
- Part C1 – Fully implemented and fully working.

1.3 How to Run

To run the project, navigate to the `src` directory and compile the Java classes using the following command:

```
javac -cp .:../lib/logicng-2.4.1.jar *.java
```

Then, execute the test suite via:

```
stacscheck /cs/studres/CS5011/Coursework/P1/Tests
```

2 Design and Implementation

2.1 Brief Problem Definition

The project entails creating an agent capable of solving the Mosaic game, requiring logical reasoning to uncover a hidden picture by interpreting clues regarding the number of adjacent painted cells.

2.2 Software Architecture

A tiered architecture was adopted, consisting of:

Presentation Layer: Command-line interface for user interactions.

Application Layer: The `P1main` class controls input handling and game flow.

Logic Layer: Includes the `Game` class for state management and various Agent classes for solving strategies.

Data Access Layer: Manages game configurations via input files.

External Libraries: Utilizes LogicNG, SAT4J, and ANTLR Runtime for logic and probabilistic reasoning.

2.3 Key Elements of Implementation

The implementation leverages arrays for the game board, strings for state representation, and integers for cell states and clues. Detailed algorithms for AgentA, AgentB and AgentC1 are provided, emphasizing initialization, strategy application, and final status determination.

2.4 Key Elements of Implementation

The implementation utilizes several data structures for efficient operations:

- Arrays for the game board facilitate quick access and manipulation of cells.
- Strings to represent game states and inputs, allowing for straightforward parsing and debugging.
- Integers for storing cell states and clues, offering memory-efficient storage and easy arithmetic operations.

2.4.1 Algorithm for AgentA

AgentA determines the final status of the game based on the current state of the Mosaic board, with the following steps:

1. **Initialization:** Parse the input string to create a 2D array of *Cell* objects, each holding its state and clue.
2. **Board Evaluation:** Iterate over each cell to check if it is covered or uncovered:
    - Mark the game as not fully processed if any cell remains covered.
    - Compare the clue with the actual count of painted cells in the adjacent eight cells, including the cell itself, for uncovered cells.

3. **Consistency Check:** Evaluate if the clues are consistent with the number of adjacent painted cells:
    - Mark clues as inconsistent if a cell's clue does not match the count of painted neighbours.
4. **Final Status Determination:** Determine the final status of the game:
    - Conclude that the game is solved correctly if all cells are processed and all clues are consistent.
    - Identify the game as incomplete or incorrectly solved if any cell remains covered or any clue lacks consistency.
5. **Result Output:** Return a final status code representing the game's outcome.

The algorithm is designed for its straightforward and efficient evaluation of the game state, based on simple iteration and condition checking. The use of a primary 2D array of cells and fundamental operations ensures swift execution, which is suitable for the game's assessment phase.

2.4.2 Algorithms for AgentB

AgentB algorithmically applies a series of logical single point moves based on the game's clues and the current state of each cell. It employs three main strategies:

1. **Direct Clue Application (DCA):** Identifies cells that must be painted based on directly comparing clues and the count of covered neighbours. If the sum of a clue and the painted count equals the number of covered neighbours, those cells are painted, marking progress on the board.
2. **Completion Strategy (CS):** Focuses on clearing cells when the number of painted neighbours matches the cell's clue, indicating no further paintings are necessary in that vicinity. This strategy aids in deducing cells that can be cleared.
3. **Contradiction Avoidance (CA):** Prevents incorrect moves by ensuring that painting a cell does not violate the clues of its neighbours. It checks if painting a covered cell would make it impossible to satisfy adjacent cells' clues, thereby avoiding contradictions.

Each strategy iterates over the game board, applying its logic to make changes—either painting or clearing cells—and continues until no further actions can be made based on the given strategy. The effectiveness of AgentB lies in its ability to incrementally solve parts of the puzzle, making it a crucial step towards fully solving or significantly advancing the game state. These strategies balance straightforward implementation and strategic depth, allowing AgentB to progress the game towards a solution effectively.

2.4.3 Algorithm for AgentC1

AgentC1 utilizes logic programming to solve the Mosaic game by converting game clues into a Disjunctive Normal Form (DNF) sentence and then using the LogicNG library to solve the SATs.

1. **Initialization:** AgentC1 starts with the game board and uses a formula factory for logic operations.
2. **DNF Conversion:** Each cell with a clue generates logical clauses representing possible states (painted or cleared) that satisfy the clue. These clauses are combined into a DNF sentence for the whole board.
3. **SAT Solving:** The DNF sentence is fed into an SAT solver, which attempts to find a satisfying assignment representing a solution to the board.
4. **Game State Update:** If a solution is found, the agent applies the solution to the game state, marking cells as painted or cleared based on the SAT model.
5. **Repeat Process:** This process repeats until no further changes are made, indicating that the board is either solved or cannot be further resolved with the given clues.
6. **Final Evaluation:** Finally, it assesses the game's status by converting the board into a format understandable by AgentA for final status determination.

Using DNF sentences and SAT solving allows AgentC1 to methodically explore and apply logical constraints derived from the game's clues, offering a robust and systematic approach to uncovering the game board's solution. This strategy is particularly effective for complex puzzles with a few variables for polynomial-time completeness.

3 Test Summary

3.1 Overall Testing Outcomes

The agents (AgentA, AgentB, and AgentC1) were validated against 18 test cases, demonstrating successful outcomes across various game states and conditions, as shown in Figure 1.

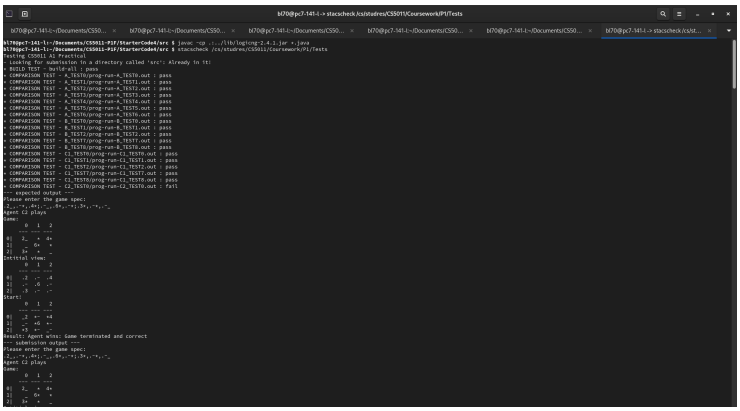

Figure 1: Overall testing outcomes

4 Iterative Testing

The development of Agents for the Mosaic game was approached systematically to ensure each method's correct functioning. This process involved strategically inserting print statements throughout the codebase to monitor the internal state and logic flow during execution. These printouts served two primary purposes:

1. **Validation of the Agent's Logic:** Print statements provided real-time feedback on operations like board initialization, adjacent cell count, and final game status determination. This was crucial for verifying the accuracy of the Agent's knowledge base and the correctness of board and clue-processing logic.

2. **Identification of Issues:** Printouts played a critical role in pinpointing the exact location and nature of problems, facilitating targeted debugging efforts.

This iterative process of inserting print statements, reviewing outputs, and refining the code continued until all methods functioned as expected under various test scenarios. Following verification, print statements were commented out to streamline the code for production, resulting in a thoroughly tested agent capable of reliably performing the required functionality.

5 Evaluation and Conclusion

5.1 Summary of AgentA's Performance

Agent A demonstrated quick processing of Mosaic games, with execution times ranging from 1.8 ms to 2.37 ms, indicating high speed and efficiency. However, evaluating its performance fully requires considering higher game board complexities and scalability.

Test ID	Execution Time (ms)
0	2.12
1	1.8
2	1.89
3	2.12
4	1.98
5	1.99
6	2.37

Table 1: Execution times of Agent A

5.1.1 Overview of Agent A:

- Efficiency: Execution times consistently under 3 ms.
- Consistency: Uniform performance across different tests.

5.1.2 Enhancements of Agent A:

- Test scalability with more complex boards.
- Optimize based on identified inefficiencies and improve logging for more precise identification of failure reasons.

5.2 Summary of AgentB's Performance

Agent B's execution times varied from 3.24 ms to 27.06 ms, indicating its capability to effectively preprocess and simplify puzzles before employing Agent A for the final status determination. Though, the variability highlights areas for efficiency improvement, especially in scenarios where no solution is found directly.

Test ID	Execution Time (ms)
0	3.24
1	3.36
2	27.06
7	4.7
8	4.06

Table 2: Execution times of Agent B

5.2.1 Overview of Agent B:

- Preprocessing Strength: Effective board simplification strategies.
- Robustness: Ability to solve a diverse set of puzzles.
- Performance Variability: Time inconsistencies observed across tests.

5.2.2 Enhancements of Agent B:

- Explore parallel execution of strategies for efficiency.
- Implement advanced heuristics for more effective problem-solving.
- Produce tests for commented out strategies including Safe Opening, Neighbor Clue Integration and Clue Summation mentioned at the bottom of AgentB.

5.3 Summary of AgentC1's Performance

Agent C1, utilizing DNF sentences and SAT solving, showed a sophisticated approach to puzzle-solving with execution times reaching up to 131.93 ms. While excellent at handling complex puzzles, it faces efficiency challenges, primarily due to the computational cost of constructing DNF sentences and the SAT solving process.

Test ID	Execution Time (ms)
0	95.23
1	83.24
2	83.64
7	126.15
8	131.93

Table 3: Execution times of Agent C1

5.3.1 Overview of Agent C1:

- Advanced Logic Use: Employs complex solving techniques effectively.
- Versatility: Capable of adapting to various puzzle complexities.

5.3.2 Enhancements of Agent C1:

- Streamline the construction of DNF sentences for efficiency.
- Apply heuristic-based SAT solving to expedite finding solutions.
- Utilize parallel processing to enhance performance.
- Adjust strategy application based on puzzle complexity to optimize resource usage.

5.4 General Conclusion

Agents A, B, and C1 each exhibit distinct strengths in solving Mosaic puzzles. While Agents A and B offer faster but simpler solutions, Agent C1 provides a complex, albeit slower, approach. By focusing on the outlined enhancements, their efficiency and problem-solving capabilities can be further improved.

5.4.1 Future Work

For future iterations of this project, I aim to:

- Explore alternative solving methodologies, such as the suggested Agent's C2, C3 and possibilites mentioned for Agent D as well as heuristic-based search algorithms, that might offer improved adaptability and efficiency.
- Integrate machine learning techniques to enable the agents to learn from past puzzles and improve their performance over time.
- Develop a graphical user interface to visualize the agent's problem-solving process, which could be an invaluable tool for both debugging and educational purposes.

In conclusion, while all agents implemented have shown promise in solving Mosaic puzzles, there is ample room for refinement. With the outlined enhancements and future work, I anticipate significant improvements in their problem-solving capabilities and efficiency.