Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Project Report

on

"DSA GameVault"

[Code No.: COMP 202]

(For partial fulfillment of Second Year/First Semester in Computer Science & Engineering)

Submitted by:

Sushant Timalsina (Roll no: 46 / Reg no: 038010-24)

Prasiddha Timalsina (Roll no: 57 / Reg no: 038008-24)

Siddhant Timalsina (Roll no: 58 / Reg no: 038009-24)

Submitted to

Er. Sagar Acharya

Department of Computer Science and Engineering

Submitted Date: 22nd February, 2026

Bona fide Certificate

This project work on

“DSA GameVault”

is the bona fide work of

“

Sushant Timalsina (Roll no: 46 / Reg no: 038010-24) ,

Prasiddha Timalsina (Roll no: 57 / Reg no: 038008-24) ,

Siddhant Timalsina (Roll no: 58 / Reg no: 038009-24)

”

who carried out the project work under my supervision.

Project Supervisor

Er. Sagar Acharya

Lecturer
Department of Computer Science and Engineering

Acknowledgements

The success and final outcome of this project **”DSA GameVault”** required a lot of guidance and assistance from many people and we are extremely privileged to have got this all along the completion of our project. All that we have done comes down to such excellent supervision and assistance and we would not forget to thank them. We would like to express our sincere thanks of gratitude to our course instructor and evaluator, Er. Sagar Acharya for his constant guidance, encouragement, patience, understanding and recommendations during the advancement of our project, without which, the project would not have been possible. We owe our deep gratitude towards him and his keen interest in our project work despite his busy schedule. We are thankful and fortunate enough to get a platform and assistance from the Department of Computer Science and Engineering to convert our ideas into a practical project and demonstrate their findings. We would once again like to thank all those individuals who have helped us by giving any amount of assistance in our project, including our seniors, faculty members and friends. We hope that our report will be approved and we will be allowed to submit the project.

Thank You.

Abstract

In the contemporary context, educational games have emerged as crucial platforms supporting students in understanding and applying fundamental computer science concepts. Ensuring the accessibility, clarity, and practical demonstration of Data Structures and Algorithms has therefore assumed significant importance. Central to this effort is the systematic integration of algorithmic principles within engaging game environments. Game-based learning encompasses the continual implementation, comparison, and interpretation of essential DSA concepts through interactive gameplay, thereby contributing to improved comprehension, retention, and practical application within the learning process.

In light of this, our intention was to develop an integrated solution that consisted of several DSA applications in real life, **"DSA GameVault"**, that enables comprehensive implementation and visualization of core DSA concepts through three classic games: Sudoku, Snake Game, and Minesweeper. This application implemented recursive backtracking for constraint satisfaction in Sudoku, linked list and queue-based movement mechanics in Snake, and graph traversal algorithms including BFS and DFS for Minesweeper's flood-fill mechanics. The application also includes level selection mechanisms and state preservation features to facilitate user engagement and demonstrate algorithmic behavior across varying difficulty levels. Subsequently, this accumulated implementation had undergone analytical evaluation to identify meaningful relations between algorithm selection and game performance, with the corresponding results presented within the system interface.

Furthermore, the system's capabilities were extended to demonstrate dynamic memory management through linked list implementation in Snake, recursive stack utilization in Sudoku solving, and queue-based expansion techniques in Minesweeper. In addition, the system incorporates mechanisms for assisting both novice and experienced learners through organized level progression, prioritized algorithm demonstration, and streamlined interaction procedures. The implementation also integrates essential DSA concepts and suitable algorithmic strategies to ensure the efficient arrangement and comparison of game states according to relevant criteria. It is pertinent to note that user accessibility remains a principal consideration; thus, the application is designed for smooth operation across desktop environments and requires minimal prerequisites for effective usage.

Keywords: *Sudoku, Snake Game, Minesweeper, Backtracking, Linked List, BFS, DFS, Recursion, Qt Framework*

Contents

List of Figures

Abbreviations

DSA – Data Structures and Algorithms

DFS – Depth-First Search

BFS – Breadth-First Search

UI – User Interface

IDE – Integrated Development Environment

STL – Standard Template Library

RAM – Random Access Memory

GPU – Graphics Processing Unit

SSD – Solid State Drive

QML – Qt Modeling Language

LIFO – Last-In-First-Out

FIFO – First-In-First-Out

Chapter 1: Introduction

1.1 Background

Learning Data Structures and Algorithms has become an important part of computer science education, especially in areas where theoretical concepts often seem abstract and disconnected from practical applications. Students frequently face challenges in understanding how algorithms operate in real-time systems and how data structures manage information efficiently. Many educational platforms provide theoretical explanations, but the practical demonstration is often limited to static code examples, making comprehension time-consuming and conceptually difficult. Additionally, algorithm visualization is usually presented through diagrams, which can lead to an incomplete understanding of dynamic behavior and runtime execution.

With the growing need for practical demonstration and interactive learning, there is a demand for a system that can intelligently visualize algorithmic behavior and demonstrate data structure operations efficiently. The proposed ”DSA GameVault” addresses this gap by using three classic games: Sudoku, Snake, and Minesweeper to demonstrate recursive backtracking, linked list management, queue operations, stack-based state preservation, and graph traversal algorithms including BFS and DFS. By integrating algorithm-driven game mechanics and interactive gameplay, the system can help both students on the subject and experienced learners save time, avoid conceptual errors, and develop more intuitive understanding of DSA concepts. Furthermore, the application provides a user-friendly interface that allows seamless interaction, ensuring that even users without extensive programming expertise can navigate the platform effectively.

1.2 Objectives

The main objectives of our project are as follows:

- To develop a multi-game application that assists users in understanding Data Structures and Algorithms through interactive gameplay by implementing Sudoku, Snake, and Minesweeper as individual modules.
- To implement recursive backtracking with constraint satisfaction in the Sudoku module, demonstrating depth-first search within a decision tree structure and pruning techniques for eliminating invalid configurations.
- To demonstrate dynamic data structure management in the Snake game through singly linked list implementation for snake body representation, queue-based movement logic following FIFO principles, and stack-based state preservation for play and pause functionality following LIFO principles.

- To enable graph-based traversal algorithms in Minesweeper through BFS and DFS implementations for automatic expansion of connected safe regions, ensuring systematic level-order expansion and proper marking of visited cells through queue management and recursive techniques.

1.3 Motivation and Significance

The main motivation behind this project was to develop an intelligent and practical solution for DSA education that improves conceptual understanding for computer science students. By integrating recursive backtracking, linked list management, graph traversal algorithms, and stack-based state preservation within engaging game environments, the application addresses gaps in existing platforms that lack interactive algorithm demonstration and real-time visualization. This project also provides a real-world scenario to apply C++, Qt framework, and DSA algorithms, enhancing practical skills while delivering a useful and user-friendly educational tool.

Chapter 2: Related and Existing Works

Game-based learning platforms for Data Structures and Algorithms have been widely explored in recent years, especially with the increased use of interactive applications and visualization-based teaching methodologies. Many existing systems provide basic algorithm demonstrations such as sorting visualizations or tree traversals through static diagrams. However, most platforms lack integrated game mechanics that demonstrate multiple DSA concepts simultaneously, real-time algorithm execution visualization, and comparative analysis of algorithmic approaches within engaging gameplay environments.

2.1 Sudoku Solving Algorithms and Constraint Propagation

Several major studies have examined efficient approaches for Sudoku solving using various algorithmic strategies. Chen [4] presents an enhanced approach to solving Sudoku puzzles by improving recursive backtracking with constraint propagation and bitmask techniques, focusing on the implementation of the naked pair strategy. This research demonstrates that adding constraints through naked pair elimination, a technique that eliminates candidates from cells when two cells share identical candidate sets, reduces backtracking steps and improves solving efficiency for difficult puzzles. The solver processes puzzles into bitmask vectors, applies singles and naked pair techniques to minimize candidates, and then uses depth-first search to backtrack and solve the puzzle. Results indicate that while the naked pair strategy slightly increases total steps, it significantly reduces depth-first search computations, making the solver potentially more efficient for difficult puzzles [4].

According to Bhattarai et al. [2] from Kathmandu University, a comparative analysis of Sudoku-solving strategies focusing on recursive backtracking and heuristic-based constraint propagation methods reveals significant performance differences. Using a dataset of 500 puzzles across five difficulty levels, the heuristic approach consistently outperformed backtracking, achieving speedup ratios ranging from 1.27 × in beginner puzzles to 2.92× in expert puzzles. These findings underscore the effectiveness of heuristic strategies, particularly in tackling complex puzzles across varying difficulty levels [2].

Ates and Cavdur [1] propose a new mathematical programming formulation for generating Sudoku puzzles, providing flexibility to control the numbers of matrix entries shown in each column, row, and sub-matrix. The initially developed non-linear program is reformulated as a linear-integer program using variate transformations, and the resulting mathematical program generates puzzles in reasonable time periods using commercial solvers. The study extends this hybrid approach to design a backtracking algorithm-based puzzle generation procedure implemented in a standalone mobile-web game application [1].

2.2 Snake Game Data Structures and Memory Optimization

The classic Snake game has been revisited from a data structural perspective by Bille et al. [3], who investigate the fundamental question of how many bits are required to represent the state of a snake game for constant-time updates. Their research introduces several innovative data structural techniques including a decomposition technique for the problem, a tabulation scheme for encoding small subproblems, and a dynamic memory allocation scheme. The main result demonstrates a data structure that uses optimal space within constant factors, contributing to theoretical understanding of efficient game state representation and update mechanisms [3].

Reinforcement learning approaches for Snake game optimization have also been explored, demonstrating how AI techniques can learn optimal strategies through gameplay experience. These studies demonstrate the intersection of game mechanics with advanced algorithmic techniques, though detailed citation of specific works is omitted here as they fall outside the primary focus of this review.

2.3 Minesweeper Graph Traversal and Game Mechanics

Minesweeper implementation frequently demonstrates fundamental graph traversal algorithms in practical applications. A comprehensive Minesweeper project documented on GitHub [6] implements both single-player and two-player modes with depth-first graph traversal and recursion for automatic tile expansion. The retrospective analysis highlights how depth-first search and recursion relate to Minesweeper mechanics: clicking to open a tile without a hidden mine reveals numbers indicating adjacent mines, and in the case of opening empty tiles (no adjacent mines), adjacent tiles continue to be revealed through recursive traversal or queue-based expansion. This demonstrates practical application of graph algorithms where each cell acts as a node connected to neighboring cells in eight possible directions [6].

The project implements traditional Minesweeper rules where the objective is to clear a rectangular board containing hidden mines without detonating them, with clues provided through numbers indicating adjacent mine counts. The implementation choices between recursion (DFS) and iteration with queues (BFS) for flood-fill operations represent fundamental algorithmic decisions affecting performance and memory utilization [6].

2.4 Gamification of Data Structures and Algorithms

Educational games have emerged as powerful tools for enhancing learning experiences across various subjects and age groups. Hamandi et al. [5] present an innovative approach to combining educational content with interactive gameplay to teach important computer science concepts such as data structures and algorithms. Their project aims to foster active learning by allowing students to solve problems through direct manipulation of nodes and edges of trees and graphs. The game presents a platform for college students to learn and actively practice heap data structure, heapsort algorithm, and minimum weight spanning tree algorithms including Kruskal and Prim. Built using JavaScript, HTML, CSS, and Three.js, the game incorporates a 3D interactive environment with multiple difficulty levels, immediate feedback, and a carefully designed hint system to support stepwise learning and enhance interactive experience [5].

The gamification approach incorporates multiple lives and scoring systems to maintain student engagement while allowing anonymous data collection of mistakes, time spent, and score changes per level. This enables instructors to track improvement and analyze student comprehension without storing identifying information [5].

Existing research and applications show that while the fundamentals of individual algorithm implementations are well-developed, very few works combine multiple games within a single integrated application to demonstrate varied DSA concepts including recursive backtracking, linked list management, queue operations, stack-based state preservation, and graph traversal algorithms. This project addresses these gaps by developing an end-to-end interactive learning system where users engage with Sudoku, Snake, and Minesweeper to observe and understand fundamental Data Structures and Algorithms in practical, engaging contexts.

Chapter 3: Design and Implementation

3.1 System Architecture Overview

The proposed system is a multi-game application developed with the main aim of incorporating fundamental Data Structures and Algorithms (DSA) concepts in an interactive manner. The application is an integration of three classic games: Sudoku, Snake Game, and Minesweeper. Each of the games is implemented as an independent module, which exhibits varied DSA concepts. The proposed system has a modular design. The home screen is a central navigation tool that enables a user to choose any of the three games. Once a game is chosen, the user is redirected to the respective game module, where level selection, game processing, and completion handling are handled within the module. Each module contains game initialization, algorithm processing, user interaction handling, state checking, and navigation handling. Although the three games have a common interface design, their internal processing logic is different, which enables the demonstration of varied DSA concepts in a single integrated application.

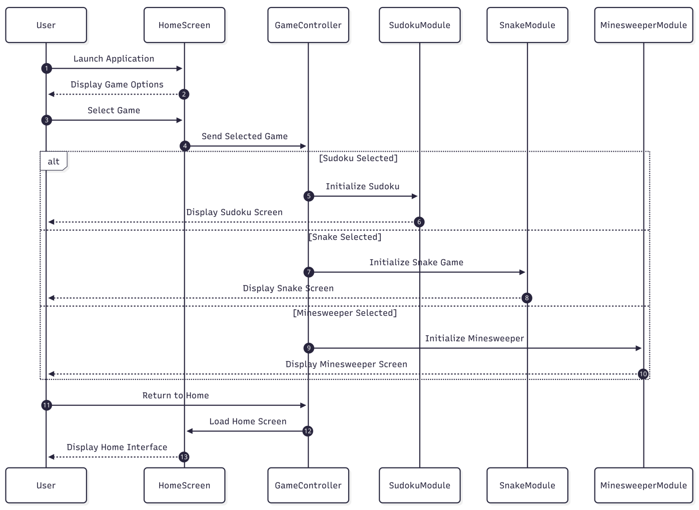

Figure 3.1: Navigation System Sequence Diagram

3.2 Sudoku

Sudoku is implemented as a constraint satisfaction problem using a 9×9 grid. The objective is to fill the grid so that every row, column, and 3×3 subgrid contains digits from 1 to 9 without repetition. The Sudoku board is represented using a two-dimensional array. This structure allows efficient traversal of rows and columns using indexed access. Each cell stores either zero (indicating an empty cell) or a valid digit between 1 and 9. The core algorithm used in Sudoku is Backtracking, implemented through Recursion. Backtracking systematically explores all possible number placements while abandoning invalid configurations as soon as constraints are violated.

The solving process works as follows: The algorithm scans the grid to find an empty cell. For that cell, it attempts to place numbers from 1 to 9 sequentially. Before inserting a number, constraint validation is performed by checking three conditions: the number does not already exist in the same row, the number does not already exist in the same column, and the number does not already exist in the corresponding 3×3 subgrid. If the number satisfies all constraints, it is temporarily placed in the cell, and the algorithm recursively attempts to solve the remaining grid. If at any stage no valid number can be placed, the algorithm backtracks by resetting the current cell and trying the next possible value. This recursive mechanism forms a decision tree where each node represents a possible board state. The algorithm performs a depth-first traversal of this state space. The time complexity, however, for the Sudoku Solver is $O(9^n)$, where n is the number of cells.

Additional DSA concepts applied in Sudoku include matrix traversal, where row-wise and column-wise iteration is performed repeatedly for validation; constraint checking and pruning, where invalid branches of the decision tree are eliminated early, significantly reducing computational complexity; and recursive stack utilization, where the system call stack implicitly stores intermediate board states during recursion. Level selection is implemented by loading predefined puzzle configurations with varying numbers of empty cells. Higher difficulty levels increase recursion depth and search complexity. Sudoku therefore demonstrates recursive algorithm design, constraint satisfaction, search-space exploration, and backtracking optimization.

3.3 Snake Game

The Snake game is implemented as a dynamic, real-time system in which the snake moves continuously within a grid environment. The primary technical challenge lies in managing the growing body of the snake efficiently while maintaining real-time responsiveness. The game board is represented using a two-dimensional grid structure; however, the snake body itself is implemented using a Singly Linked List. Each node of the linked list represents a segment of the snake and stores the X-coordinate, Y-coordinate, and a pointer to the next node. The head of the linked list represents the current position of the snake. During movement, a new node is inserted at the head to represent the new position. If the snake does not consume food, the tail node is removed. This mechanism follows the Queue (First-In-First-Out) principle, where insertion occurs at one end and deletion occurs at the opposite end.

The use of a linked list allows dynamic memory allocation and efficient resizing of the snake body without shifting elements, which would be required in an array implementation. This ensures constant time insertion and deletion operations. Collision detection is implemented by traversing the linked list and checking whether the head's coordinates match any other node in the body. This represents a linear

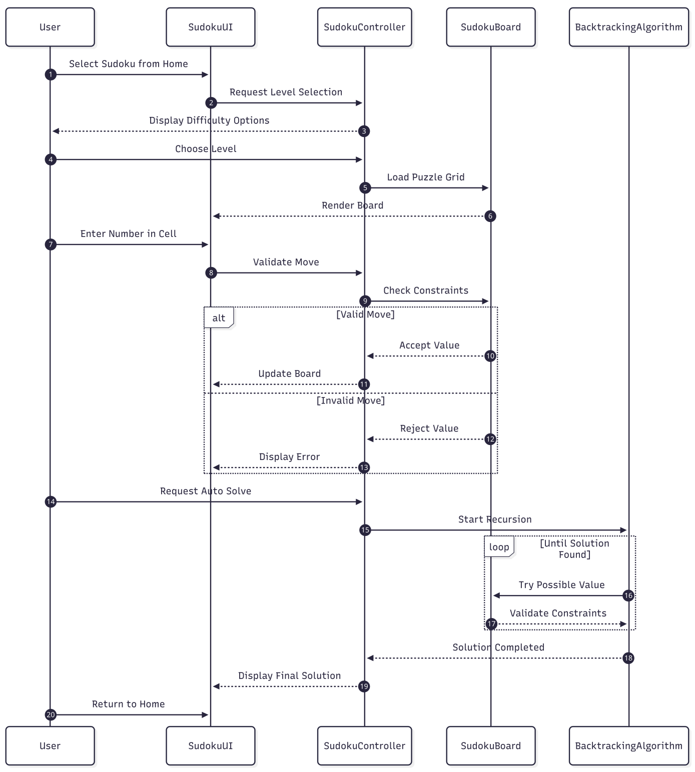

Figure 3.2: Sudoku Sequence Diagram

search operation with time complexity $O(n)$, where n is the snake length. Randomized algorithms are used for food generation, where a random coordinate is generated within grid boundaries and verified to ensure that it does not overlap with any segment of the snake body.

A Stack data structure is used to implement the Play and Pause mechanism. When the game is paused, the current state of the snake, including its position, direction, and score, is pushed onto the stack. When the game resumes, the most recent state is popped from the stack and restored. This follows the Last-In-First-Out (LIFO) principle and ensures that the most recent game state is preserved accurately. This implementation demonstrates linked list usage for dynamic structure management, queue-based movement logic, stack for state preservation, linear traversal for collision detection, dynamic memory allocation, and real-time event handling. Difficulty levels are implemented by modifying the snake speed. Higher levels require faster updates, indirectly testing algorithm efficiency and responsiveness. Thus, the Snake game demonstrates dynamic data structures, memory management, stack operations, and real-time algorithm execution.

3.4 Minesweeper

Minesweeper is implemented as a grid-based logic game where each cell may either contain a mine or indicate the number of adjacent mines. The board is represented

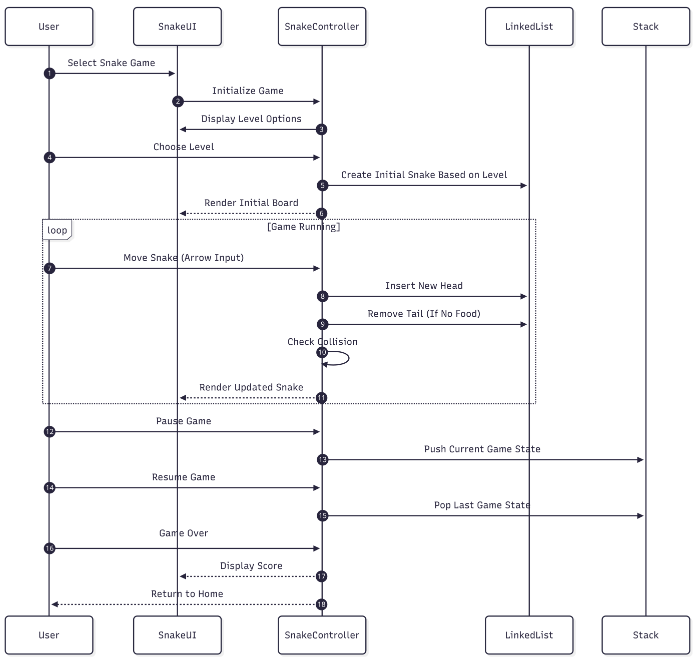

Figure 3.3: Snake Sequence Diagram

using a two-dimensional array of structured objects. Each cell stores a boolean indicating whether it contains a mine, a boolean indicating whether it has been revealed, and an integer representing the number of adjacent mines. Conceptually, the Minesweeper board can be modeled as a Graph, where each cell acts as a node and edges connect neighboring cells in eight possible directions.

The primary DSA concept used in Minesweeper is Breadth-First Search (BFS). When a user clicks a cell with zero adjacent mines, the game automatically reveals all connected safe cells. This expansion is implemented using BFS. The process begins by inserting the selected cell into a queue. The algorithm then repeatedly removes a cell from the queue, reveals it, and examines its neighbors. If neighboring cells are also safe and unrevealed, they are added to the queue. This continues until all connected safe cells are processed. BFS ensures systematic level-order expansion and prevents repeated processing through proper marking of visited cells. Alternatively, the same functionality can be implemented using Depth-First Search (DFS) through recursion. In this approach, recursive calls reveal connected cells, forming a depth-first traversal of the grid graph.

Mine placement is performed using a randomized algorithm that generates unique coordinate positions. Duplication is avoided by verifying whether a cell already contains a mine before placement. Adjacent mine calculation involves checking all eight neighboring cells for each grid cell. This demonstrates multidirectional matrix traversal. Minesweeper therefore illustrates graph modeling, BFS and DFS traversal algorithms, queue implementation, recursion, flood-fill techniques, and grid-based computations.

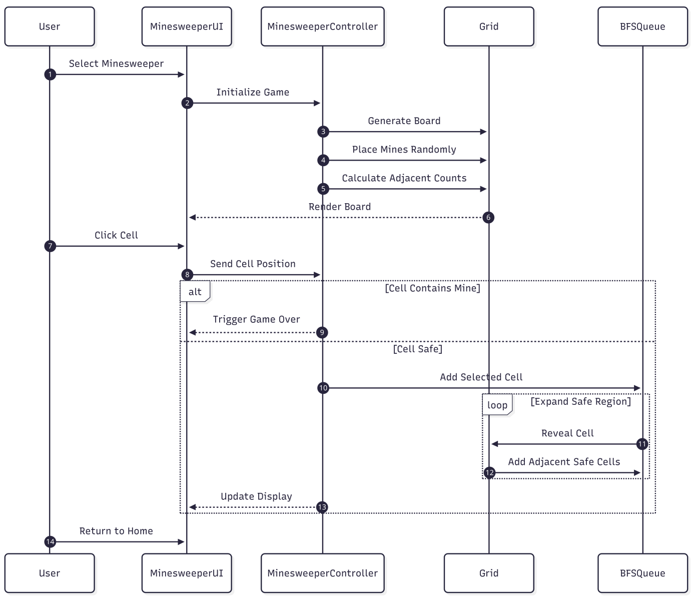

Figure 3.4: Minesweeper Sequence Diagram

Chapter 4: System Requirements

4.1 Hardware Requirements

4.1.1 Development Environment

To develop the Sudoku Solver, Snake Game, and Minesweeper using C++ and the Qt framework, the development system should have at least an Intel Core i3 or AMD Ryzen 3 processor, with Intel Core i5 or Ryzen 5 recommended for smoother compilation and debugging. A minimum of 8 GB RAM is required, while 16 GB is preferred for running Qt Creator, debugging tools, and multiple instances simultaneously. At least 10 GB of free storage is needed, with 20 GB SSD recommended for faster compilation and storage of project files, Qt libraries, and build artifacts. Integrated graphics are sufficient for 2D game rendering, though a dedicated GPU can improve UI responsiveness during development. The operating system should be Windows 10/11, macOS 10.15+, or Linux (Ubuntu 20.04+ recommended). A stable internet connection is required for downloading Qt libraries, updates, and additional dependencies.

4.1.2 User Devices

The games are designed to run on desktop and laptop computers with the Qt runtime installed. Devices should have at least 4 GB RAM and 200 MB of free storage for application installation and execution. For optional cross-platform deployment (Qt supports multiple platforms), Windows 10+, macOS 10.15+, or Linux systems are supported. Continuous internet connectivity is required only for downloading the application or receiving updates; gameplay is fully functional offline.

4.2 Software Requirements

4.2.1 Development Tools

The games will be developed using C++ as the primary programming language, chosen for its performance, memory management capabilities, and extensive support for object-oriented and generic programming. The Qt framework (version 6.2 or later) is used for GUI development, providing cross-platform widgets, signal-slot communication, and multimedia support. The IDE used is Qt Creator, which offers integrated tools for UI design, debugging, and project management. Git is employed for version control to manage code changes and facilitate collaboration among team members.

4.2.2 Frontend Tools

Qt Widgets:

Qt Widgets form the core of the graphical user interface for all three games. Custom widgets are developed to render the Sudoku grid (9×9 cell layout), the Snake game board (grid with dynamic snake representation), and the Minesweeper board (clickable cells with flagging and revealing functionality). The widget-based approach allows precise control over user interactions, including keyboard events for Snake movement and mouse clicks for Minesweeper and Sudoku.

Qt Designer:

Qt Designer is optionally used for prototyping and designing UI layouts, particularly for the home screen and level selection dialogs. It enables rapid iteration of interface components without manual coding of widget geometries.

Qt Multimedia:

The Qt Multimedia module provides sound effects and feedback mechanisms, enhancing user engagement through audio cues for events such as food consumption in Snake, cell reveals in Minesweeper, and puzzle completion in Sudoku.

Qt Core:

This module supplies essential non-GUI functionality, including event loops, timers for real-time game updates (Snake movement), and file I/O for saving and loading game states, high scores, and statistics.

4.2.3 Backend and Supporting Libraries

Standard Template Library (STL):

The STL is extensively used for data structure implementations. Vectors (‘std::vector’) manage dynamic arrays for game boards and collections. Algorithms such as ‘std::find’ and ‘std::generate’ support linear searches and random generation. The STL’s containers and algorithms complement the custom data structures central to each game.

Qt Core (Data Structures):

Beyond GUI, Qt Core provides ‘QList’, ‘QVector’, and ‘QStack’ which are used in the Snake game for stack-based state preservation (play/pause). The ‘QQueue’ class (based on ‘QList’) facilitates BFS implementations in Minesweeper. Qt’s implicit sharing and memory management simplify dynamic data handling.

Qt GUI (Rendering):

The Qt GUI module handles all 2D painting operations through ‘QPainter’. Custom paint events render the game boards, snake segments, mines, and numbers. Double-buffering ensures flicker-free animation, critical for smooth Snake gameplay.

4.3 Text Editor / IDE

Qt Creator is used as the primary integrated development environment for all three games. It provides strong C++ and Qt support, including code completion, syntax highlighting, integrated debugging with GDB or LLDB, and a visual form designer for UI layouts. The Qt Creator environment streamlines the build process through qmake integration and offers profiler tools for performance analysis, which is essential for optimizing real-time game loops and algorithm efficiency.

4.4 Functional and Non-Functional Requirements

4.4.1 Functional Requirements

Sudoku:

Players must be able to start a new game and select difficulty levels (Easy, Medium, Hard) that determine the number of pre-filled cells. The system validates player inputs against Sudoku constraints: rows, columns, and 3×3 subgrids must each contain digits 1–9 without repetition. Players may request an automatic solution using the recursive backtracking algorithm, with the solution displayed stepwise or in full. Puzzles can be reset, and progress can be saved to and loaded from persistent storage.

Snake Game:

Players initiate a new game and control the snake’s direction using arrow keys. Food appears at random grid positions not occupied by the snake; upon consumption, the snake grows by one segment via linked list insertion, and the score increments. Collision detection identifies when the snake hits the walls or its own body, ending the game. The game supports pause and resume, with the current state (snake body, direction, score) pushed onto a stack and restored later. High scores are stored in a text file.

Minesweeper:

Players select difficulty levels (Beginner: 9×9 with 10 mines, Intermediate: 16×16 with 40 mines, Expert: 16×30 with 99 mines). Left-clicks reveal cells; right-clicks flag potential mines. When a cell with zero adjacent mines is revealed, the flood-fill algorithm (BFS or DFS) automatically uncovers all connected safe cells. The game tracks remaining mine counts and detects win/loss conditions. Game states can be saved and reloaded.

4.4.2 Non-Functional Requirements

The games must respond instantly to user input, with no perceptible lag in UI rendering or game logic. Reliability is paramount; the application must not crash or freeze during normal use, and memory leaks must be avoided through proper resource management. The user interface must be intuitive and visually consistent across all three games, using a unified design language for buttons, dialogs, and game boards. Maintainability is ensured through modular code organization: each game resides in its own set of classes (‘SudokuBoard’, ‘Snake’, ‘Minesweeper’), with common base classes for shared functionality. The application is cross-platform, running on Windows, macOS, and Linux without modification. Loading times for game initialization and state restoration should be under 100 milliseconds. Algorithm implementations must handle worst-case scenarios efficiently; for example, Sudoku backtracking prunes invalid branches early, and Minesweeper BFS processes large empty regions without excessive recursion depth.

Chapter 5: Project Planning and Scheduling

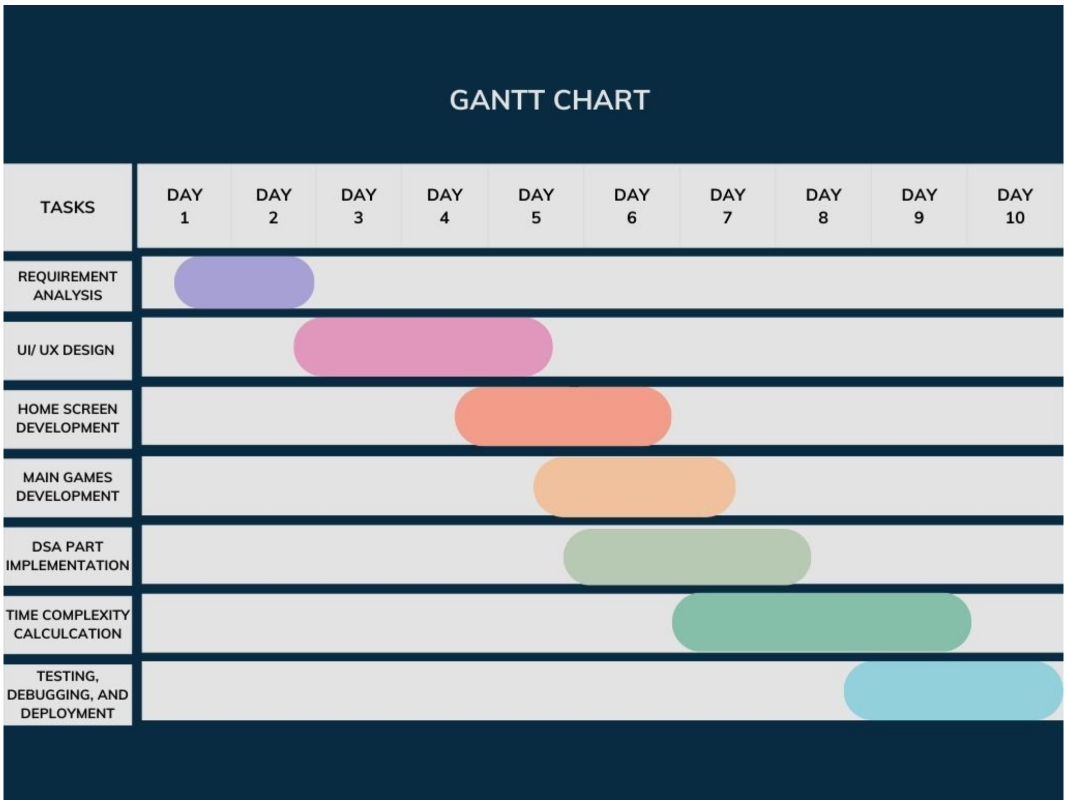

Figure 5.1: Project Timeline Gantt Chart

5.1 Project Timeline

The development of the DSA GameVault application was managed over a 12-days period, following an agile methodology with daily sprints and milestones. The timeline was divided into four major phases, with tasks distributed among the three team members.

5.1.1 Phase 1: Requirement Analysis and Design (Day 1-3)

- **Day 1:** Project initiation, requirement gathering, and literature review on DSA game implementations. Team discussions to finalize features and select appropriate algorithms for each game module.
- **Day 2:** System architecture design, module interface specifications, and creation of UML diagrams (Sequence Diagrams, Class Diagrams, Navigation Flowcharts).

- **Day 3:** Finalization of data structure selection for each module: backtracking with recursion for Sudoku, linked list for Snake, BFS/DFS for Minesweeper.
- **Responsible:** All members contributed to brainstorming and design. Sushant Timalsina and Prasiddha Timalsina focused on architectural diagrams and algorithm selection.

5.1.2 Phase 2: Core Module Development (Day 4-7)

- **Day 4-5:** Sudoku module development: 9×9 grid representation using 2D array, recursive backtracking implementation, constraint validation for rows, columns, and sub-grids.
- **Day 5-6:** Snake module development: Singly linked list implementation for snake body, queue-based movement logic, collision detection through linear traversal.
- **Day 6-7:** Minesweeper module development: Grid representation using 2D array of structured objects, BFS implementation for flood-fill expansion, DFS alternative implementation.
- **Responsible:** Sushant Timalsina (Minesweeper module), Prasiddha Timalsina (Sudoku module), Siddhant Timalsina (Snake module).

5.1.3 Phase 3: Integration and UI Development (Day 8-10)

- **Day 8:** Home screen development with Qt Widgets, implementing centralized navigation system for game selection.
- **Day 9:** Integration of all three game modules with the home screen, implementing level selection interfaces and difficulty adjustment mechanisms.
- **Day 10:** Stack-based play and pause mechanism implementation for Snake game, file I/O for high score persistence, UI polishing and consistency checks.
- **Responsible:** All members contributed to integration, with Siddhant Timalsina leading UI development.

5.1.4 Phase 4: Testing, Debugging, and Documentation (Day 11-12)

- **Day 11:** Unit testing for each algorithm implementation, integration testing for navigation flow, performance testing across difficulty levels.
- **Day 12:** Bug fixing, memory leak detection, code optimization, and final report compilation.
- **Responsible:** Testing was a team effort. Report compilation was shared, led by Sushant Timalsina.

5.1.5 Key Milestones

- **Milestone 1 (End of Day 3):** Completion of all system design diagrams and final algorithm selection.
- **Milestone 2 (End of Day 7):** Fully functional individual game modules with core algorithm implementations.
- **Milestone 3 (End of Day 10):** Complete working application with all three games integrated through home screen navigation.
- **Milestone 4 (End of Day 12):** Successful project demonstration and submission of final report.

Chapter 6: Discussion and Achievements

6.1 Project Overview

The development of this multi-game application provided a practical platform to implement and analyze fundamental Data Structures and Algorithms in a structured and meaningful way. The objective of the project was not limited to creating a game-based application, but to demonstrate how theoretical DSA concepts can be transformed into working computational systems. By integrating three different games—Sudoku, Snake, and Minesweeper, the project successfully covered multiple algorithmic paradigms while maintaining a modular and organized system architecture.

In the Sudoku module, the use of recursion and backtracking clearly demonstrated the concept of solving constraint satisfaction problems through systematic search. The algorithm explores possible number placements and eliminates invalid configurations through constraint validation in rows, columns, and sub-grids. This implementation highlighted how depth-first search operates within a decision tree structure and how pruning reduces unnecessary computations. Managing recursive calls and ensuring proper backtracking required careful validation logic, which strengthened understanding of recursion flow and stack behavior.

The Snake game introduced real-time dynamic data structure management. The implementation of a singly linked list allowed efficient insertion and deletion of snake body segments without excessive memory shifting. The movement logic reflected queue principles, while the play and pause mechanism demonstrated stack operations through state preservation and restoration. Collision detection required traversal of the linked list, emphasizing linear search complexity and performance considerations as the snake length increased. The integration of randomized food generation further enhanced logical validation and conditional processing.

In the Minesweeper module, the grid was conceptually modeled as a graph structure where each cell is connected to its neighboring cells. The implementation of Breadth-First Search enabled automatic expansion of connected safe regions, while the alternative Depth-First Search approach reinforced recursive traversal techniques. The use of queues to manage cell expansion and proper marking of visited cells ensured correct and efficient processing. Mine placement and adjacent count calculation required careful multidirectional traversal and boundary checking, strengthening understanding of grid-based algorithms.

Overall, the project demonstrated how selecting appropriate data structures directly impacts performance, clarity, and scalability. Each module required different computational strategies, reinforcing the importance of understanding when and where specific algorithms should be applied. The challenges faced during development improved debugging skills, logical thinking, and algorithm optimization capabilities.

6.2 Achievements

The primary achievement of this project lies in the successful implementation of multiple Data Structures and Algorithms within a single integrated application. The project effectively demonstrates recursive backtracking in Sudoku, linked list and queue-based movement in Snake, stack-based state management for play and pause functionality, and graph traversal algorithms such as BFS and DFS in Minesweeper. These implementations confirm the practical applicability of core DSA concepts in interactive systems.

Beyond algorithm implementation, the project strengthened understanding of time complexity, recursion depth handling, dynamic memory management, and traversal optimization. It provided hands-on experience in transforming theoretical knowledge into functional program logic. The modular system design further ensures scalability and maintainability, allowing future enhancements without restructuring the entire application.

From an academic perspective, the project bridges the gap between theoretical coursework and real-world implementation. It demonstrates structured problem solving, proper data structure selection, and systematic algorithm design. Successfully completing this mini project reflects both technical competency and analytical capability in applying Data Structures and Algorithms within a practical software development context.

Chapter 7: Conclusion and Recommendations

7.1 Conclusion

DSA GameVault is an efficient and user-friendly multi-game application developed to simplify the learning and demonstration of Data Structures and Algorithms. The application combines C++ for core logic implementation, Qt framework for cross-platform GUI development, and algorithmic strategies for game mechanics and state management. These technologies provide a seamless, interactive, and scalable experience for both novice and experienced learners seeking to understand fundamental DSA concepts.

The application allows users to engage with three classic games: Sudoku, Snake, and Minesweeper each demonstrating distinct algorithmic paradigms. In the Sudoku module, recursive backtracking with constraint satisfaction systematically explores possible number placements while eliminating invalid configurations through validation in rows, columns, and sub-grids. The Snake game implements a singly linked list for dynamic body management, queue principles for movement logic, and stack operations for state preservation during play and pause functionality. The Minesweeper module models the grid as a graph structure, implementing Breadth-First Search and Depth-First Search for automatic expansion of connected safe regions, demonstrating systematic traversal techniques and proper visited cell management. The project demonstrates the practical application of modern C++ programming techniques and algorithmic strategies to solve real-world educational challenges in computer science.

7.2 Recommendations and Future Enhancements

Despite its successful implementation, several improvements can further enhance the system's performance and user experience. Currently, the application demonstrates fundamental algorithmic implementations, but performance optimization for higher difficulty levels could be improved. For the Sudoku module, incorporating constraint propagation techniques such as naked pair elimination could reduce backtracking steps and improve solving efficiency for difficult puzzles, as demonstrated in recent researches. The heuristic-based approaches have shown significant speedup ratios compared to pure backtracking, particularly in expert-level puzzles.

The Snake game implementation currently uses linear traversal for collision detection with O(n) complexity. Future enhancements could implement spatial partitioning techniques to optimize collision detection for longer snake lengths. The Minesweeper module could benefit from optimized graph traversal algorithms for larger grid sizes and improved mine placement randomization.

Additionally, integrating persistent leaderboards and achievement systems would increase user engagement and provide motivation for continued learning. Further improvements in algorithm visualization, step-by-step execution tracing, and performance analytics would enhance usability and provide a smoother, more engaging educational experience for all users. The application could also be extended to demonstrate additional DSA concepts such as tree traversals, hashing techniques, and dynamic programming through new game modules.

References

[1] Tugce Ates and Fatih Cavdur. Sudoku puzzle generation using mathematical programming and heuristics: Puzzle construction and game development. *Expert Systems with Applications*, 2025. In press, available online.

[2] Apekshya Bhattarai, Dinisha Uprety, Pooja Pathak, Safal Narshing Shrestha, Salina Narkarmi, and Sanjog Sigdel. A study of sudoku solving algorithms: Backtracking and heuristic, 2025. Comparative analysis from Kathmandu University.

[3] Philip Bille, Martín Farach-Colton, Inge Li Gørtz, and Ivor van der Hoog. Snake in optimal space and time. In *12th International Conference on Fun with Algorithms (FUN 2024)*, volume 291 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. Data structure techniques for optimal Snake game representation.

[4] Kaiqi Chen. Exploring a better way to constraint propagation using naked pair. *ITM Web of Conferences*, 70:04025, 2025. Enhanced backtracking with constraint propagation for Sudoku.

[5] Lama Hamandi, Hla Htoo, Senay Tilahun, and Haider Amin. Demo 4b: Gamification of computer science algorithms. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education (SIGCSE TS 2025)*, 2025. Interactive 3D environment for teaching heap, heapsort, and spanning tree algorithms.

[6] Chuan Jin Lee. Minesweeper: Full-stack game implementation with graph traversal. `https://github.com/leechuanxin/p3-minesweeper`, 2021. Implementation demonstrating DFS and recursion for flood-fill mechanics.

Appendix

A.1 Source Code Structure

The source code follows a modular structure with clear separation of concerns. The main application is built using C++ with the Qt framework following this directory structure: The project root contains all source files organized into game-specific modules including sudokuboard.cpp/h, sudokucontroller.cpp/h, sudokusolver.cpp/h for the Sudoku module; snake.cpp/h, gamescreen.cpp/h for the Snake module; and minesweeper.cpp/h for the Minesweeper module. Navigation to the home screen is implemented through home.cpp/h, homescreen.cpp / h, and mainHomeScreen.cpp / h. Configuration files include MySnakeGame.pro for the Qt project configuration and .qtc-clangd for IDE integration.

A.2 Additional Diagrams

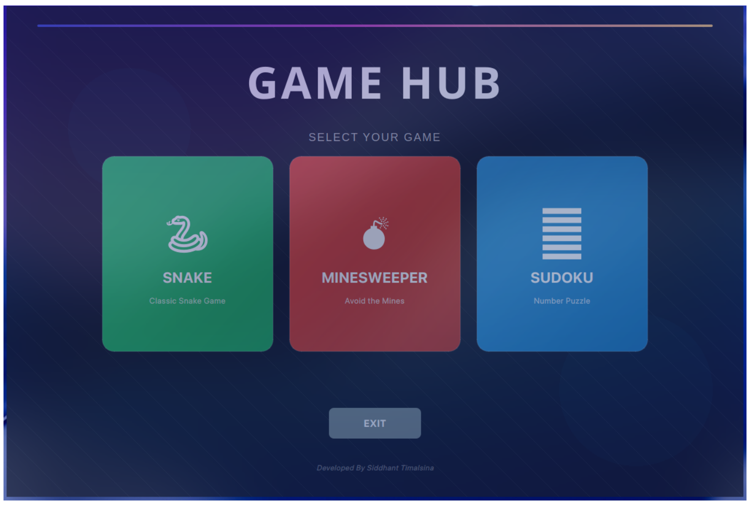

Figure 7.1: Game Home Screen

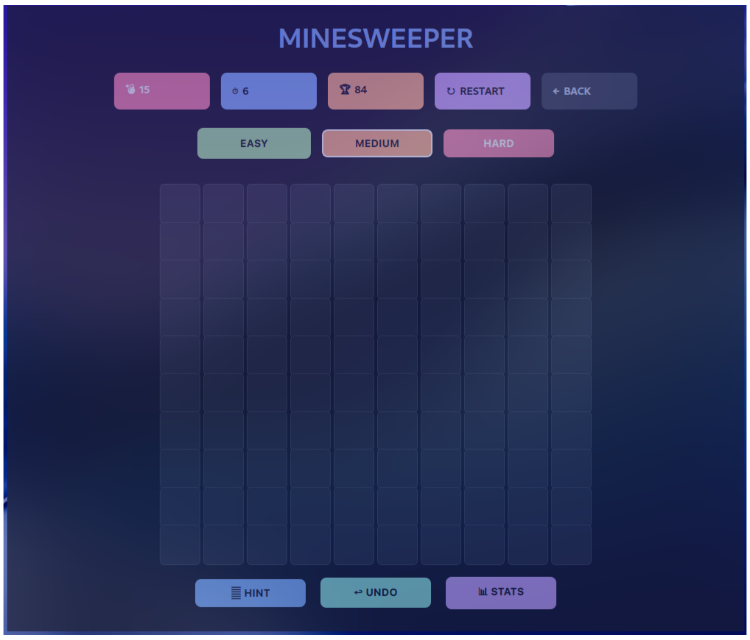

Figure 7.2: Minesweeper Game Screen

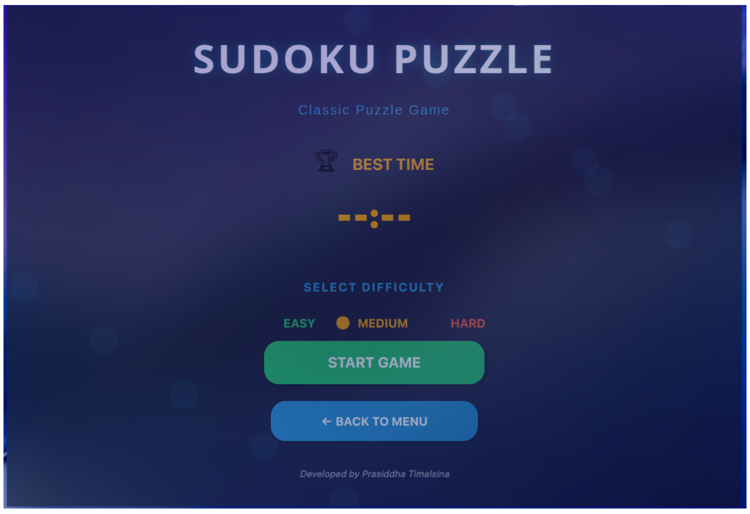

Figure 7.3: Sudoku Home Screen

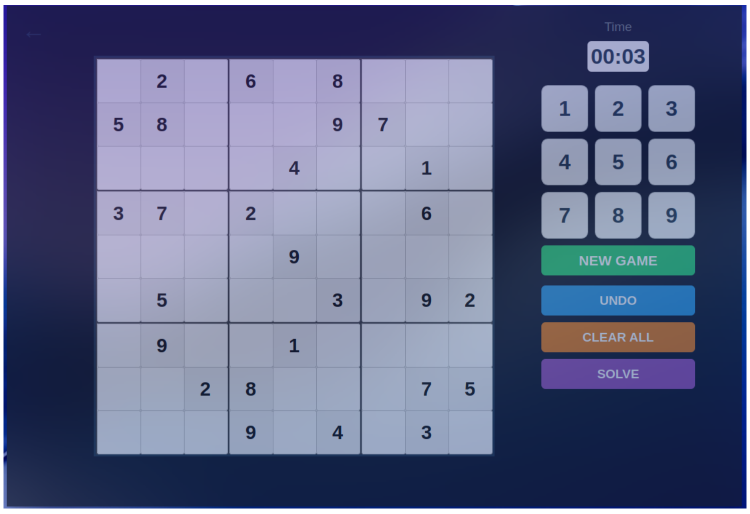

Figure 7.4: Sudoku Game Screen

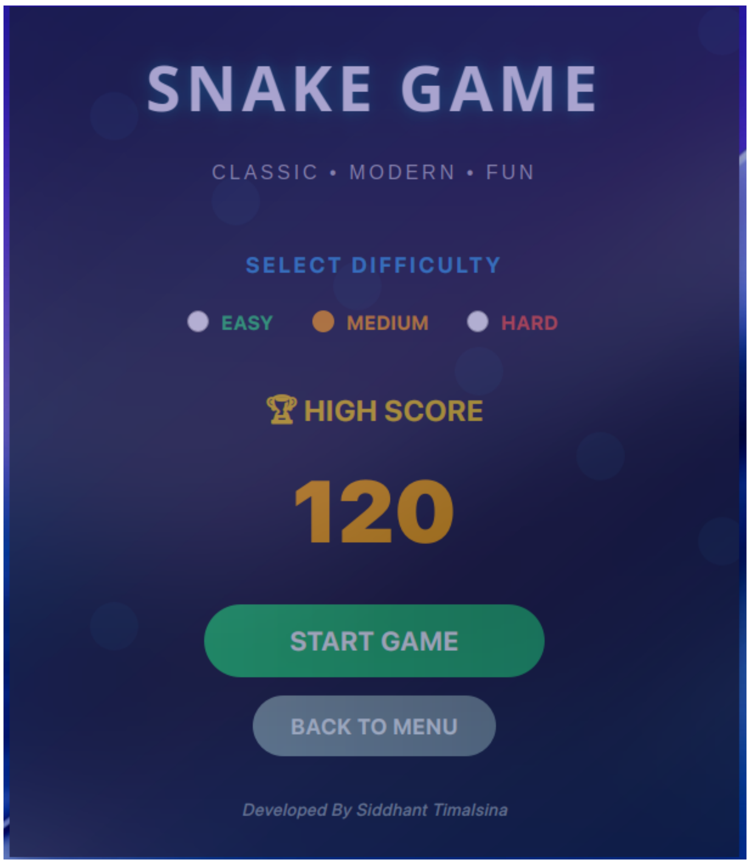

Figure 7.5: Snake Home Screen

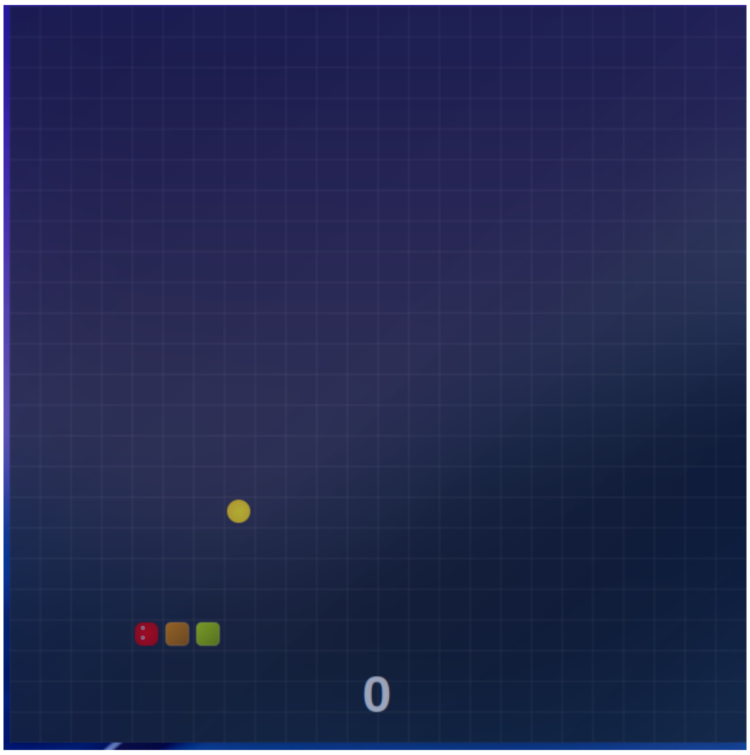

Figure 7.6: Snake Game Screen

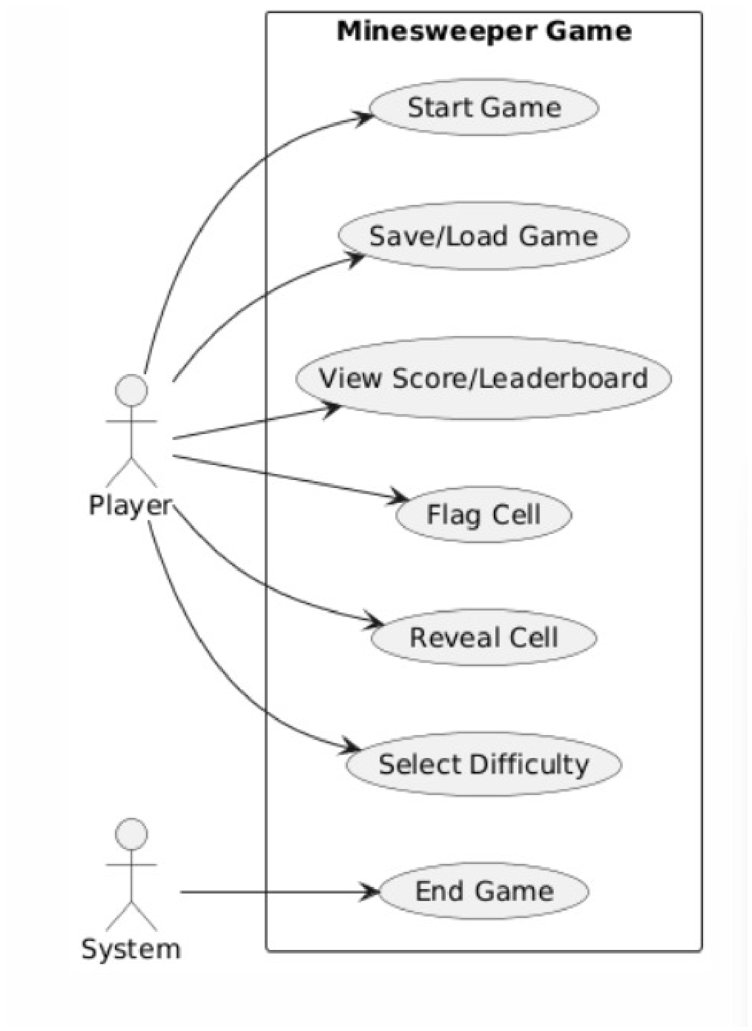

Figure 7.7: Use Case Diagram: Minesweeper

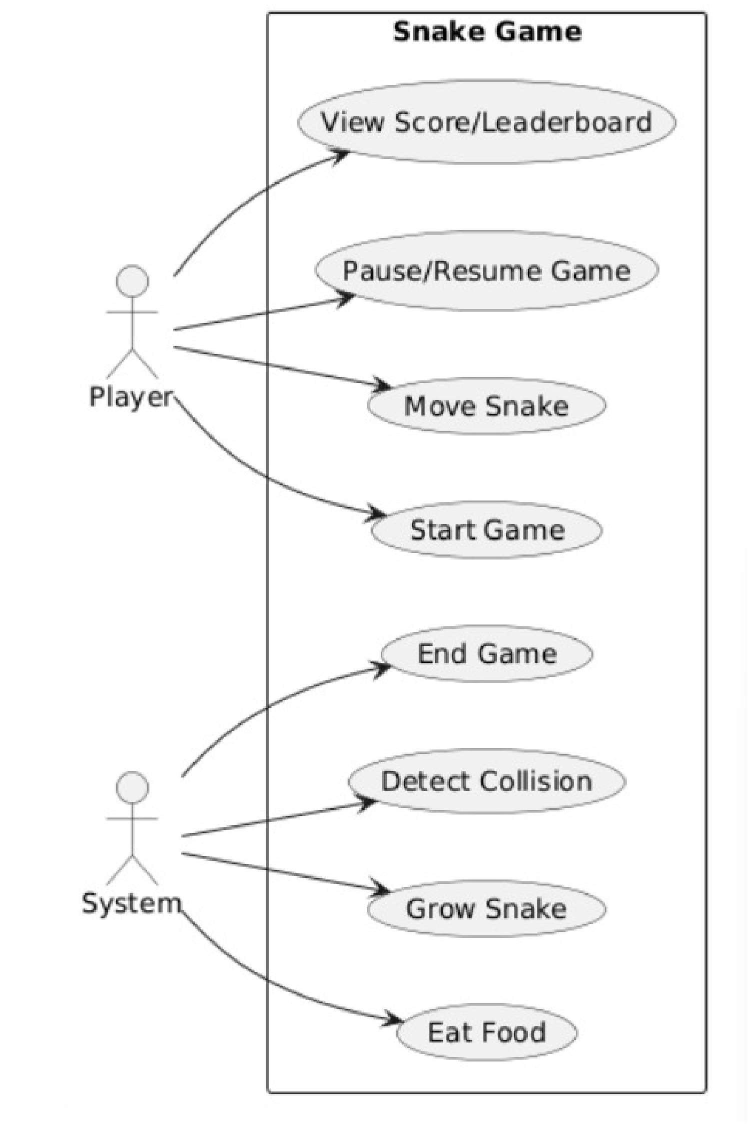

Figure 7.8: Use Case Diagram: Snake

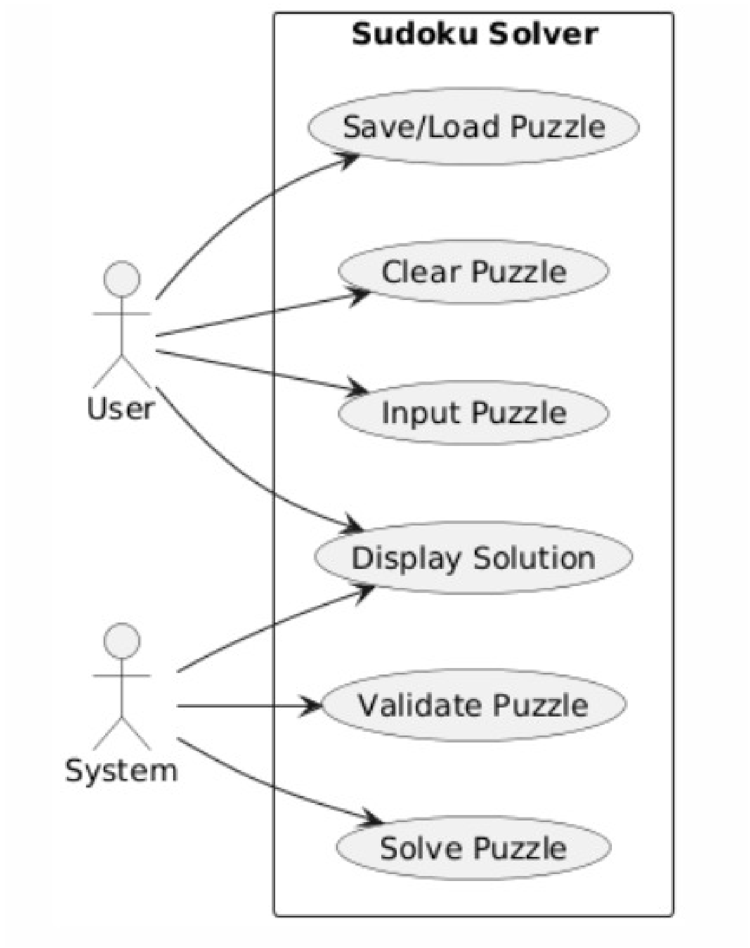

Figure 7.9: Use Case Diagram: Sudoku Solver