Connect 4

Course Project Report

Group Name: 3 Bytes

Team Members:

Anji Abdallah

Leen Dheini

Waed Atwi

Course: CMPS 241 Systems Programming

Semester: Fall 2025 / 26

Instructor: Dr Mohamad Zalghout

Date: November 23 2025

1. Project Overview

The goal of this project was to develop a console based Connect 4 game with multiple levels of bot difficulty. The project was divided into four sprints:

- Sprint 1: Two-player game
- Sprint 2: Easy-level bot
- Sprint 3: Medium-level bot
- Sprint 4: Hard-level bot

The final sprint implemented a highly advanced bot capable of near-perfect gameplay using minimax, iterative deepening, transposition tables, and opening tables.

2. Game Description

Connect 4 is a turn-based strategy game played on a 7×6 board. Two players take turns placing checkers, aiming to connect four in a row horizontally, vertically, or diagonally.

Our project implemented:

- Full board display with colored pieces for players (‘A‘ in red, ‘B‘ in blue).
- Move validation, win/draw detection, and console-based interaction.
- Bot difficulty selection (Easy, Medium, Hard).

3. Code Structure

- `main.c`: handles game loop, input, mode selection.
- `board.c / board.h`: board management and scoring logic.
- `utils.c / utils.h`: helper functions.
- `bot_easy.c / bot_easy.h`: easy bot implementation.
- `bot_medium.c / bot_medium.h`: medium bot implementation.
- `minimax.c / minimax.h`: minimax algorithm implementation.
- `hashing.c / hashing.h`: Zobrist hashing for transposition table.

- `evaluation.c / evaluation.h`: Alpha Beta Pruning evaluation function.
- `transposition.c / transposition.h`: transposition table implementation.
- `makefile`: build automation.

4. Sprint-wise Implementation

4.1 Sprint 1 – Two-Player Game

- Implemented a console-based two-player game.
- Players alternate turns using `alternatePlayers()` and select columns using `getColumnFromUser()`.
- The board was printed after each move using `printBoard()`.
- Win and draw conditions were implemented using `checkWin()` and `checkDraw()`.

4.2 Sprint 2 – Easy Bot

- Added a bot that makes random valid moves.
- Ensured the bot never picks invalid columns.
- Outputted basic feedback to the console about bot moves.
- Complexity: $O(1)$ per move, since the bot selects a random column.

4.3 Sprint 3 – Medium Bot

1. Attempt to win immediately.
    - The bot loops through all columns and checks if dropping a piece there would achieve a valid winning move through `checkWin()`.
    - $O(R \times C) = O(42) = O(1)$
2. Block opponent's immediate win.
    - Used `checkOpponentWin()` to get the opponent's winning move and place the bot's piece there.
    - $O(R \times C) = O(42) = O(1)$

3. Prefer central columns for strategic advantage.
    - $O(R \times C^2) = O(294) = O(1)$
4. Fallback to random moves if no other option.
    - $O(C) = O(1)$

Complexity: $O(C^2 \times R) = O(7^2 \times 6) = O(294) = O(1)$

4.4 Sprint 4 – Hard Bot

Core Features

1. **Minimax Algorithm with Alpha-Beta Pruning**
    - Searches the game tree to a fixed depth (depth = 11).
    - Alpha-beta reduces unnecessary branches, improving efficiency.
2. **Iterative Deepening**
    - Depth $1 \to 2 \to \ldots \to 11$, populating the transposition table at each depth.
    - Allows the bot to always have a strong move even under time constraints.
3. **Transposition Table with Zobrist Hashing**
    - Stores previously evaluated positions to avoid redundant calculations.
    - Supports fast lookups and stores the best move for each position.
4. **Opening Table**
    - Guarantees optimal first 1–3 moves, ensuring strong early-game play.
5. **Evaluation Function**
   Scores positions based on:
    - 4-in-a-row and 3-in-a-row patterns
    - Centre column preference
    - Immediate threats (win/block)
    - Potential three-in-a-row setups
6. **Threat-Driven Move Ordering**

- Moves that win immediately or block opponent are evaluated first.
- Fork creation and blocking prioritized to reduce risk of traps.

7. **Null-Move Pruning**
   - Skips moves in certain positions to speed up evaluation while maintaining accuracy.

Complexity analysis

- Worst-case complexity per move: $O(C^D)$.
  - C = columns, D = depth.
  - $O(7^{11}) = O(1)$
- In practice, alpha-beta pruning, move ordering, iterative deepening, and the transposition table reduce the effective branching factor significantly.

Performance

- Against humans: essentially unbeatable.
- Against other bots: highly likely to force draw or win.
- Efficient and fast due to iterative deepening and TT optimizations.

5. Testing Strategy

Unit Testing

- Tested board operations: add/remove pieces, check win/draw.
- Tested evaluation function against known board patterns.
- Verified move ordering and threat detection logic.

Integration Testing

- Played against one another for Human vs. Human option.
- Played games between Easy, Medium, and Hard bots.
- Used an online bot as Hard bot's opponent.

- Tested edge cases by trying invalid input.
- Verified iterative deepening and TT behaviour.
- Checked console output correctness for each move.

Debugging Tools

- GDB: used to step through minimax and threat detection.

  Command: `make debug` then `gdb ./my_program`

- Valgrind: ensured no memory leaks.

  Heap summary:

  - In use at exit: 0 bytes.
  - All heap blocks freed.
  - 0 errors.

6. Improvements Across Sprints

Sprint	Improvement
1	Two-player game with full board and win detection.
2	Added easy bot (random valid moves).
3	Added medium bot: blocks opponent, prioritizes centre, tactical moves.
4	Hard bot: minimax, alpha-beta, iterative deepening, TT, opening table, threat-aware evaluation.

7. Tournament Readiness

- Hard bot is ready for knockout tournament.
- Features like opening table and iterative deepening make it fast, tactical, and practically unbeatable.

8. Limitations

- Hard bot sometimes slows down at depth 11 if TT is cold.
- Opening table covers only first 2–3 plies.
- Console interface limits visualization.
- Not fully multi-threaded.

9. Alpine Setup

1. Install and update Alpine:
2. `apk update`
3. `apk add build-base`
4. Clone the GitHub repository:
5. `git clone <repo-url>`
6. Set tty1 to auto-update your project by editing `/etc/inittab`.

10. Network Programming (Player vs Player)

Architecture: Client-server connection over 127.0.0.1 using TCP.

Server Responsibilities:

- Wait for a client connection.
- Maintain game board.
- Receive moves and update board.
- Send updated board back.

Client Responsibilities:

- Connect to server.
- Send moves.
- Receive and display updated board.

Typical functions used:

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `send()`
- `recv()`

11. Conclusion

- Easy bot → medium bot → hard bot.
- Hard bot incorporates iterative deepening, transposition tables, opening tables, threat-aware evaluation.
- Depth 11 makes it dominate human players and strong bots.

Future Enhancements:

- Dynamic depth adjustment.
- Multithreading.
- Full-width endgame solver.

12. References

- Science Buddies. (2024). *Simple Explanation of the Minimax Algorithm with Alpha-Beta Pruning with Connect 4* https://www.youtube.com/watch?v=DV5d31z1xTI
- Koushik Chandra Maji. (2024). *AI: Minimax AlphaBeta Pruning Connect4 Game in Java.* https://medium.com/@koushiknsec34/ai-minimaxalphabeta-pruning-connect4-game-in
- Part 4 – Alpha-beta algorithm. (2017). http://blog.gamesolver.org/solving-connect-four/04-alphabeta/
- Sawwan, A. (2021). *Artificial Intelligence-Based Connect 4 Player Using Python.* https://cis.temple.edu/~pwang/5603-AI/Project/2021S/Sawwan/AI%20Project.pdf
- Aberent. *Transposition Table and Zobrist Hashing.* https://adamberent.com/transposition-tabl
- GeeksforGeeks. (2023). *Minimax Algorithm in Game Theory — Set 5 (Zobrist Hashing).* https://www.geeksforgeeks.org/dsa/minimax-algorithm-in-game-theory-set-5-zobri