Sudoku Game

SUTDENTS:

-Abdelrahman Ahmed Sobhy Mohamed

Id:20220252

-Ahmed Ameen Ahmed Ameen

Id:20220011

-George Ibrahim Aziz

Id:20220128

-Abdelrahman Mohamed Osama

Id:20220264

-Mohamed Mahmoud Mousa

Id:20220429

-Abdelrahman Mostafa Mohamed

Id:20220267

1. Introduction

This project presents an implementation of a **Sudoku game and solver** using the **Imperative Programming Paradigm**.

The purpose of this implementation is to demonstrate:

- Object-Oriented Programming (OOP)
- Mutable program state
- Explicit control flow using loops
- Iterative backtracking algorithm
- Use of higher-order functions via callbacks

The Sudoku solver modifies the game state directly and solves the puzzle step by step using commands and loops, which clearly reflects imperative programming principles.

---

2. Sudoku Problem Description

Sudoku is a logic-based puzzle played on a **9×9 grid**, divided into **nine 3×3 subgrids**.

The objective is to fill the grid so that:

- Each row contains numbers from 1 to 9 exactly once
- Each column contains numbers from 1 to 9 exactly once
- Each 3×3 subgrid contains numbers from 1 to 9 exactly once

Empty cells are represented by the value **0**.

---

3. Programming Paradigm Used

This implementation follows the **Imperative Paradigm**, which is characterized by:

- Explicit instructions that change program state
- Step-by-step execution control
- Mutable variables and data structures
- Commands that directly modify memory

---

4. System Design

4.1 Object-Oriented Structure

The system is implemented using a single class:

`SudokuGame`

This class encapsulates:

- The Sudoku board (mutable state)
- Game logic
- Validation logic
- Solving logic
- User interaction

---

5. Board Representation

The Sudoku board is represented as a **2D list (9×9 matrix)**:

- Numbers (1–9): filled cells
- Zero (0): empty cells

The board is stored internally as:

```
self.board
```

This state is **mutable** and modified directly during gameplay and solving.

---

6. Core Methods Explanation

6.1 `print_board()`

This method prints the Sudoku board in a formatted way:

- Uses dots (`.`) to represent empty cells
- Adds visual separators for rows and columns
- Improves readability for the user

This method only displays state and does not modify it.

6.2 `is_valid(row, col, num)`

This method checks whether a number can be placed in a given position.

Validation steps:

1. Check the row
2. Check the column
3. Check the corresponding 3×3 subgrid

Returns:

- `True` if the move follows Sudoku rules
- `False` otherwise

6.3 `is_complete()`

This method checks whether the puzzle is fully solved.

- If no cell contains 0 → puzzle is complete
- Otherwise → puzzle is incomplete

6.4 `find_empty_cell()`

This method scans the board and returns the position of the first empty cell.

- Returns `(row, col)` if found
- Returns `None` if the board is full

---

7. Iterative Sudoku Solver (AI)

7.1 Algorithm Overview

The solver uses an **ITERATIVE BACKTRACKING ALGORITHM** instead of recursion.
Main steps:

1. Collect all empty cells at the start
2. Traverse empty cells using an index variable
3. Try numbers from 1 to 9 for each cell
4. If no number fits → explicitly backtrack
5. Continue until the board is solved or no solution exists

This approach provides full control over execution using loops.

7.2 `solve_sudoku(callback=None)`

The `solve_sudoku` method performs the automated solving process.

Key characteristics:

- Uses `while` and `for` loops
- Directly mutates `self.board`
- Implements backtracking explicitly using index manipulation
- No recursion is used

---

8. Higher-Order Function Usage (Callback)

The solver demonstrates the use of a **higher-order function** through the `callback` parameter.

A higher-order function:

- Accepts another function as an argument

In this project:

- `solve_sudoku` receives a `callback`
- The callback is called after each significant board modification
- This enables visualization or logging without changing the solving logic

Example use case:

- Visualizing the solving process step by step
- Debugging algorithm behavior
- Separating logic from presentation

---

9. Manual Gameplay

The program allows the user to play Sudoku manually. **`play()`**

Method

- Prompts user to enter row, column, and number
- Validates inputs
- Prevents overwriting filled cells
- Updates board state directly (imperative style)
- Ends when the puzzle is solved or user exits

---

10. Execution Modes

At runtime, the user can choose:

1. **Manual Play** – user solves the puzzle
2. **AI Solve with Visualization** – solver with callback visualization
3. **AI Solve Fast Mode** – solver without visualization

---

11. Key Imperative Programming Characteristics

This implementation clearly demonstrates:

- Mutable shared state
- Explicit step-by-step control
- In-place state modification
- Loop-based algorithm design
- Side effects during execution
- Higher-order function usage

---

3. Declarative / Functional Sudoku Solver

3.1 Overview

The declarative version of the Sudoku game is implemented using **functional programming principles**.
This approach focuses on **what** the solution should be rather than **how** to change the program state step by step.

In this implementation:

- The Sudoku board is treated as **immutable data**
- No function modifies the board directly
- Each move produces a **new board state**
- Recursion is used instead of loops
- Higher-order functions are used for visualization and interaction

---

3.2 Functional Programming Principles Used

1. Immutability

The Sudoku grid is never modified directly.
Instead, every valid move creates and returns a **new grid**.

Example:

```
new_state = apply_move(state, row, col, num)
state = new_state
```

This ensures:

- No side effects
- Easier debugging
- Clear separation between states

---

2. Pure Functions

Most functions are **pure**, meaning:

- They depend only on their inputs
- They do not modify external variables
- They always return the same output for the same input

Examples of pure functions:

- `is_valid(board, row, col, num)`

- `is_complete(board)`
- `find_empty_cell(board)`
- `apply_move(board, row, col, num)`

---

3. Recursion Instead of Loops

Traditional loops (`for`, `while`) are avoided.
Instead, recursion is used to traverse the board and control program flow.

Examples:

- Printing the board using recursive row and cell traversal
- Searching for empty cells recursively
- Trying Sudoku numbers recursively (`try_num`)

This reinforces the declarative style and avoids imperative control flow.

3.3 Board Validation Logic

The function `is_valid` checks Sudoku constraints **without modifying the board**.

It validates:

- Row constraint
- Column constraint (using recursion)
- 3×3 sub-grid constraint (using recursion)

This guarantees that every move follows Sudoku rules.

3.4 Applying a Move (Immutable State Transformation)

The function `apply_move` is the core of immutability:

```
def apply_move(board, row, col, num) -> Optional[Grid]:
```

Behavior:

- If the cell is not empty → returns `None`
- If the move violates Sudoku rules → returns `None`
- Otherwise → returns a **new board**

The new board is built recursively row by row, ensuring:

- The original board remains unchanged
- Only the selected cell is updated

3.5 Functional Backtracking Solver

The function `solve_sudoku` implements a **pure recursive backtracking algorithm**.

Key characteristics:

- No shared mutable state
- Each recursive call works on a new board
- Backtracking is done by returning `None`, not by undoing changes

This function returns:

- A solved board if a solution exists
- `None` if the puzzle is unsolvable

3.6 Higher-Order Function Usage

The solver supports a **callback function** as a parameter:

```
solve_sudoku(board, callback)
```

This demonstrates a **higher-order function**, where:

- A function is passed as an argument
- The callback is invoked during solving steps

Purpose:

- Visualization of the solving process
- Separation of logic and side effects
- Better modularity

Example usage:

```
def show_progress(board):
    print_board(board)
```

```
solve_sudoku(initial_board, show_progress)
```

3.7 Functional Game Loop

The `play` function demonstrates **functional state replacement**.

Instead of modifying the board:

- The current state is replaced with a new one
- The game loop is implemented recursively

```
state = new_state
```

This aligns with functional programming philosophy:

Replace state, don’t mutate it.

3.8 Advantages of the Declarative Approach

- Clear logical flow
- No side effects
- Safe state handling
- Easier reasoning about correctness
- Strong alignment with functional programming concepts taught in AI and PL courses

---

3.9 Limitations

- Slower execution due to:
    - Recursive calls
    - Creation of new board copies
- Higher memory usage
- Less efficient compared to imperative backtracking

3.10 Summary

The declarative Sudoku solver emphasizes:

- **Correctness over performance**
- **Immutability over mutation**

- **Recursion over loops**
- **Higher-order functions for flexibility**

This makes it ideal for academic purposes and for demonstrating functional programming concepts clearly.

---

12. Conclusion

This project provides a clear example of solving Sudoku using **Imperative Programming**.

The iterative backtracking algorithm highlights the importance of:

- Explicit control flow
- State management
- Mutable data structures

The addition of higher-order functions improves flexibility by separating logic from visualization, making the design more modular while remaining imperative in nature.