# ISTANBUL TECHNICAL UNIVERSITY

# COMPUTER ENGINEERING DEPARTMENT

## BLG 223E

## DATA STRUCTURES
## HOMEWORK REPORT

**HOMEWORK NO** : 2

**HOMEWORK DATE** : 21.07.2025

820220336 : UYGAR GÖK

**SUMMER 2025**

# Contents

# 1 INTRODUCTION

The Peg Solitaire, commonly known as the "Solo Test", is a classic single-player puzzle game played on a cross-shaped board with 33 valid positions. The objective of the game is to remove pegs by jumping one over another until only one remains in the center. This project presents a complete implementation of the Peg Solitaire game in the C programming language, utilizing a dynamic data structure based on linked nodes to represent the board and allow for flexible move validation and simulation.

In this implementation, each playable position on the board is modeled using a Node structure, which maintains pointers to its adjacent positions (left, right, up, and down). The board is represented as an array of such nodes, facilitating efficient traversal and move application. Two main functionalities are developed: a player mode, where the user can interactively make moves by specifying the from/over/to indices; and a solver mode, where the computer autonomously attempts to solve the puzzle using a depth-first search (DFS) approach implemented with a stack structure.

Compared to the initial version, improvements have been made for code clarity and efficiency. A generalized movement check using directional vectors replaced repetitive conditionals, memory deallocation functions were added to prevent leaks, and the structure of the solver loop was optimized for maintainability. This project demonstrates fundamental principles of data structures, memory management, and algorithmic problem-solving in C.

# 2 METHODS

In this project, the Peg Solitaire board is implemented using a dynamic data structure based on linked nodes. Each valid position on the 7 * 7 cross-shaped board is represented by a Node structure containing pointers to its adjacent nodes: left, right, up, and down. These nodes are stored in a Board structure, which holds all 33 playable positions. The board is initialized using the init board function, where the center node is set to empty ('E') and all others are filled ('F'). Connections between nodes are assigned by mapping valid grid coordinates to linear indices.

The program operates in two modes: Play mode and Solver mode. In play mode, the user interacts with the board by entering three indices corresponding to a move: FROM, OVER, and TO. In solver mode, the program uses a depth-first search (DFS) approach

implemented via a custom stack structure. At each step, it copies the current board using the copy board function, applies all possible valid moves in four directions, and pushes resulting states onto the stack. Moves are validated using the try move function and calculated dynamically using a helper function called get neighbor index, which maps directional offsets to index changes. All dynamically allocated memory is properly deallocated with free board to prevent memory leaks.

# 3 RESULTS

The implemented Peg Solitaire solver successfully explores possible move sequences to reduce the number of pegs remaining on the board. In solver mode, the program prints each iteration, showing how many pegs are left at that stage, and terminates when a board state with only one peg remains. During testing, the program consistently reached the optimal solution with a single peg remaining at the center, demonstrating the effectiveness of the depth-first search strategy.

The console output provides a step-by-step visualization of the solver's progress, along with the final board state when the solution is found. Additionally, in play mode, users were able to manually enter moves and interact with the board using the indexed positions. This allowed for both automated solving and manual experimentation, making the program flexible for both analysis and gameplay purposes.

# 4 CONCLUSION [10 points]

In this project, we developed a Peg Solitaire game in C using linked list structures and implemented both a solver and interactive play mode. Throughout the process, we learned how to manage dynamic memory, work with pointers, and apply algorithmic thinking using depth-first search. Some parts—especially copying board states and managing neighbor relationships—were challenging, but helped deepen our understanding of data structures and problem-solving in C. Overall, the project was a valuable hands-on experience in both game logic and low-level programming.

# REFERENCES