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 two different implementations of a Sudoku game and solver, each based on a distinct programming paradigm:

1. **Imperative Programming Paradigm**
2. **Declarative / Functional Programming Paradigm**

The main goal of the project is to compare how the same problem (Sudoku solving) can be approached using different paradigms, highlighting their design philosophies, strengths, and limitations.

Through this comparison, the project demonstrates: - How program state is handled (mutable vs immutable) - Different control flow mechanisms (loops vs recursion) - Backtracking strategies in both paradigms - Use of higher-order functions for visualization and modularity

---

2. Project Overview

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 the numbers 1–9 exactly once - Each column contains the numbers 1–9 exactly once - Each 3×3 subgrid contains the numbers 1–9 exactly once

Empty cells are represented by the value 0.

Both implementations solve the same problem but follow very different design philosophies, making Sudoku an ideal case study for paradigm comparison.

---

3. Imperative Sudoku Solver

3.1 Programming Paradigm Used

The imperative implementation follows the Imperative Programming Paradigm, characterized by: - Explicit instructions that change program state - Step-by-step execution control - Mutable variables and data structures - Commands that directly modify memory

---

3.2 System Design

3.2.1 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

---

3.3 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.

3.4 Core Methods

3.4.1 `print_board()`

- Prints the board in a formatted way
- Uses dots (.) for empty cells
- Adds visual separators for readability
- Does not modify the game state

3.4.2 `is_valid(row, col, num)`

Checks whether a number can be placed at a given position by validating: 1. The row 2. The column 3. The corresponding 3×3 subgrid

Returns: - `True` if the move is valid - `False` otherwise

3.4.3 `is_complete()`

- Checks whether the board contains any empty cells
- Returns `True` if solved, otherwise `False`

3.4.4 `find_empty_cell()`

- Scans the board for the first empty cell
- Returns `(row, col)` if found

- Returns `None` if the board is full

3.5 Iterative Backtracking Solver

3.5.1 Algorithm Overview

The solver uses an iterative backtracking algorithm (no recursion): 1. Collect all empty cells at the start 2. Traverse them using an index variable 3. Try numbers from 1 to 9 4. If no number fits, explicitly backtrack 5. Continue until solved or no solution exists

This approach provides full control over execution using loops.

3.5.2 `solve_sudoku(callback=None)`

Key characteristics: - Uses `while` and `for` loops - Directly mutates `self.board` - Explicit index-based backtracking - No recursion

---

3.6 Higher-Order Function Usage

The imperative solver supports a callback function: - Passed as a parameter to `solve_sudoku` - Invoked after significant board changes

Purpose: - Visualization - Debugging - Separation of logic from presentation

---

3.7 Manual Gameplay and Execution Modes

The program supports: 1. Manual play 2. AI solving with visualization 3. Fast AI solving without visualization

The `play()` method: - Accepts user input - Validates moves - Updates board state directly - Ends when solved or exited

---

3.8 Imperative Characteristics Summary

- Mutable shared state
- Explicit step-by-step control
- Loop-based algorithms
- In-place state modification
- Side effects during execution

---

4. Declarative / Functional Sudoku Solver

4.1 Overview

The declarative version is implemented using functional programming principles. It focuses on what the solution is, rather than how the state is modified.

Key ideas: - Immutable board state - No direct state modification - Recursion instead of loops - Pure functions

---

4.2 Functional Programming Principles Used

4.2.1 Immutability

- The board is never modified directly
- Every move produces a new board
- Previous states remain unchanged

4.2.2 Pure Functions

Functions: - Depend only on inputs - Have no side effects - Always return the same output for the same input

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

4.3 Recursion-Based Control Flow

- Loops are avoided
- Recursion is used for:
    - Traversing the board
    - Searching for empty cells
    - Trying candidate numbers

4.4 Immutable State Transformation

The `apply_move` function: - Returns `None` if the move is invalid - Returns a new board if valid - Builds the new board recursively

This guarantees: - No side effects - Safe state transitions

4.5 Functional Backtracking Solver

The solver: - Uses recursive backtracking - Never undoes changes - Backtracks by returning `None`

Returns: - A solved board if successful - `None` if unsolvable

4.6 Higher-Order Function Usage

The functional solver also supports callbacks: - Passed as a function argument - Used for visualization or logging - Keeps logic pure and modular

---

4.7 Functional Game Loop

- State is replaced, not mutated
- The game loop is recursive

This aligns with functional philosophy: > Replace state, don’t mutate it.

4.8 Advantages and Limitations

Advantages: - Clear logic - No side effects - Easier reasoning - Strong correctness guarantees

Limitations: - Slower execution - Higher memory usage - Less efficient than imperative backtracking

---

5. Comparative Summary

Aspect	Imperative	Declarative / Functional
State	Mutable	Immutable
Control Flow	Loops	Recursion
Backtracking	Explicit undo	Return-based
Performance	Faster	Slower
Reasoning	Harder	Easier

---

6. Conclusion

This project demonstrates how the same problem can be solved effectively using two different programming paradigms.

- The imperative implementation emphasizes performance, explicit control flow, and direct state manipulation.
- The declarative/functional implementation emphasizes correctness, immutability, and clarity of logic.

By comparing both approaches, the project highlights the trade-offs between efficiency and maintainability, making it a valuable academic example for understanding programming paradigms and algorithm design.