THE AMERICAN UNIVERSITY IN CAIRO

Sudoku Solver Project

Spring 2025

CSCE 221102- Applied Data Structure

The American University in Cairo

Ahmed A. Abdelgelil	**900226445**
Ahmed A. Amin	**900202813**
Beshoy M. Botros	**900226446**
Dalia T. El-Masry	**900223685**
Sara Sharkawy	**900225825**

Submitted to:

Dr. Dina G. Mahmoud

Table of Contents:

1. Project Overview....................3
    1.1 Introduction....................3
    1.2 Problem Definition....................3
    1.3 High-Level Architecture....................4
        1.3.1 Architectural Overview....................4
        1.3.2 Class Diagram....................4
        1.3.3 Algorithm Flowchart....................5
2. Methodology....................7
    2.1 Input Layer Description....................7
    2.2 Solver Layer Description....................9
    2.3 InterProcess Communication Description....................10
    2.4 Graphical User Interface Description....................11
3. Data Structure Specification....................12
    3.1 Graph Specification....................12
    3.2 Stack Specification....................14
    3.3 Original Code....................15
4. Experimental Results....................16
    4.1 Backtracking Performance Results....................16
    4.2 Constraint Propagation Performance Results....................16
    4.3 Dancing Links Performance Results....................17
5. Conclusion....................18

1. Project Overview

1.1 Introduction

The Sudoku Solver project implements a complete solution for solving Sudoku puzzles using graph-based constraint satisfaction techniques. CSP is a type of problem where the solution must satisfy a set of restrictions. In general, a CSP consists of variables, each one of them has a domain of possible values, and a set of constraints that limits the values that the variables can take.The solver can process puzzles from text files or direct input, solve them using backtracking algorithms, and display the results either in a command-line interface or through a modern web-based GUI.Sudoku is a logic-based number placement puzzle where the objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids contains all of the digits from 1 to 9. The puzzle starts with a partially completed grid, and the solver must find a valid completion.

1.2 Problem Definition

The Sudoku solving problem can be formalized as a constraint satisfaction problem (CSP) with the following components:

- **Variables:** The 81 cells of the 9×9 grid
- **Domains:** Each cell can take values from 1 to 9
- **Constraints:**
    - Each row must contain all digits 1-9 without repetition
    - Each column must contain all digits 1-9 without repetition
    - Each 3×3 subgrid must contain all digits 1-9 without repetition

The solver must assign values to all empty cells while respecting these constraints. A valid solution must satisfy all constraints simultaneously.

1.3 High-Level Architecture

1.3.1 Architectural Overview

The Sudoku Solver project implements a layered architecture that separates concerns between data representation, algorithmic logic, communication, and user interface. This design allows for flexibility in how the solver is used, supporting both command-line operation for batch processing and a web-based graphical interface for interactive use.The architecture consists of four primary layers:Input Layer: Responsible for parsing Sudoku puzzles from various sources (text files, string content, or direct GUI input)

- **Solver Layer:** Contains the core solving algorithms and data structures that represent the Sudoku puzzle as a constraint graph
- **IPC Layer:** Handles communication between the C++ backend and the web frontend through a simple HTTP server
- **GUI Layer:** Provides a user-friendly interface implemented in Vue.js for visualizing and interacting with the solver

This separation of concerns allows each component to be developed, tested, and maintained independently, while also enabling different deployment configurations based on user needs.

1.3.2 Class Diagram

The class diagram illustrates the relationships between the key components of the system:

- The **Graph** class serves as the central data structure, containing a collection of Node objects that represent individual Sudoku cells
- Each **Node** maintains connections to other nodes through Edge objects, representing the constraints between cells
- The **Stack** class supports the backtracking algorithm by tracking moves during the solving process
- The **SudokuGuiIpc** class facilitates communication with the frontend when running in GUI mode

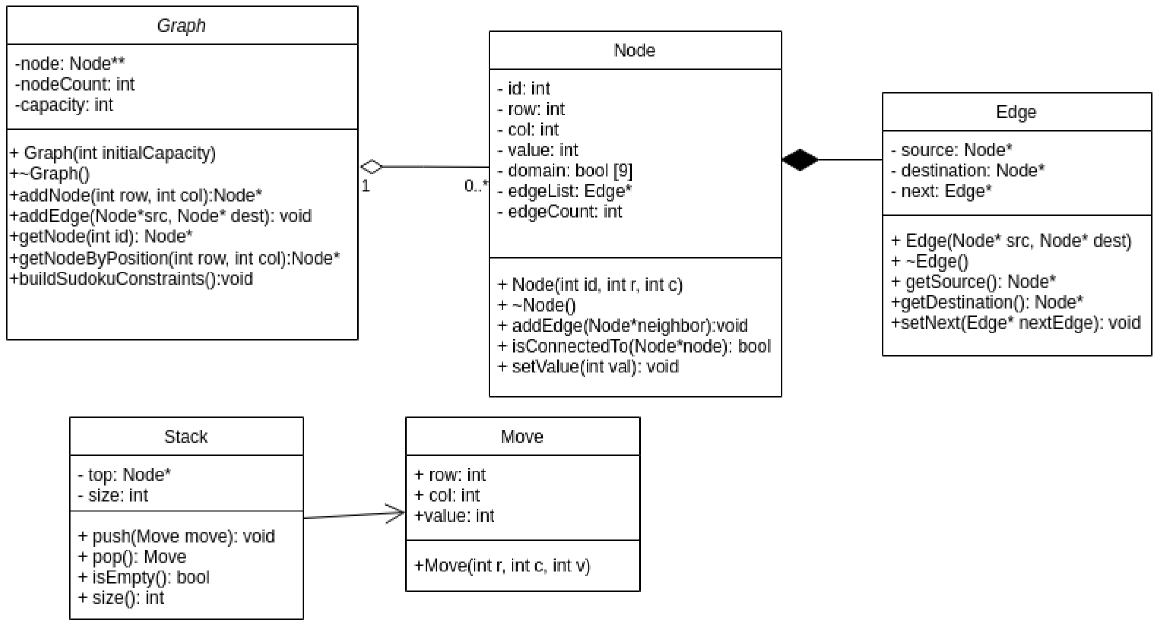

Figure (1): Project Class diagram

1.3.3 Algorithm Flowchart

The core solving algorithm uses backtracking with constraint checking to find a valid solution. The flowchart below illustrates this process:

1. The algorithm begins by creating an empty stack to track moves
2. It scans the grid from left to right, top to bottom, looking for empty cells
3. For each empty cell, it tries values 1-9, checking if each value is valid according to Sudoku constraints
4. When a valid value is found, it's assigned to the cell, and the move is pushed onto the stack
5. If no valid value can be found for a cell, the algorithm backtracks by popping the last move from the stack and trying the next possible value
6. This process continues until either all cells are filled (solution found) or all possibilities are exhausted (no solution exists)

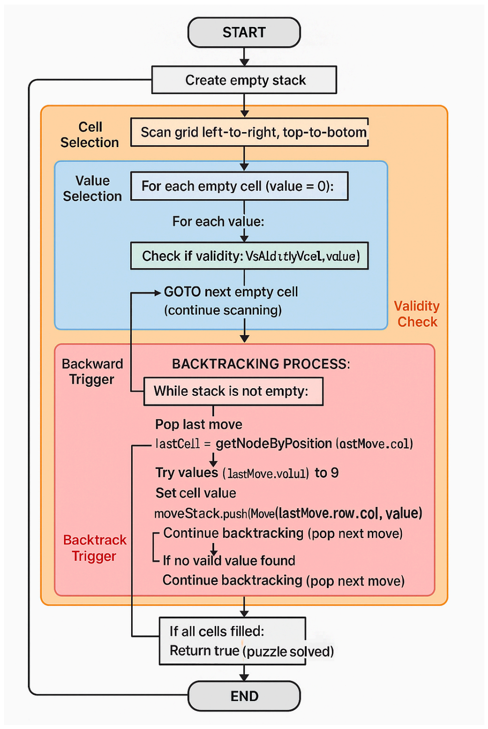
START
Create empty stack
Cell Selection
Scan grid left-to-right, top-to-botom
Value Selection
For each empty cell (value = 0):
For each value:
Check if validity: VsAldttlyVcel, value)
GOTO next empty cell (continue scanning)
Validity Check
Backward Trigger
BACKTRACKING PROCESS:
While stack is not empty:
Pop last move
lastCell = getNodeByPosition (astMove.col)
Try values (lastMove.volul) to 9
Set cell value
moveStack.push(Move(lastMove.row.col, value)
Continue backtracking (pop next move)
If no vaild value found
Continue backtracking (pop next move)
Backtrack Trigger
If all cells filled:
Return true (puzzle solved)
END

Figure (2): Algorithmic Flowchart

This backtracking approach systematically explores the solution space, pruning invalid branches early to improve efficiency. The algorithm is guaranteed to find a solution if one exists, though the worst-case time complexity is exponential.

2. Methodology

2.1 Input Layer Description

The input layer of the Sudoku Solver is responsible for parsing and validating Sudoku puzzles from various sources. This layer provides flexibility in how puzzles are loaded into the system, supporting multiple input formats and sources.

File Input Processing

The primary input method is through text files, implemented in the readSudokuFromFile function. This function:

1. Opens and reads a specified file
2. Parses the content line by line, extracting numerical values
3. Populates the graph data structure with the initial puzzle state
4. Validates the input format, ensuring it contains a valid 9×9 grid

String Content Processing

For web-based input, the system also supports parsing puzzles directly from string content via the readSudokuFromString function. This allows puzzles to be uploaded through the web interface without requiring local file access on the server.

Direct Grid Input

The system also supports direct grid input through the loadSudokuFromGrid function, which accepts a 2D vector of integers representing the puzzle state. This is particularly useful for the GUI interface where users might input values directly into the grid.

Input Validation

All input methods include validation to ensure:

- The grid is exactly 9×9
- All values are either empty (0) or valid digits (1-9)
- The initial state doesn't violate any Sudoku constraints

2.2 Solver Layer Description

The solver layer implements the core algorithm for solving Sudoku puzzles using a backtracking approach with constraint propagation.

Graph-Based Representation

The puzzle is modeled as a constraint graph where:

- Each cell is represented by a Node object
- Constraints between cells (same row, column, or 3×3 box) are represented by Edge objects
- The **Graph** class manages the overall structure and enforces Sudoku rules

This representation allows for efficient constraint checking and propagation during the solving process.

Backtracking Algorithm

The main solving algorithm uses depth-first search with backtracking:

- Find the first empty cell in the grid
- Try placing digits 1-9 in this cell
- For each digit, check if it violates any constraints
- If valid, place the digit and recursively solve the rest of the grid
- If no valid digit can be placed or the recursive call fails, backtrack by undoing the last placement

Constraint Checking

The constraint checking is performed by the isValidSudokuValue method in the Graph class, which verifies that a value doesn't conflict with any connected nodes (cells in the same row, column, or box).

2.3 InterProcess Communication Description

The IPC layer facilitates communication between the C++ backend solver and the web-based frontend, allowing users to interact with the solver through a modern interface.

HTTP Server Implementation

The SudokuGuiIpc class implements a simple HTTP server that:

1. Listens for incoming connections on a specified port
2. Processes HTTP requests from the frontend
3. Sends responses containing puzzle data or solving results

JSON-Based Communication Protocol

Communication between the backend and frontend uses JSON for data serialization:

- Puzzle grids are sent as 2D arrays
- Status messages include success/failure indicators and descriptive text
- Commands from the frontend specify actions like "solve" or "load sample"

Callback-Based Event Handling

The IPC layer uses a callback architecture to handle events from the frontend:

- **onPuzzleReceived** - Called when a new puzzle is input directly in the GUI
- **onSolveRequested** - Called when the user requests the puzzle to be solved
- **onFileUploaded** - Called when a puzzle file is uploaded through the web interface

2.4 Graphical User Interface Description

The GUI layer provides a user-friendly interface for interacting with the Sudoku solver, implemented as a web application using Vue.js and Vuetify.

Vue.js Component Architecture

The frontend is built using Vue.js components:

- SudokuBoard.vue - The main component displaying the puzzle grid and controls
- Supporting components for file input, status messages, and solution display

Responsive Grid Display

The Sudoku grid is rendered using a responsive layout that:

- Adapts to different screen sizes
- Visually distinguishes original puzzle values from solved values
- Provides visual feedback during the solving process

User Interaction Features

The GUI provides several ways for users to interact with the solver:File upload for custom puzzles

- Sample puzzle selection from predefined options
- One-click solving with visual feedback
- Clear status messages indicating success or failure

3. Data Structure Specification

3.1 Graph Specification

The graph data structure is the core component of the Sudoku solver, representing the puzzle as a constraint satisfaction problem. It consists of three main classes: Edge, Node, and Graph.

Edge Class

The Edge class represents constraints between Sudoku cells, forming a linked list of connections for each node.

Attributes:

- Node* source: Pointer to the source node
- Node* destination: Pointer to the destination node
- Edge* next: Pointer to the next edge in the linked list

Key Methods:

- Edge(Node* src, Node* dest);
- ~Edge();
- Node* getSource() const;
- Node* getDestination() const;
- Edge* getNext() const;
- void setNext(Edge* nextEdge);

The edge implementation uses a singly linked list approach, allowing for efficient traversal of all constraints for a given cell without requiring a dynamic array or STL containers.

Node Class

The Node class represents a single cell in the Sudoku grid, maintaining its position, value, and domain of possible values.

Attributes:

- int id: Unique identifier for the node
- int row, col: Position in the Sudoku grid
- int value: Current assigned value (0 if unassigned)
- bool domain[9]: Array tracking which values (1-9) are still possible
- Edge* edgeList: Head of the linked list of edges
- int edgeCount: Number of edges connected to this node

Key Methods:

- Node(int nodeId, int r, int c);
- ~Node();
- int getValue() const;
- void setValue(int val);
- bool isValueInDomain(int val) const;
- void removeFromDomain(int val);
- void resetDomain();
- int getDomainSize() const;
- void addEdge(Node* neighbor);
- bool isConnectedTo(Node* node) const;

The domain tracking is particularly important for constraint propagation, allowing the solver to efficiently determine which values are still valid for a cell based on its constraints.

Graph Class

The Graph class manages the entire Sudoku puzzle, containing all nodes and providing methods to build and query the constraint network.

Attributes:

- Node** nodes: Dynamic array of node pointers
- int nodeCount: Number of nodes in the graph
- int capacity: Maximum capacity of the nodes array

Key Methods:

- Graph(int initialCapacity = 81);
- ~Graph();
- Node* addNode(int row, int col);
- void addEdge(Node* source, Node* destination);
- Node* getNode(int id) const;
- Node* getNodeByPosition(int row, int col) const;
- void buildSudokuConstraints();
- void printSudokuGrid() const;
- bool isValidSudokuValue(Node* node, int value) const;

3.2 Stack Specification

The Stack class implements a dynamic array-based stack specifically designed for tracking moves during the backtracking process.

Move Structure

The Move structure represents a single placement in the Sudoku grid:

```
struct Move {
    int row;
    int col;
    int value;
    Move(int r = 0, int c = 0, int v = 0);
};
```

This compact representation stores just the essential information needed to track and potentially reverse a move.

Stack Implementation

The Stack class provides a last-in-first-out (LIFO) data structure with dynamic resizing:

Attributes:

- Move* data: Dynamic array of Move objects
- int topIndex: Index of the top element (-1 if empty)
- int capacity: Current capacity of the array

Key Methods:

- Stack(int initialCapacity = 10);
- ~Stack();
- void push(const Move& move);
- Move pop();
- Move top() const;
- bool isEmpty() const;
- int size() const;
- void resize();

The stack is crucial for the backtracking algorithm, allowing it to efficiently track the sequence of moves and revert to previous states when necessary. This implementation avoids the overhead of STL containers while still providing dynamic resizing capabilities.

3.3 Original Code

This the link to the GitHub repository with all the necessary code and documentation. Sudoku Solver Codebase

4. Experimental Results

4.1 Backtracking Performance Results

After doing the experiments for the 15 puzzles using backtracking, most of the puzzles from different difficulty levels were solved in very small execution times. However, it couldn't solve puzzles 4 and 12.

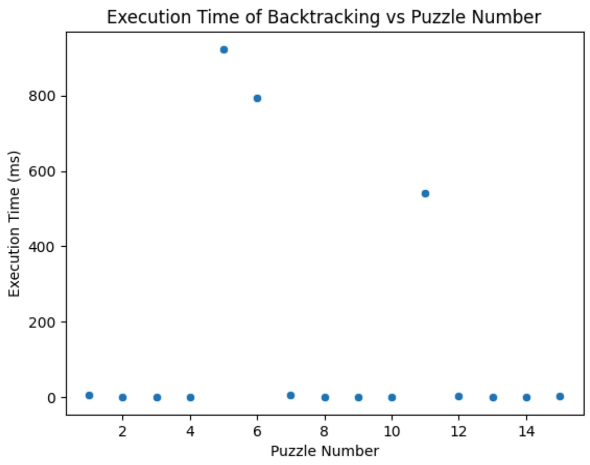

Figure (3): Execution Time Results

4.2 Constraint Propagation Performance Results

After doing the experiments for the 15 puzzles using the Constraint Propagation (AC-3), it was found that compared to the Backtracking algorithm, it takes more time to solve the puzzles. However, it also couldn't solve puzzles 4 and 12. The time complexity of this algorithm is in order of O(ed^3), where e is the number of constraints and d is the domain size.

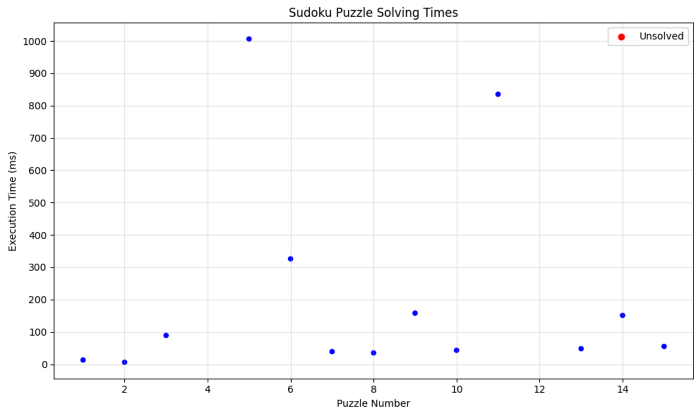

Figure (4): Execution Time Results

4.3 Dancing Links Performance Results

Despite labeled difficulties, DLX solved all puzzles in ≤6 ms, with an 87% success rate. The absence of time scaling suggests that timing resolution masked true differences, or puzzle labels poorly reflect algorithmic complexity. Unsolved cases (Puzzle 4, 12) failed rapidly (2 ms), hinting at implementation limits.

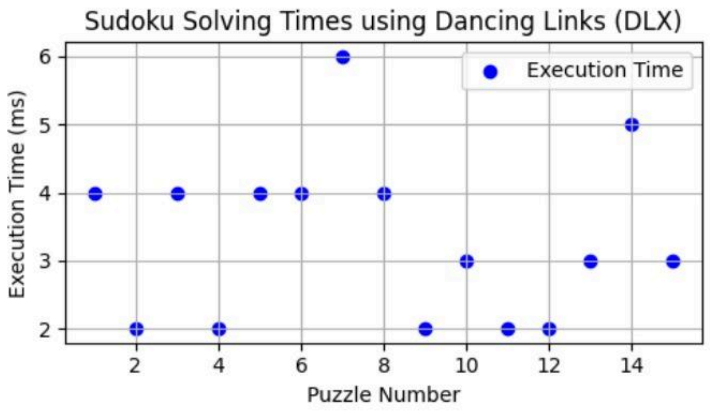

Figure (5): Execution Time Results

The experimental results indicate clear performance differences among the algorithms. While backtracking and constraint propagation show limitations with harder puzzles, Dancing links outperforms both in terms of speed and success rate. It is worth noting that puzzles 4 and 12 were not solved by any of the algorithms due to an issue with the design of these puzzles. The graphs (Figures 3-5) further highlight these trends, demonstrating that more advanced algorithms like Dancing links provide better scalability for solving Sudoku puzzles. These findings align with theoretical expectations and suggest that for applications requiring rapid and reliable Sudoku solving, Dancing links is the preferred approach.

5. Conclusion

The Sudoku Solver project successfully demonstrates the application of graph-based constraint satisfaction techniques to solve Sudoku puzzles efficiently. By implementing custom data structures—including a constraint graph and dynamic stack without relying on standard template libraries, the project provides deeper insights into fundamental computer science concepts. The backtracking algorithm with constraint propagation effectively handles puzzles of varying difficulty levels, while the layered architecture cleanly separates the core solving logic from the user interface concerns.