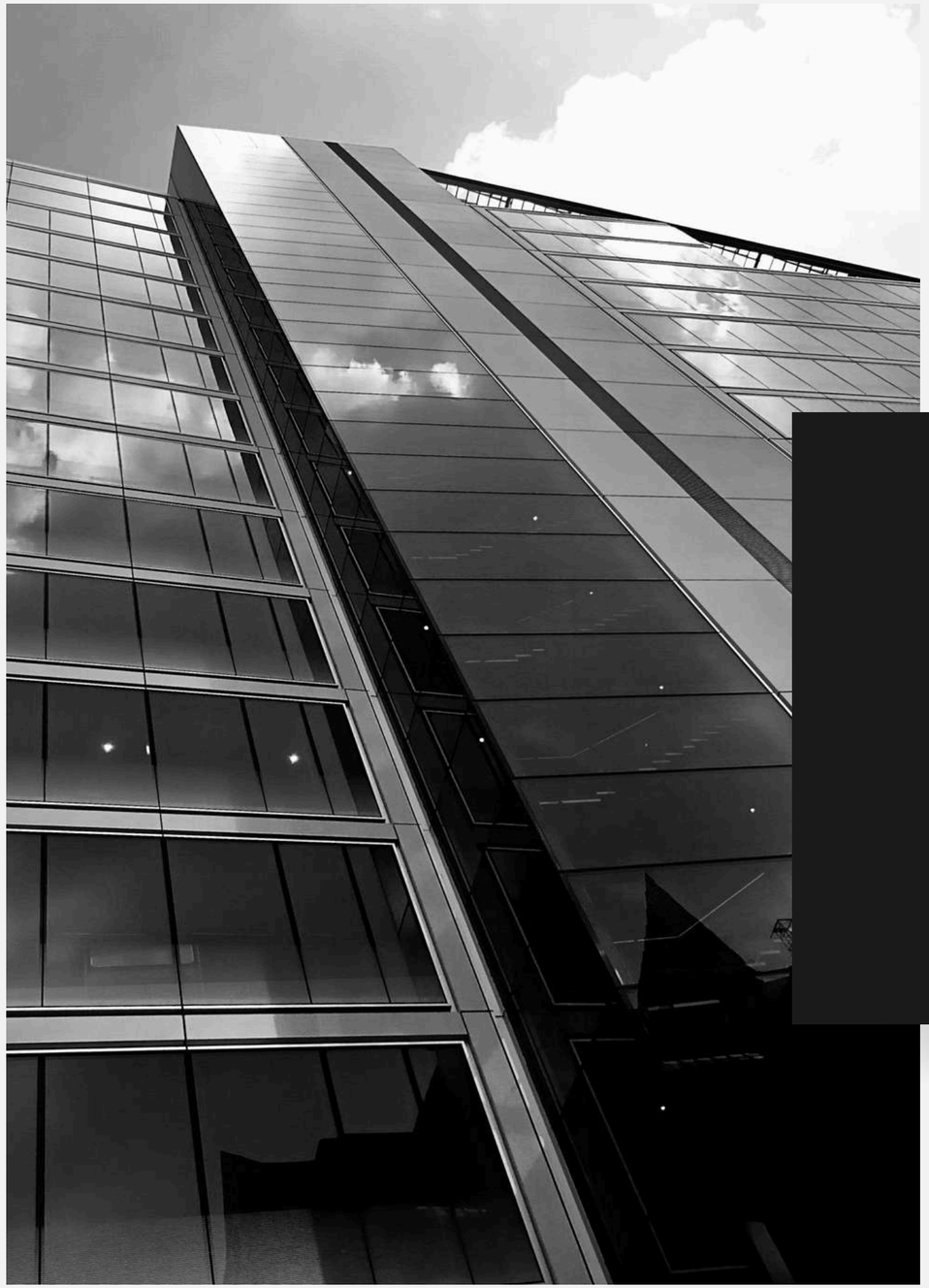
# DATA STRUCTURE ALGORITHM

Implementation of Classical AI Algorithms in a Console-Based Connect 4 Game

# IMPLEMENTATION OF CLASSICAL AI ALGORITHMS IN A CONSOLE-BASED CONNECT 4 GAME

By Group 4:
Agha Aryo Utomo / 5026241003
Dyandra R-Noor Batari / 5026241051
Maria Stephanie Febryana Kristijanto / 5026241052

# CONTENT



# INTRODUCTION

We set out to develop a **game of connect 4** that is **playable in console** and features an **intelligent opponent**. To achieve this we used search algorithms such as **linear search** and **BFS**, **minimax**, and **merge sort** to prune our minimax and prediction tree

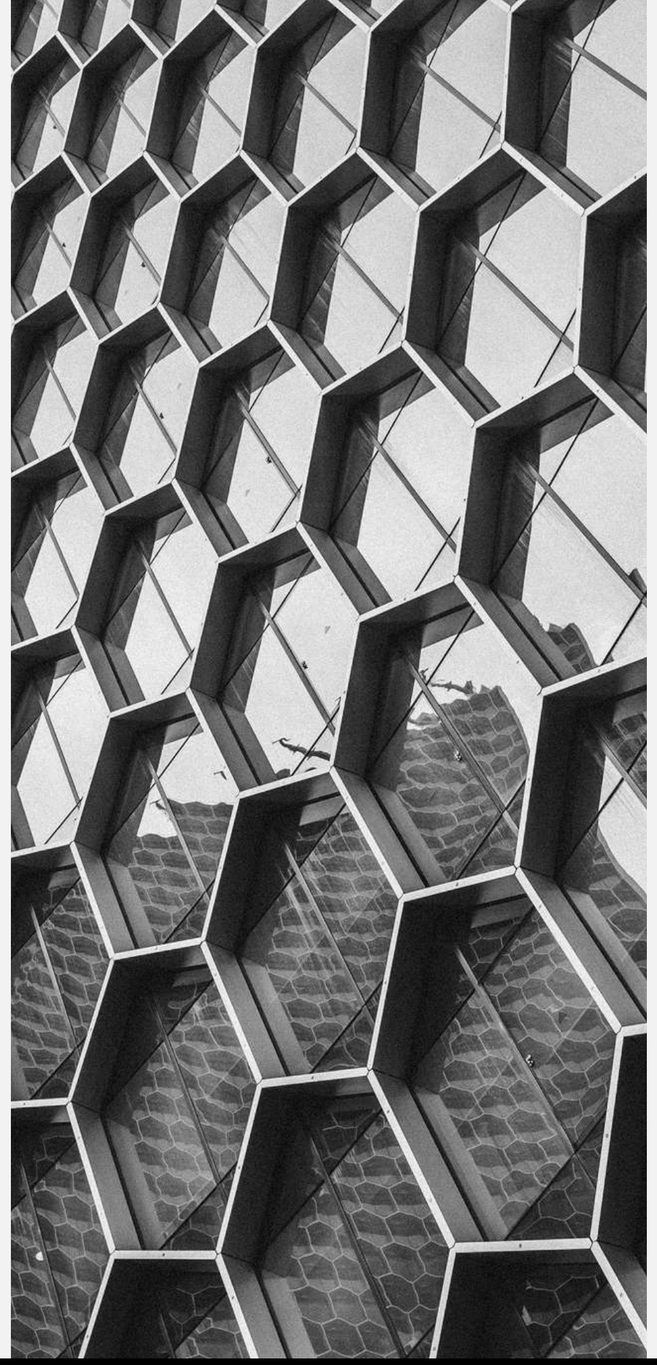
# BACKGROUND

**Artificial Intelligence (AI)** is used to improve the gameplay experience in strategy games like Connect 4. Games such as this require the ability to analyze board states, detect winning conditions, and make predictions about optimal moves.

Connect 4 is a simple yet strategic game, making it an ideal candidate for applying fundamental computer science algorithms, particularly those focused on searching, sorting, and decision-making.

The algorithms implemented in this project include:

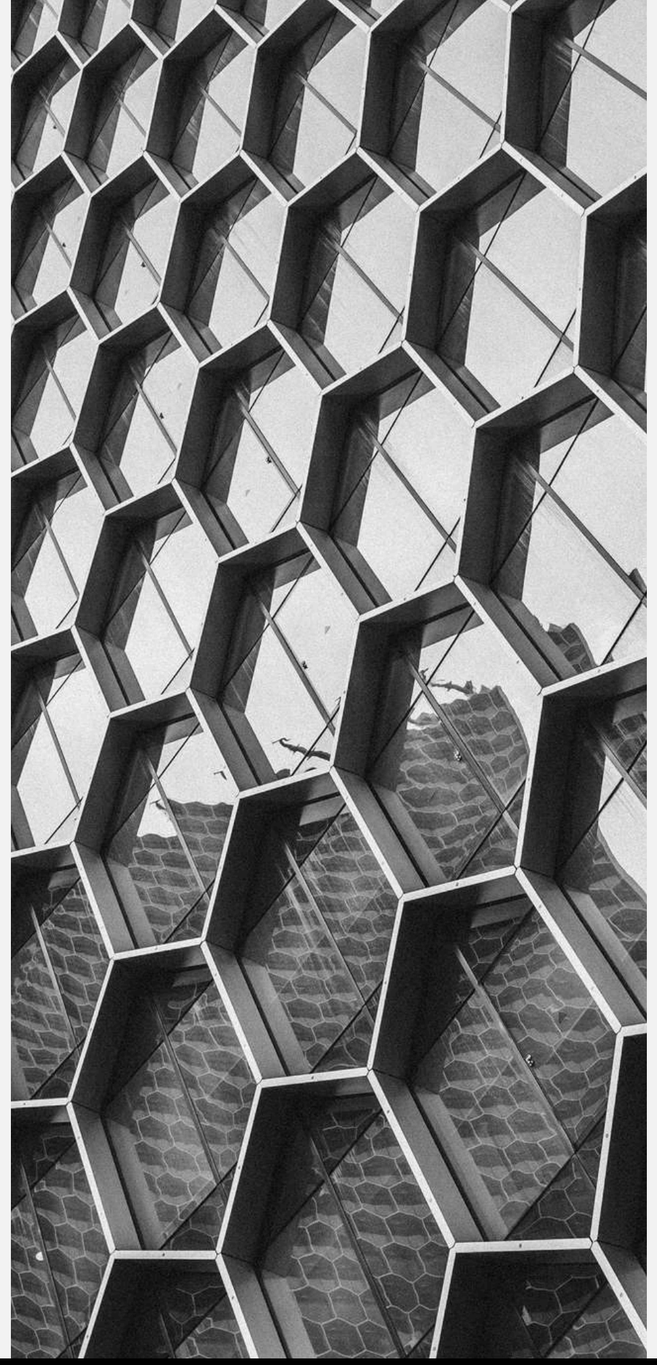
- Graph-based algorithms for detecting winning connections.
- Sorting algorithms to rank potential moves by their strength.
- Search algorithms to find valid moves and detect threats.-
- Tree-based decision-making through the Minimax algorithm to predict the optimal moves.



# GOALS

**Project Goals:**

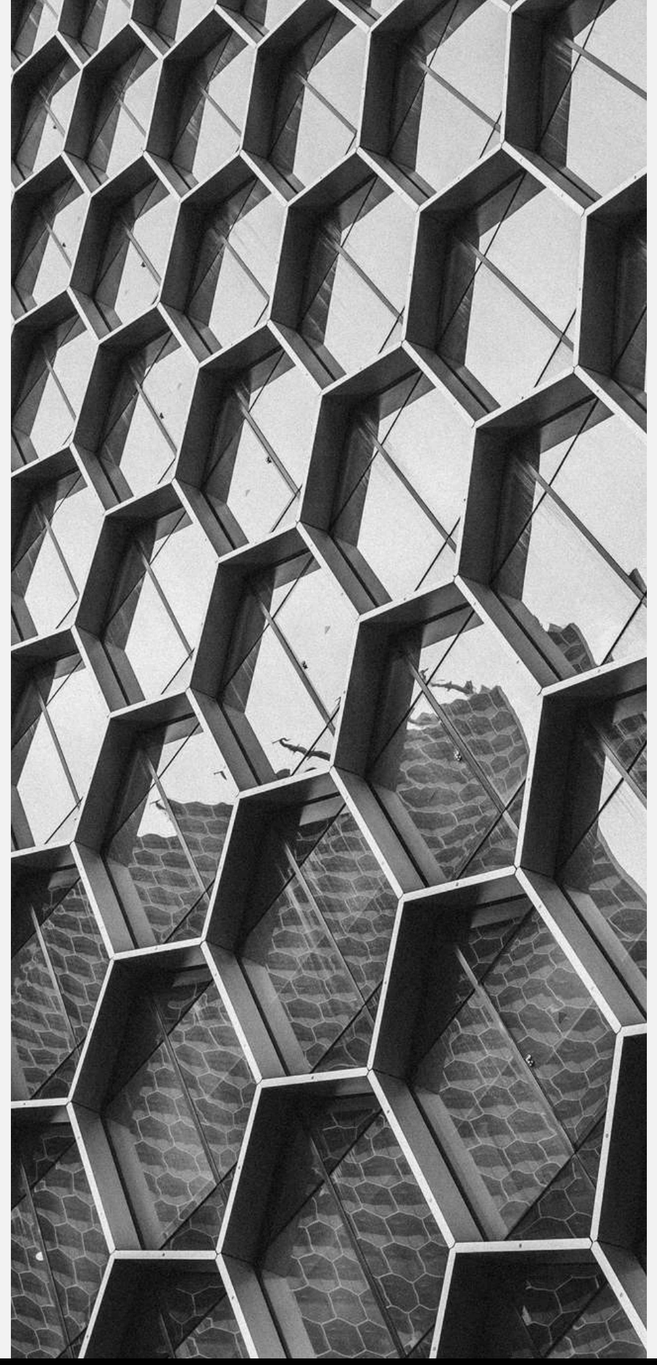
- **Design and Implement an Intelligent AI Opponent**
- **Implement Classical AI Algorithms**
- **Create a Console-Based Game**



# GAME FEATURES & FLOW

**Features**:

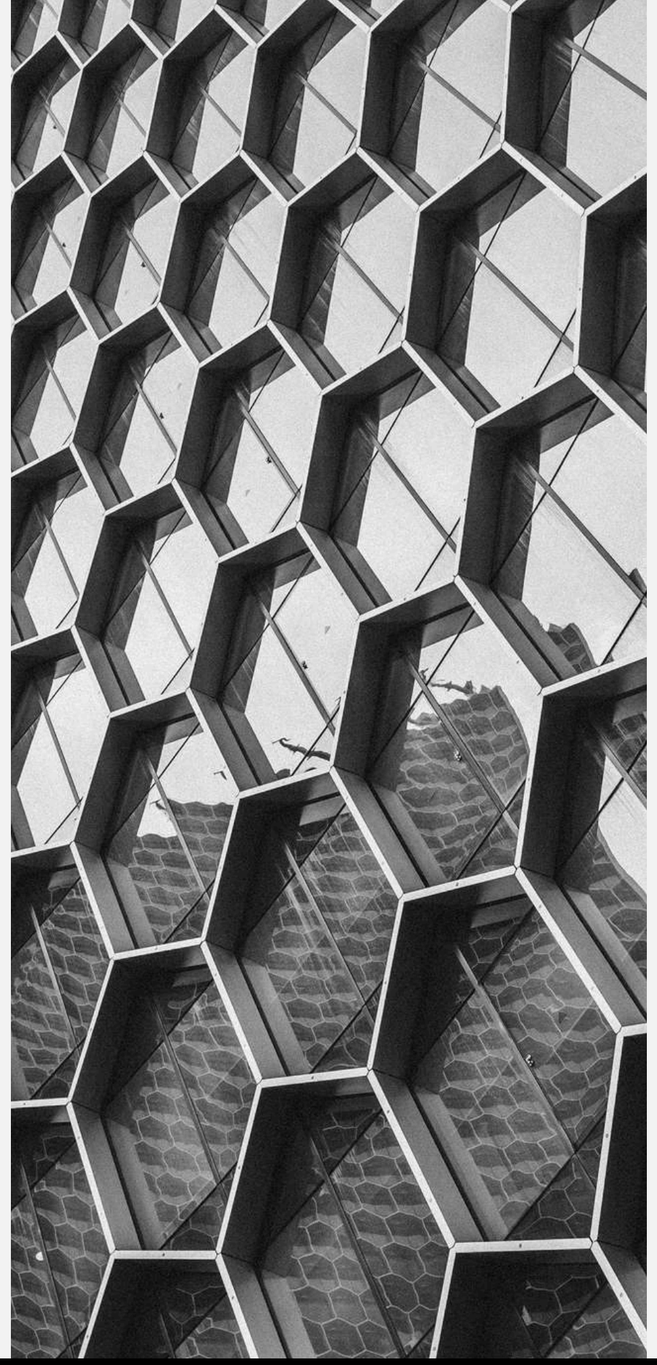
- **Multiple Difficulty Levels for AI**:
  - Easy
  - Medium
  - Hard
- **Interactive Console UI**: The game prompts the user for input (choosing columns) and outputs the game board after each move.
- **Game End Conditions**: The game checks for a win (four discs connected) after each move.



# GAME FEATURE & FLOW

**Game Flow:**

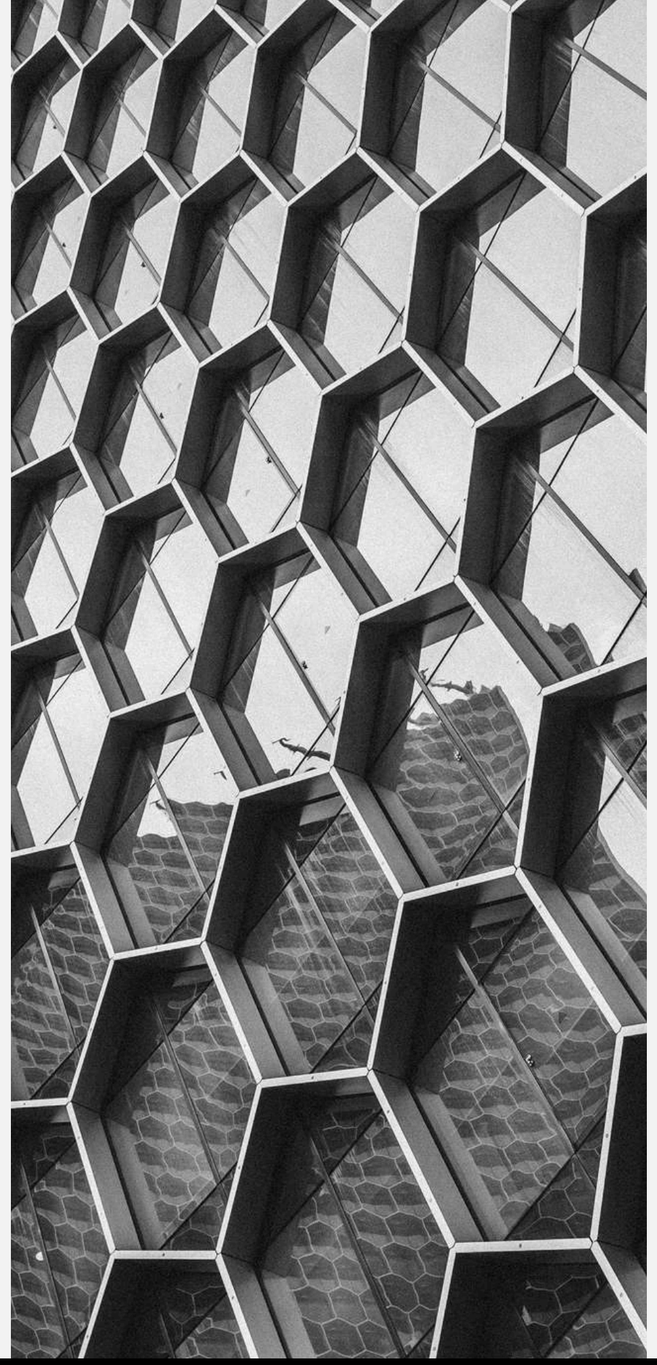
1. **Start Game:**
   - The user selects the difficulty level for the AI.
   - The board is initialized, and the game starts with the first player (human).
2. **Player Turn:**
   - The player selects a column (0–6) to drop their disc (represented by 'O').
   - The AI then takes its turn according to the chosen difficulty.
3. **AI Turn:**
   - The AI chooses a column based on the selected difficulty (random, scored, or optimal using Minimax).
4. **Win/Draw Check:**
   - After every move, the game checks for a winner using the BFS algorithm for detecting connected discs.
   - If no winner is found and the board is full, the game ends in a draw.



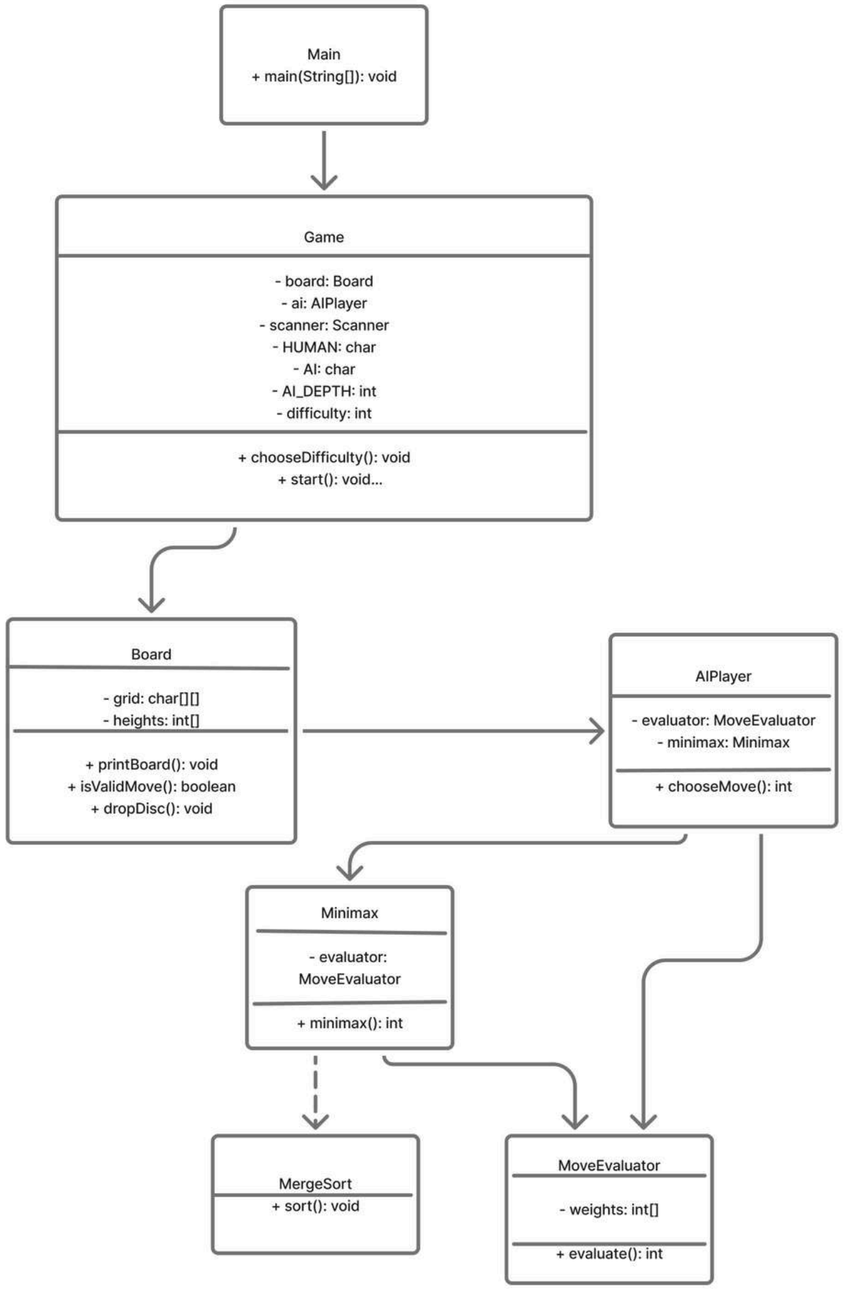
# SYSTEM ARCHITECTURE

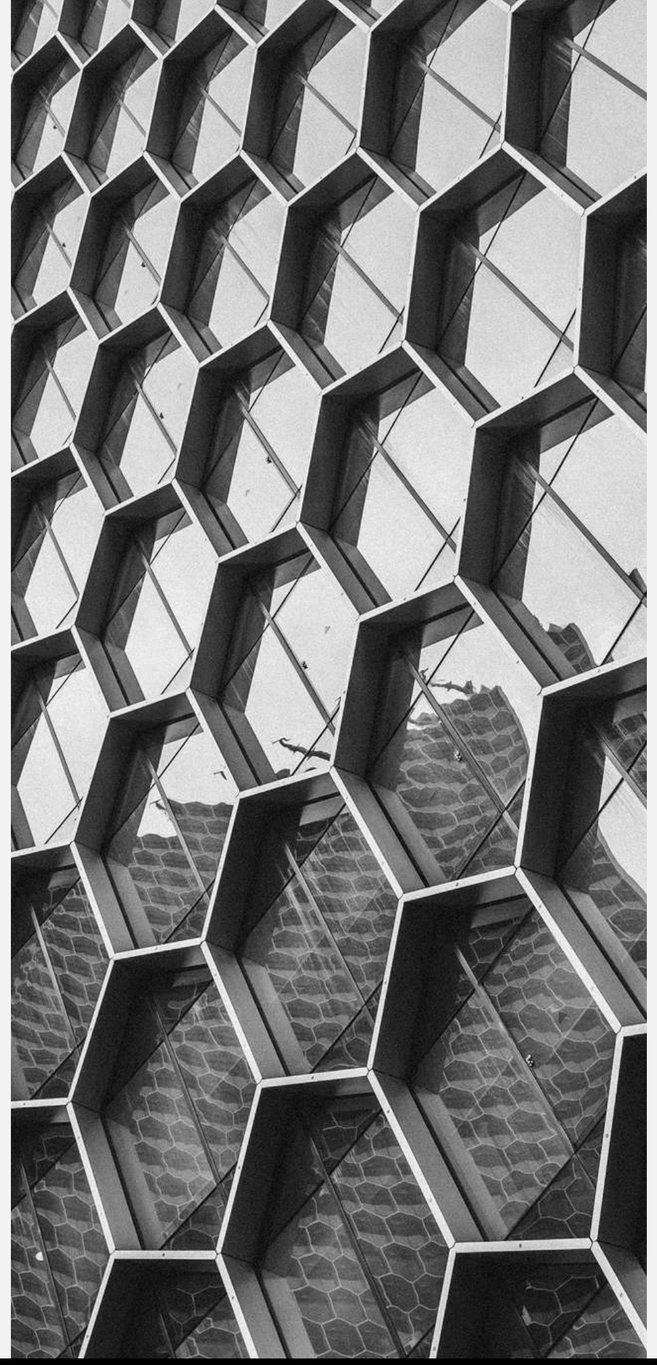
**UML Overview:**
The system follows an **Object-Oriented Programming (OOP)** approach with 4 main classes:

- **Main.java:** The entry point of the game. It initializes the game and starts the gameplay loop.
- **Game.java:** This class controls the game flow, including player turns and alternating between the human player and AI.
- **Board.java:** This class manages the board grid and contains methods for placing discs, printing the board, and checking for winning conditions.
- **AIPlayer.java:** Implements the AI logic, including decision-making through algorithms like Minimax and heuristic-based evaluations.



# SYSTEM ARCHITECTURE





# GAME LEVELING

## Easy

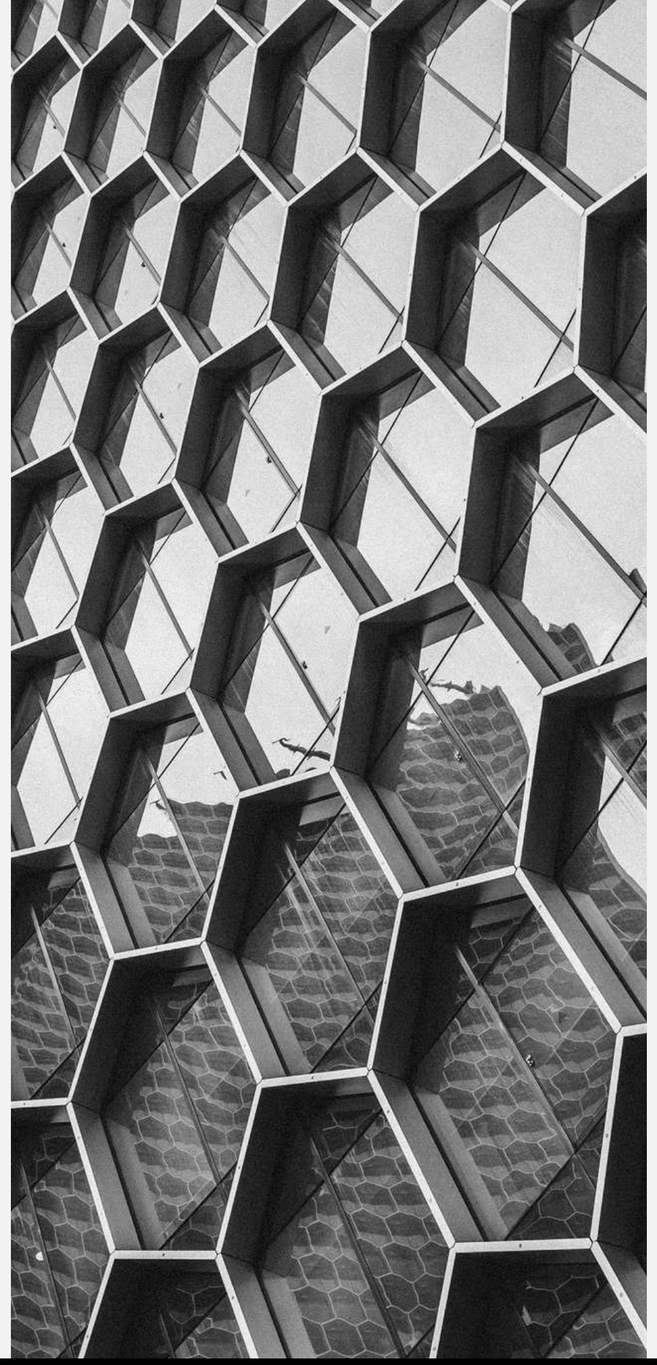
100% of randomness (random number generator from java)

## Medium

- Heuristic Scoring from all moves
- Sorting (merge sort)
- 30% of randomness

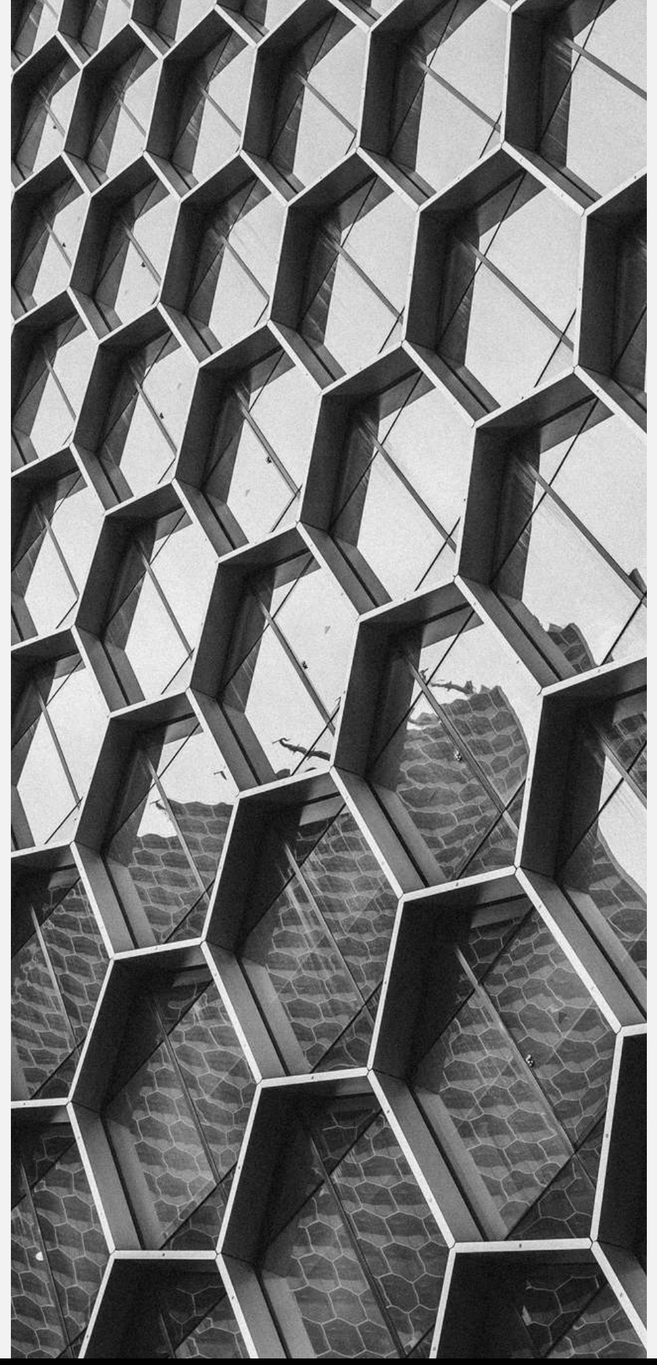
## Hard

- Immediate win check
- Heuristic pre-sorting of moves
- Minimax tree
- Alpha beta pruning



# EASY

```
private int easyMove(Board board, List<Integer> moves) {
    Random rand = new Random();
    return moves.get(rand.nextInt(moves.size()));
}
```



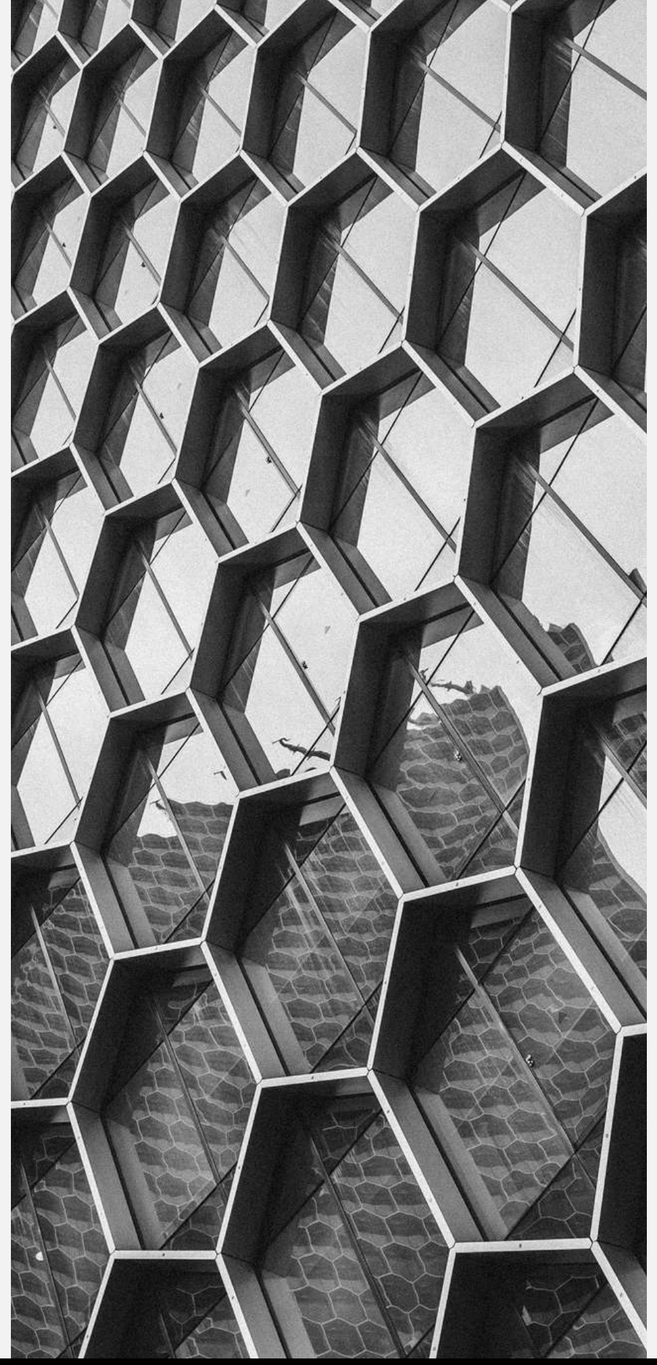
# MEDIUM

```
private int mediumMove(Board board, List<Integer> moves) {
    List<MoveScore> scored = new ArrayList<>();
    for (int c : moves) {
        board.dropDisc(c, AI);
        int s = evaluator.quickScore(board);
        board.undo(c);
        scored.add(new MoveScore(c, s));
    }
    MoveScore[] arr = scored.toArray(new MoveScore[0]);
    MergeSort.sort(arr);
    Random rand = new Random();
    if (rand.nextInt(100) < 30 && arr.length > 1) {
        return arr[1].col;
    }
    return arr[0].col;
}
```

```
private int hardMove(Board board, List<Integer> moves) {
    Integer winCol = immediateWin(board, AI);
    if (winCol != null) return winCol;
    Integer blockCol = immediateWin(board, HUMAN);
    if (blockCol != null) return blockCol;
    List<MoveScore> scored = new ArrayList<>();
    for (int c : moves) {
        board.dropDisc(c, AI);
        int s = evaluator.quickScore(board);
        board.undo(c);
        scored.add(new MoveScore(c, s));
    }
    MoveScore[] arr = scored.toArray(new MoveScore[0]);
    MergeSort.sort(arr);
    List<Integer> orderedMoves = new ArrayList<>();
    for (MoveScore ms : arr) orderedMoves.add(ms.col);
    Minimax.Result result =
            minimax.search(board, DEPTH, Integer.MIN_VALUE, Integer.MAX_VALUE, true, orderedMoves);
    int best = result.bestMove;
    if (best == -1 && !orderedMoves.isEmpty()) best = orderedMoves.get(0);
    return best;
}
```

# HARD

# ALGORITHMS IN CONNECT 4 AI

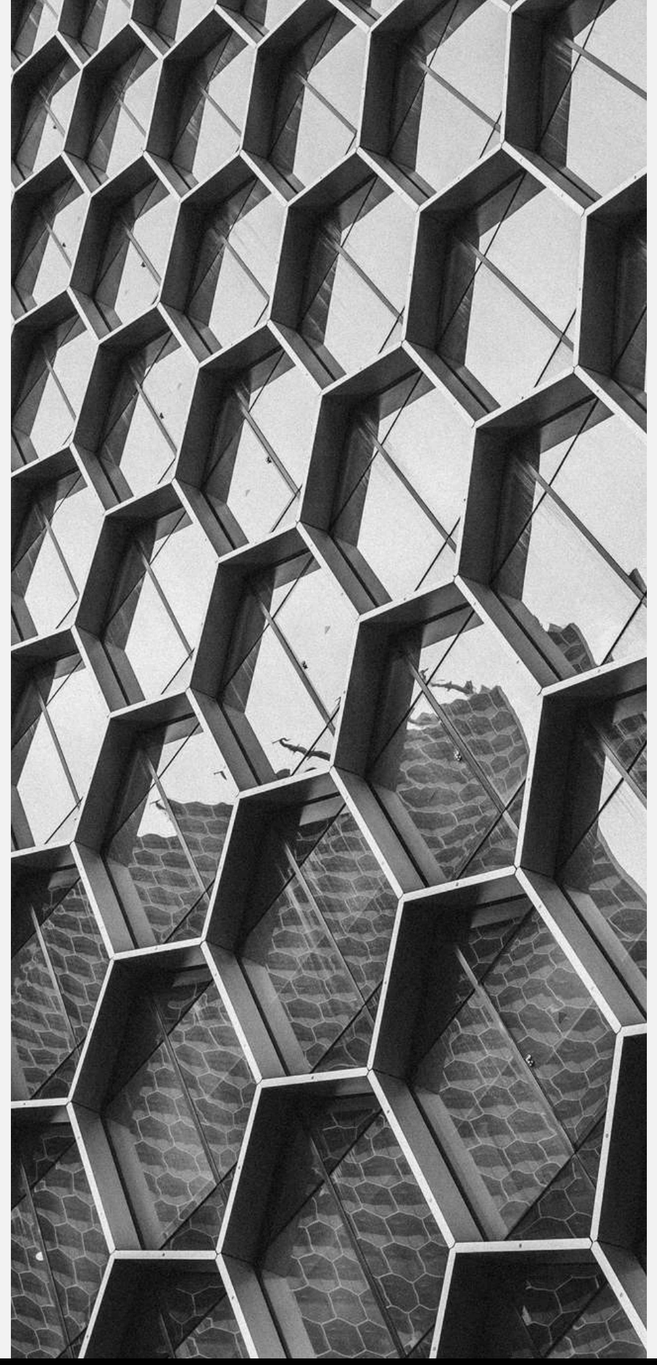


# LINEAR SEARCH

**is a simple algorithm for finding a target value within a list or array.**

## getLegalMoves() → scanning columns

```
public List<Integer> getLegalMoves() {
    List<Integer> list = new ArrayList<>();
    for (int c = 0; c < COLS; c++) {
        if (isValidMove(c)) list.add(c);
    }
    return list;
}
```

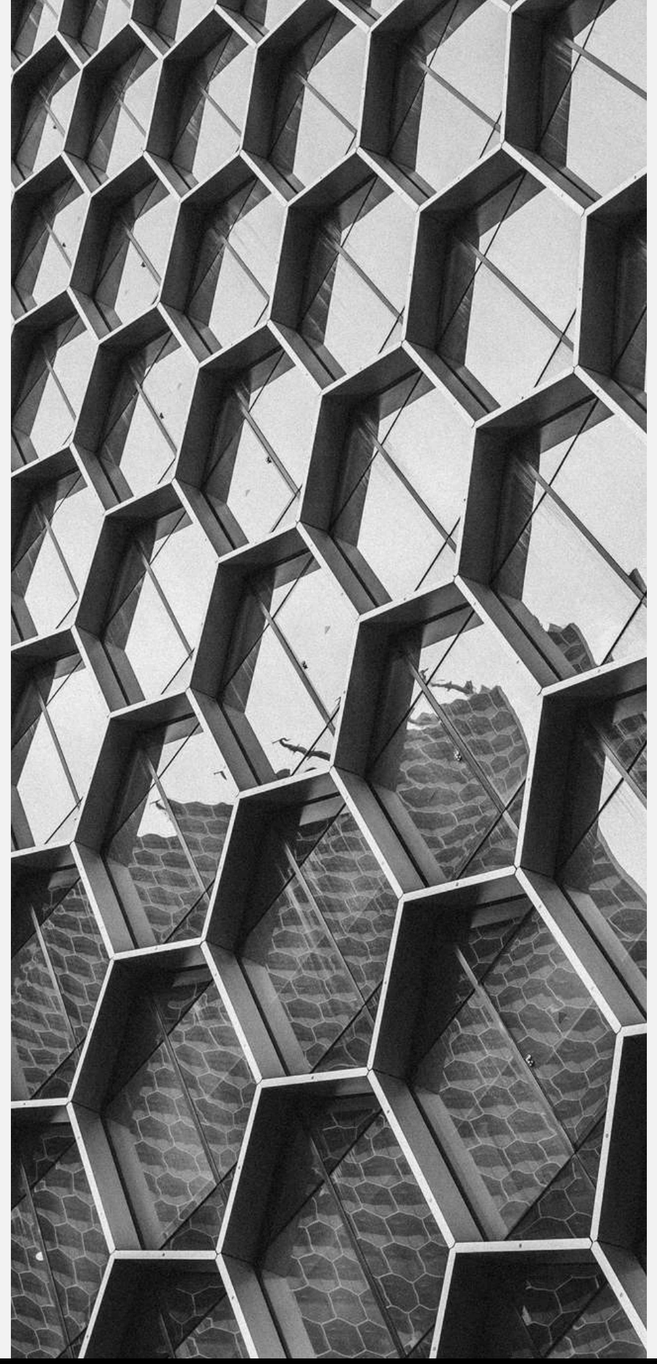


# LINEAR SEARCH

**is a simple algorithm for finding a target value within a list or array.**

## dropDisc() → scanning rows

```
public int dropDisc(int col, char symbol) {
    for (int r = ROWS - 1; r >= 0; r--) {
        if (grid[r][col] == '.') {
            grid[r][col] = symbol;
            heights[col]++;
            return r;
        }
    }
    return -1;
}
```

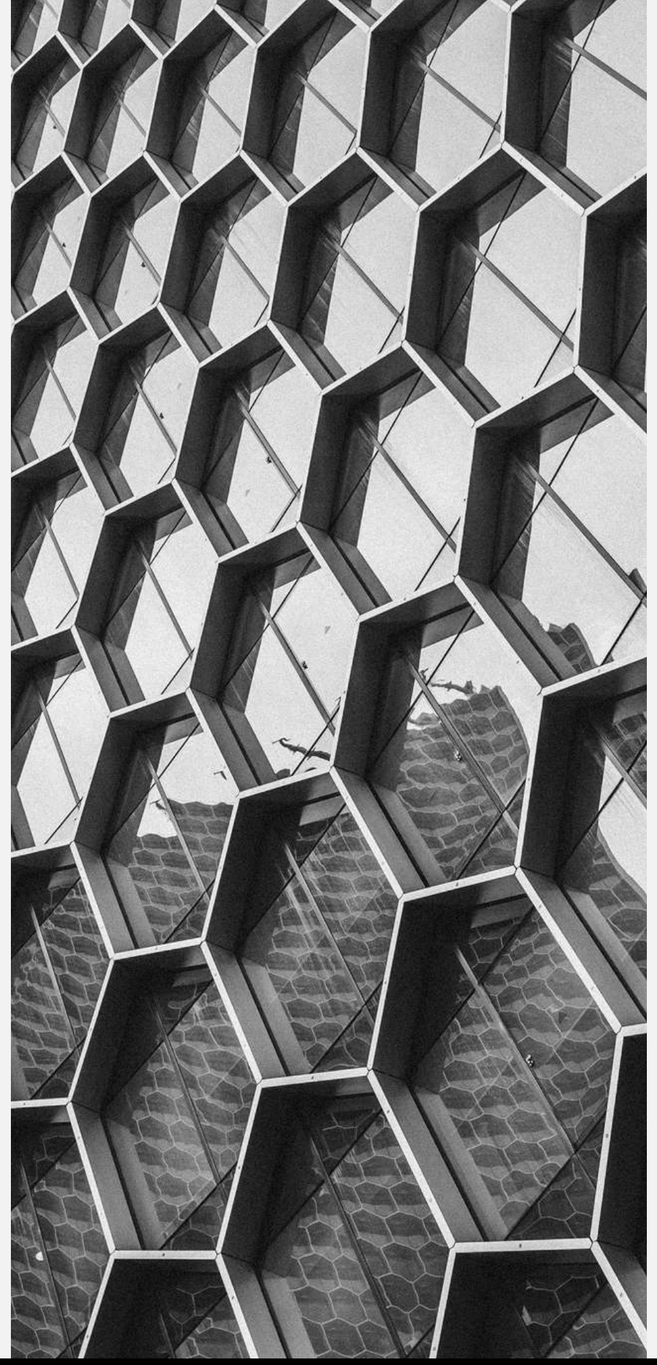


# LINEAR SEARCH

**is a simple algorithm for finding a target value within a list or array.**

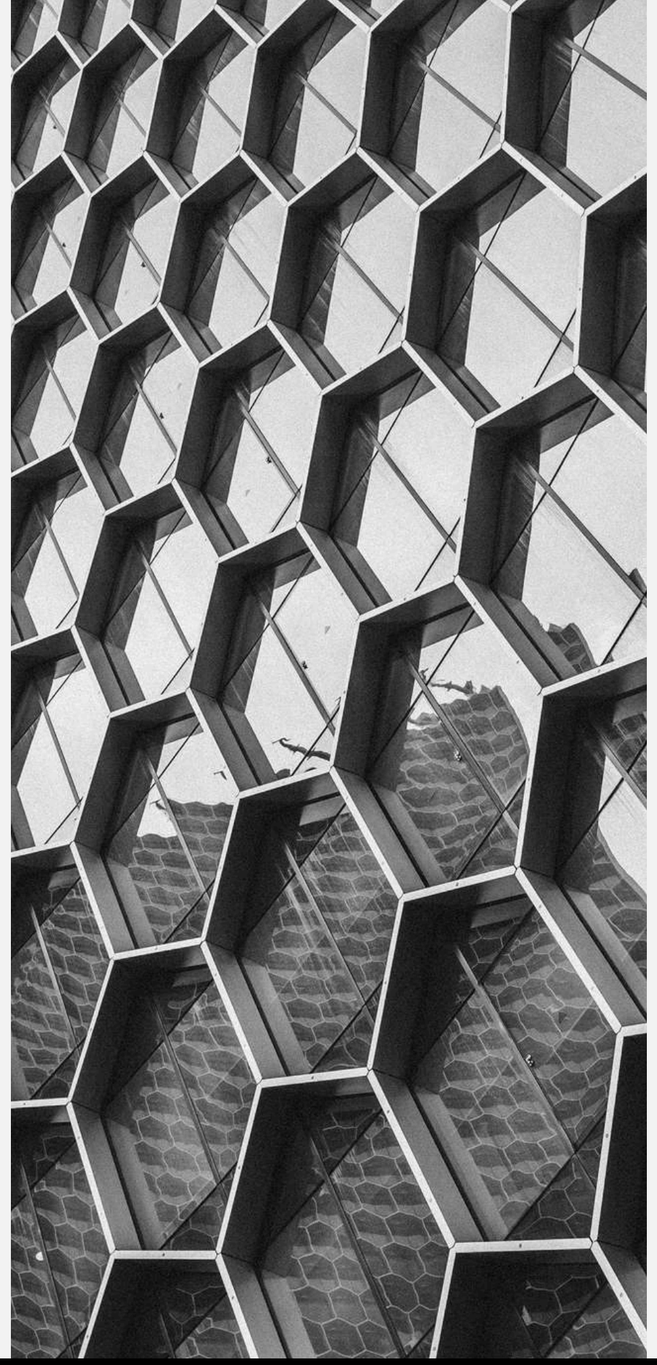
## isFull() → scanning heights array

```
public boolean isFull() {
    for (int h : heights) if (h < ROWS) return false;
    return true;
}
```



# BFS GRAPH

**It explores nodes “level by level” using a queue. In this case study, BFS graph are used in checkWin() and bfsCheck() method because these methods are interrelated with each other.**



# checkWin()

```
public boolean checkWin(char symbol) {
    boolean[][] visited = new boolean[ROWS][COLS];
    int[][] directions = {
            {0, 1},
            {1, 0},
            {1, 1},
            {1, -1}
    };
    for (int r = 0; r < ROWS; r++) {
        for (int c = 0; c < COLS; c++) {
            if (grid[r][c] == symbol && !visited[r][c]) {
                if (bfsCheck(r, c, symbol, visited, directions)) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

- It looks for every cell containing the player's disc (X or O)
- It runs BFS from that cell to explore connected discs
- During BFS, it checks in all directions whether there is a 4-in-a-row

If any BFS finds 4 in a row → player wins.

# bfsCheck()

```
private boolean bfsCheck(int startRow, int startCol, char symbol,
                         boolean[][] visited, int[][] directions) {
    Queue<int[]> queue = new LinkedList<>();
    queue.add(new int[]{startRow, startCol});
    visited[startRow][startCol] = true;
    while (!queue.isEmpty()) {
        int[] cell = queue.poll();
        int r = cell[0];
        int c = cell[1];
        for (int[] dir : directions) {
            int count = 1;
            int nr = r + dir[0];
            int nc = c + dir[1];
            while (nr >= 0 && nr < ROWS && nc >= 0 && nc < COLS
                    && grid[nr][nc] == symbol) {
                count++;
                nr += dir[0];
                nc += dir[1];
            }
            if (count >= 4) {
                return true;
            }
        }
```

- A BFS queue starts with the initial disc position.
- That cell is marked as visited.

- We process every connected disc belonging to the same player.

Directions being checked:

- Horizontal (0,1)
- Vertical (1,0)
- Diagonal down (1,1)
- Diagonal up (1,-1)

This part does the actual winning detection.

# bfsCheck()

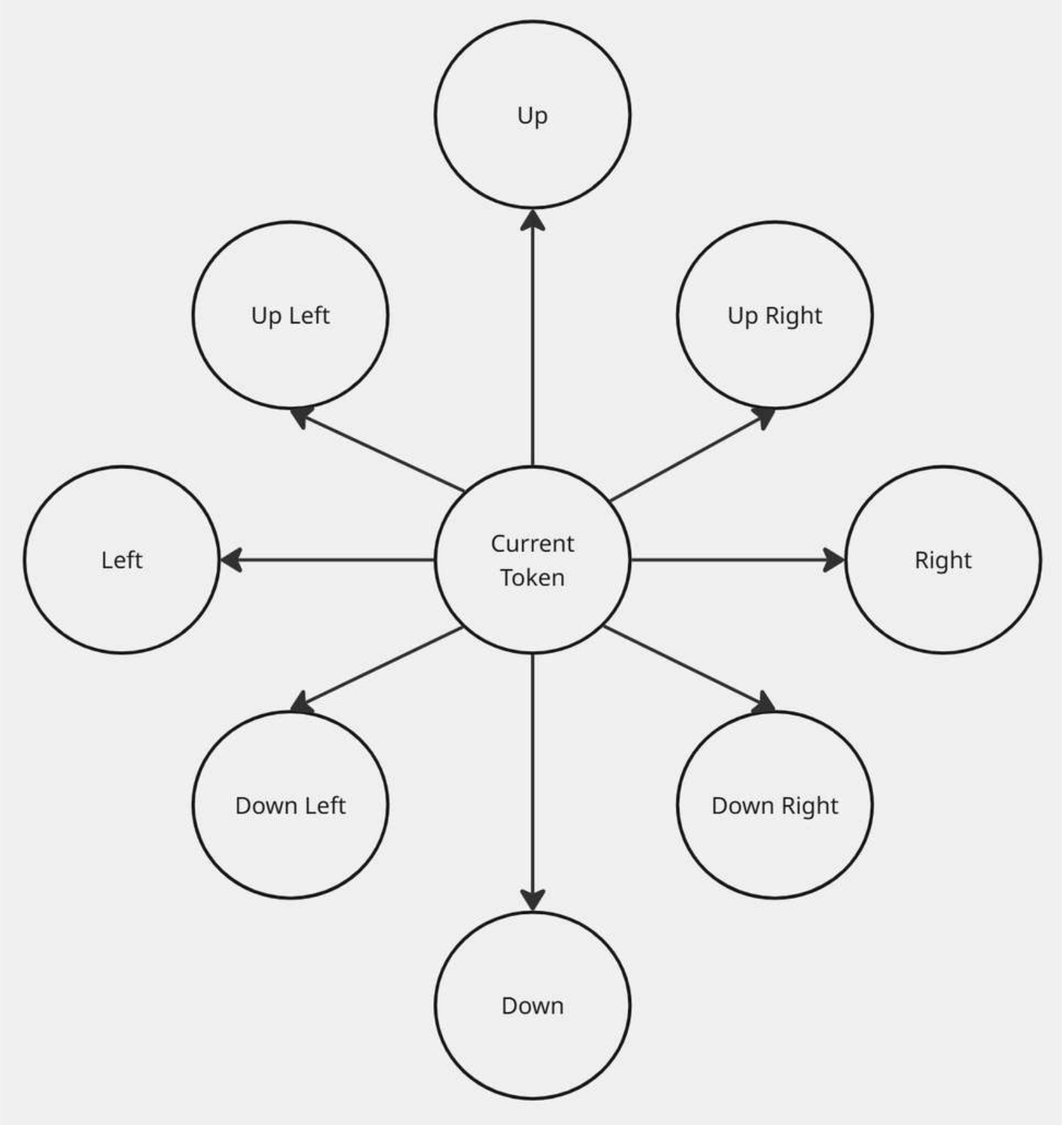
```
// BFS expansion to neighbors
        int[][] neighbors = {
                {0, 1}, {1, 0}, {-1, 0}, {0, -1},
                {1, 1}, {1, -1}, {-1, 1}, {-1, -1}
        };
        for (int[] n : neighbors) {
            int nr = r + n[0];
            int nc = c + n[1];
            if (nr >= 0 && nr < ROWS && nc >= 0 && nc < COLS
                    && !visited[nr][nc]
                    && grid[nr][nc] == symbol) {

                visited[nr][nc] = true;
                queue.add(new int[]{nr, nc});
            }
        }
    }
    return false;
}
```

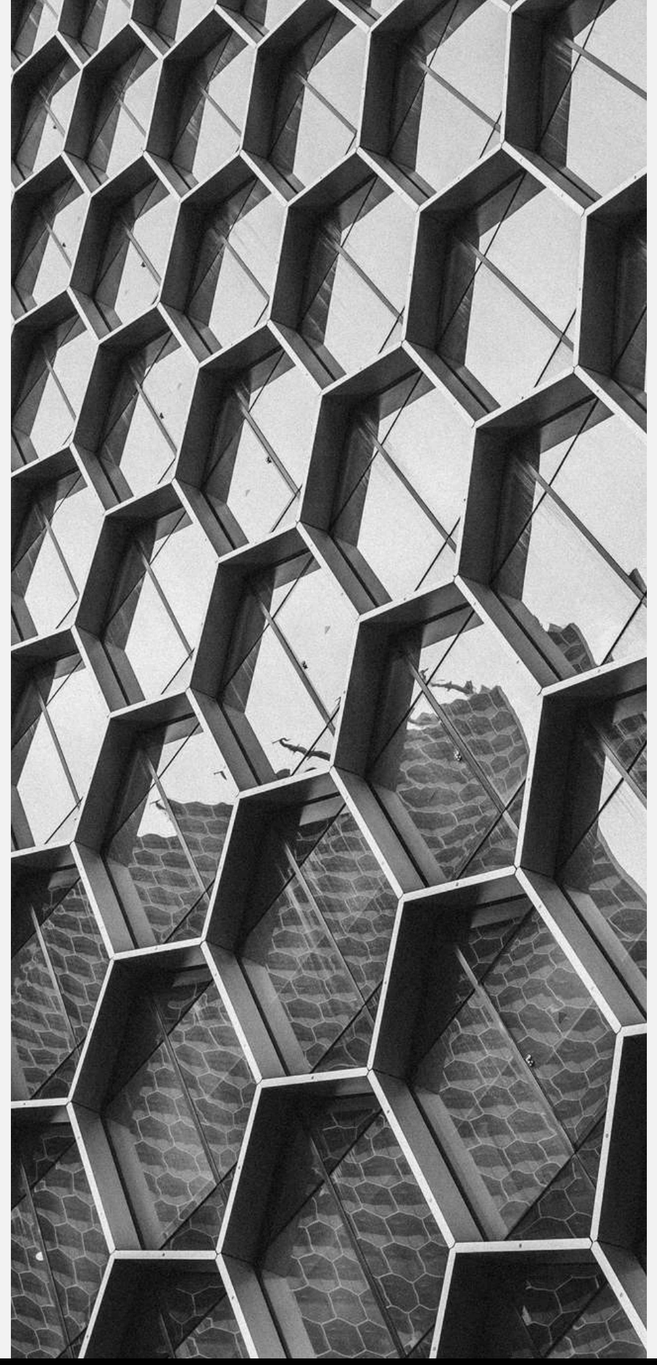
- This checks all 8 neighbors
- It continues BFS to find all discs connected to the first disc, like a flood fill.

Only neighbors with the same symbol are added

If BFS fully explores all connected discs but never finds a line of 4 → no win.

# BFS GRAPH


Up
Up Left
Up Right
Left
Current
Token
Right
Down Left
Down Right
Down



# MERGE SORT

**Merge sort is used for sorting the score from heuristic scoring in medium level.**

## heuristic scoring

```
board.dropDisc(c, AI);
int s = evaluator.quickScore(board);
board.undo(c);
```

**What algorithm is this?**

- **Simulates each possible move.**
- **Uses weights to score AI lines vs human lines.**

**Purpose**

- **Gives the AI a basic understanding of:**
  - **good moves**
  - **bad moves**
  - **dangerous positions**

## merge sort

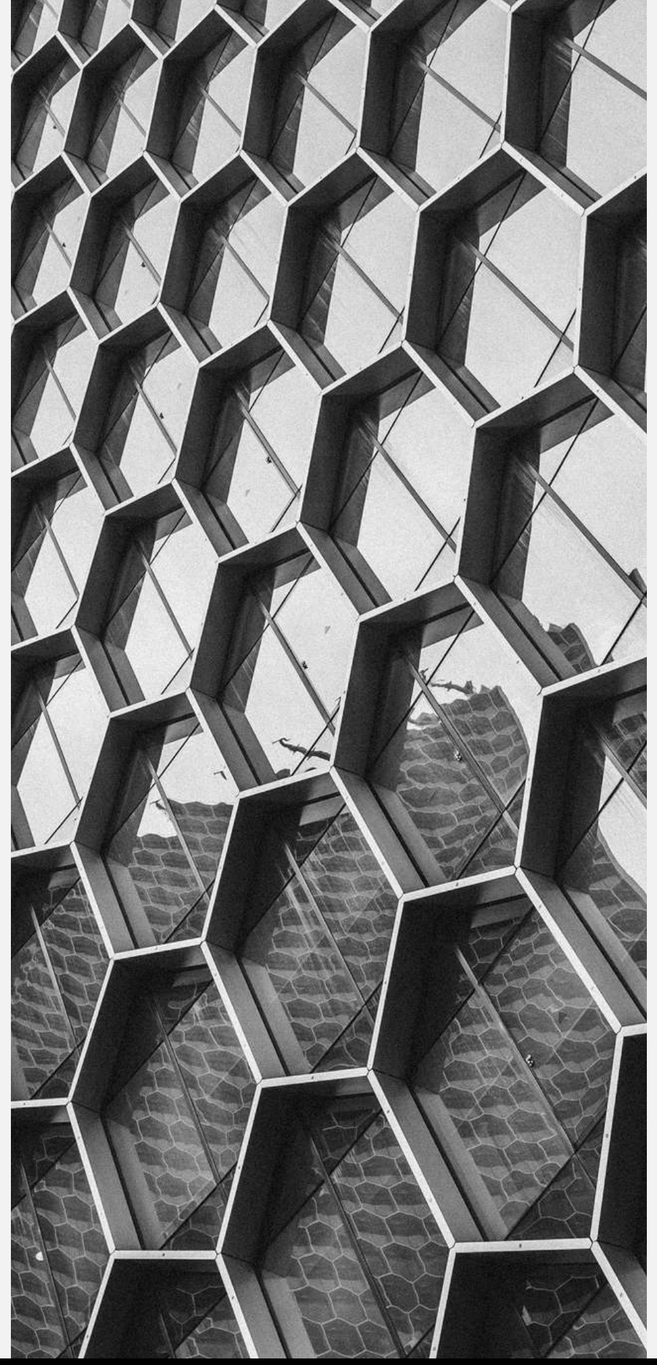
```
MoveScore[] arr = scored.toArray(new
MoveScore[0]);
MergeSort.sort(arr);
```

**This merge sort is called in the mediumMove() method in AIPlayer.java**

# Merge Sort Class

```
public static void sort(MoveScore[] arr) {
    if (arr.length <= 1) return;
    mergeSort(arr, 0, arr.length - 1);
}

private static void mergeSort(MoveScore[] a, int l, int r) {
    if (l >= r) return;
    int m = (l + r) / 2;
    mergeSort(a, l, m);
    mergeSort(a, m + 1, r);
    merge(a, l, m, r);
}
```

```
private static void merge(MoveScore[] a, int l, int m, int r) {
        int n1 = m - l + 1;
        int n2 = r - m;
        MoveScore[] L = new MoveScore[n1];
        MoveScore[] R = new MoveScore[n2];
        System.arraycopy(a, l, L, 0, n1);
        System.arraycopy(a, m + 1, R, 0, n2);
        int i = 0, j = 0, k = l;
        while (i < n1 && j < n2) {
            if (L[i].score >= R[j].score) {
                a[k++] = L[i++];
            } else {
                a[k++] = R[j++];
            }
        }
        while (i < n1) a[k++] = L[i++];
        while (j < n2) a[k++] = R[j++];
    }
}
```



# BRUTE FORCE ONE DEPTH

**evaluating all immediate possible moves and selecting the best one based purely on a static evaluation function, without looking further into subsequent moves or the opponent's responses.**

## immediateWin()

```
private Integer immediateWin(Board board, char symbol) {
    for (int c : board.getLegalMoves()) {
        board.dropDisc(c, symbol);
        boolean win = board.checkWin(symbol);
        board.undo(c);
        if (win) return c;
    }
    return null;
}
```

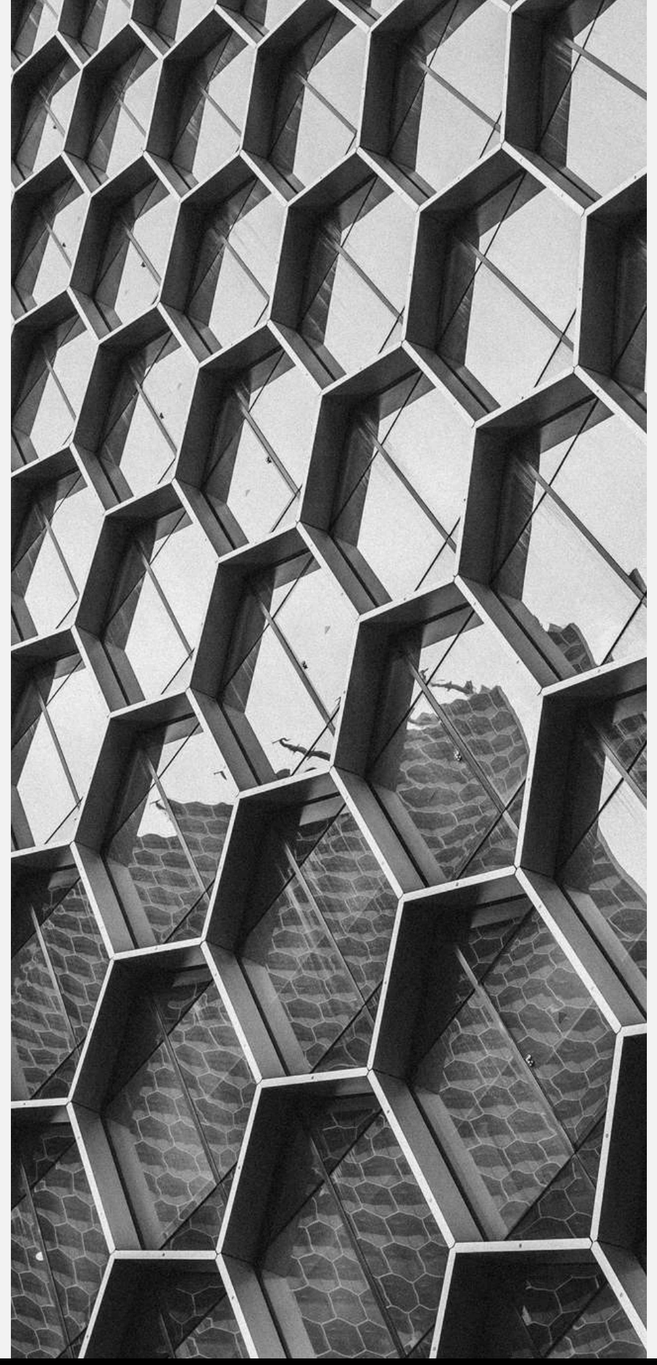
**For each legal move:**

1. **Simulate dropping a disc.**
2. **Check if it produces a win.**
3. **Undo the move.**

**Purpose**

**In Hard mode, before Minimax:**

- **If AI can win → do it.**
- **If Human can win → block it.**



# MINIMAX

**is a decision-making algorithm used in two-player turn-based games (like chess, tic tac toe, or connect 4)**

**In this case study, minimax implemented in 2 methods**

- **search() → top level call**
- **minimax() → deeper recursion**

## search()

**Inputs:**

- **board:** current board state
- **depth:** how many layers ahead AI should think
- **alpha:** best guaranteed score for MAX
- **beta:** best guaranteed score for MIN
- **maximizing:** whose turn? (AI = true, Human = false)
- **orderedMoves:** list of moves sorted using heuristics

# search() method → top level call

## If it's AI turn:

```
if (maximizing) {
    bestScore = Integer.MIN_VALUE;
    for (int col : orderedMoves) {
        if (!board.isValidMove(col)) continue;
        board.dropDisc(col, AI);
        int val = minimax(board, depth - 1, alpha, beta, false);
        board.undo(col);
        if (val > bestScore) {
            bestScore = val;
            bestMove = col;
        }
        alpha = Math.max(alpha, bestScore);
        if (alpha >= beta) break;
    }
}
```

## If it's human turn:

```
else {
    bestScore = Integer.MAX_VALUE;
    for (int col : orderedMoves) {
        if (!board.isValidMove(col)) continue;
        board.dropDisc(col, HUMAN);
        int val = minimax(board, depth - 1, alpha, beta, true);
        board.undo(col);
        if (val < bestScore) {
            bestScore = val;
            bestMove = col;
        }
        beta = Math.min(beta, bestScore);
        if (alpha >= beta) break;
    }
}
```

# minimax() method → deep recursion

## before running minimax

### base case:

```
if (depth == 0 || board.isFull() ||
        board.checkWin(AI) || board.checkWin(HUMAN)) {
    return evaluator.evaluate(board);
}
```

Stops when:

- reached depth
- board is full
- someone has already won

Then it returns board score.

### move ordering:

```
MoveScore[] arr = new MoveScore[moves.size()];
int idx = 0;
for (int col : moves) {
    board.dropDisc(col, maximizing ? AI : HUMAN);
    arr[idx++] = new MoveScore(col,
evaluator.quickScore(board));
    board.undo(col);
}
MergeSort.sort(arr);
```

Before running minimax:

1. generates all moves
2. gives each move a quick heuristic score
3. sorts them descending (best first)

This dramatically improves alpha-beta pruning:

- good moves explored first → more cut branches → faster AI

# minimax() method → deep recursion

when running minimax

## If it’s AI turn:

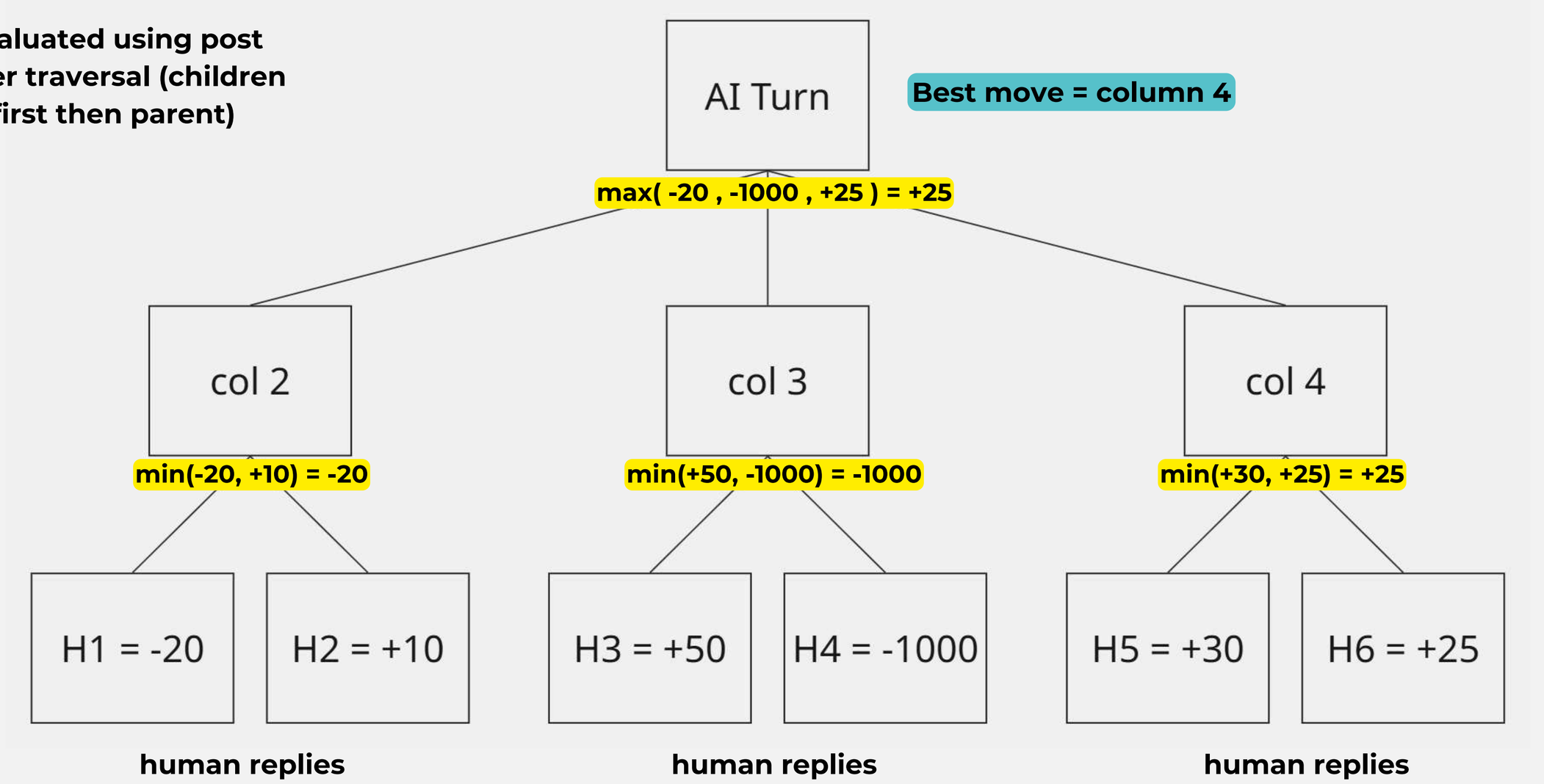
```
if (maximizing) {
    int value = Integer.MIN_VALUE;
    for (int col : ordered) {
        board.dropDisc(col, AI);
        value = Math.max(value, minimax(board, depth - 1,
alpha, beta, false));
        board.undo(col);
        alpha = Math.max(alpha, value);
        if (alpha >= beta) break;
    }
    return value;
}
```

## If it’s human turn:

```
else {
    int value = Integer.MAX_VALUE;
    for (int col : ordered) {
        board.dropDisc(col, HUMAN);
        value = Math.min(value, minimax(board, depth - 1,
alpha, beta, true));
        board.undo(col);
        beta = Math.min(beta, value);
        if (alpha >= beta) break;
    }
    return value;
}
```

# MINIMAX TREE TRAVERSAL

**evaluated using post order traversal (children first then parent)**



# CONCLUSION

- The project successfully built **an intelligent AI opponent** for Connect 4 using **classical algorithms** such as Minimax and Alpha-Beta Pruning.

- Classical algorithms including **Linear Search, Merge Sort, BFS, Brute-Force Evaluation, and Minimax** were effectively integrated to support game logic and AI behavior.

- A **fully functional console-based game was created**, demonstrating how algorithmic techniques can be applied to real gameplay scenarios.

**Check our github:**
**https://github.com/Stephanie020207/Connect4AIGame/tree/master**


THANK YOU