1. Tic-Tac-Toe

Problem: Small 3×3 grid, two players, goal is 3 in a row.

Parameter	BFS	DFS
Time Complexity	O(b^d), b = branching factor (~9 initially), d = depth (max 9 moves). BFS explores all states level by level.	O(b^d), same as BFS. DFS explores a single path first.
Space Complexity	O(b^d), must store all nodes at current level. Can be high for large state spaces, but Tic-Tac-Toe is small.	O(d), only needs stack for current path, very memory efficient.
Failure Conditions	BFS may fail if memory exhausted (rare for Tic-Tac-Toe).	DFS may fail if depth limit is too low or if it hits infinite loops (not an issue for Tic-Tac-Toe because finite states).

Observation: For Tic-Tac-Toe, DFS is usually preferred for simple backtracking; BFS is overkill due to small board size.

DFS

```
#include <bits/stdc++.h>
using namespace std;

const char AI = 'O';
const char HUMAN = 'X';
const char EMPTY = '_';

bool movesLeft(vector<vector<char>>& board) {
    for (auto &r : board)
        for (char c : r)
            if (c == EMPTY) return true;
    return false;
}

int evaluate(vector<vector<char>>& board) {
    for (int i = 0; i < 3; i++) {
        // rows
        if (board[i][0] == board[i][1] &&
            board[i][1] == board[i][2]) {
            if (board[i][0] == AI) return 10;
            if (board[i][0] == HUMAN) return -10;
        }
        // columns
        if (board[0][i] == board[1][i] &&
            board[1][i] == board[2][i]) {
            if (board[0][i] == AI) return 10;
            if (board[0][i] == HUMAN) return -10;
        }
    }

    // diagonals
    if (board[0][0] == board[1][1] &&
```

```
        board[1][1] == board[2][2]) {
        if (board[0][0] == AI) return 10;
        if (board[0][0] == HUMAN) return -10;
    }

    if (board[0][2] == board[1][1] &&
        board[1][1] == board[2][0]) {
        if (board[0][2] == AI) return 10;
        if (board[0][2] == HUMAN) return -10;
    }

    return 0;
}

int minimax(vector<vector<char>>& board, bool isMax) {
    int score = evaluate(board);
    if (score != 0) return score;
    if (!movesLeft(board)) return 0;

    if (isMax) { // AI turn
        int best = -1000;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = AI;
                    best = max(best, minimax(board, false));
                    board[i][j] = EMPTY;
                }
            }
        }
        return best;
    } else { // Human turn
        int best = 1000;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = HUMAN;
                    best = min(best, minimax(board, true));
                    board[i][j] = EMPTY;
                }
            }
        }
        return best;
    }
}

pair<int,int> bestMove(vector<vector<char>>& board, bool aiTurn) {
    int bestVal = aiTurn ? -1000 : 1000;
    pair<int,int> move = {-1, -1};

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
```

```
            if (board[i][j] == EMPTY) {
                board[i][j] = aiTurn ? AI : HUMAN;
                int val = minimax(board, !aiTurn);
                board[i][j] = EMPTY;

                if ((aiTurn && val > bestVal) ||
                    (!aiTurn && val < bestVal)) {
                    bestVal = val;
                    move = {i, j};
                }
            }
        }
    }
    return move;
}

int main() {
    vector<vector<char>> board(3, vector<char>(3));
    int xCount = 0, oCount = 0;

    cout << "Enter board (3 lines):\n";
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cin >> board[i][j];
            if (board[i][j] == 'X') xCount++;
            if (board[i][j] == 'O') oCount++;
        }
    }

    bool aiTurn = (oCount < xCount);  // O plays after X

    auto move = bestMove(board, aiTurn);

    if (move.first == -1)
        cout << "No valid moves (game over)\n";
    else
        cout << "Best move: Row " << move.first
             << ", Col " << move.second << endl;

    return 0;
}
```

BFS

```
#include <bits/stdc++.h>
using namespace std;

const char AI = 'O';
const char HUMAN = 'X';
const char EMPTY = '_';

struct State {
    string board;
```

```
    char turn;
};

int evaluate(const string &b) {
    int wins[8][3] = {
        {0,1,2},{3,4,5},{6,7,8},
        {0,3,6},{1,4,7},{2,5,8},
        {0,4,8},{2,4,6}
    };

    for (auto &w : wins) {
        if (b[w[0]] == b[w[1]] && b[w[1]] == b[w[2]]) {
            if (b[w[0]] == AI) return 10;
            if (b[w[0]] == HUMAN) return -10;
        }
    }
    return 0;
}

bool movesLeft(const string &b) {
    return b.find(EMPTY) != string::npos;
}

vector<string> nextStates(const string &b, char turn) {
    vector<string> states;
    for (int i = 0; i < 9; i++) {
        if (b[i] == EMPTY) {
            string nb = b;
            nb[i] = turn;
            states.push_back(nb);
        }
    }
    return states;
}

int main() {
    string start = "_________"; // empty board
    char startTurn = AI;

    unordered_map<string, int> value;
    unordered_map<string, vector<string>> graph;
    unordered_map<string, int> outdeg;

    queue<State> q;
    q.push({start, startTurn});

    // BFS: build game graph
    while (!q.empty()) {
        auto [b, turn] = q.front(); q.pop();
        if (graph.count(b)) continue;

        int score = evaluate(b);
```

```
    if (score != 0 || !movesLeft(b)) {
        value[b] = score;
        continue;
    }

    vector<string> children = nextStates(b, turn);
    graph[b] = children;
    outdeg[b] = children.size();

    for (auto &c : children)
        q.push({c, turn == AI ? HUMAN : AI});
}

// Reverse graph
unordered_map<string, vector<string>> parent;
for (auto &[p, childs] : graph)
    for (auto &c : childs)
        parent[c].push_back(p);

queue<string> rq;
for (auto &[s, v] : value)
    rq.push(s);

// Retrograde BFS (minimax propagation)
while (!rq.empty()) {
    string cur = rq.front(); rq.pop();

    for (auto &p : parent[cur]) {
        if (value.count(p)) continue;

        char turn = (count(p.begin(), p.end(), AI) ==
                    count(p.begin(), p.end(), HUMAN)) ? AI : HUMAN;

        if (turn == AI)
            value[p] = -1000;
        else
            value[p] = 1000;

        for (auto &c : graph[p]) {
            if (!value.count(c)) {
                value.erase(p);
                break;
            }
            if (turn == AI)
                value[p] = max(value[p], value[c]);
            else
                value[p] = min(value[p], value[c]);
        }

        if (value.count(p))
            rq.push(p);
    }
```

```
    }

    // Find best move
    int best = -1000;
    int bestMove = -1;

    for (int i = 0; i < 9; i++) {
        if (start[i] == EMPTY) {
            string nb = start;
            nb[i] = AI;
            if (value[nb] > best) {
                best = value[nb];
                bestMove = i;
            }
        }
    }

    cout << "Best move index (0–8): " << bestMove << endl;
    return 0;
}
```

2. Sudoku

Problem: 9×9 grid, fill numbers 1–9 without violating rules.

Parameter	BFS	DFS
Time Complexity	O(b^d), b = number of valid choices per empty cell (~1–9), d = number of empty cells. BFS explores all partial solutions at once.	O(b^d), same as BFS, but DFS explores one full path before backtracking.
Space Complexity	O(b^d), huge for large number of empty cells. Can explode quickly.	O(d), just a stack for recursion. Much more memory efficient.
Failure Conditions	BFS may run out of memory with many empty cells.	DFS may be inefficient if not using heuristics (like most constrained cell first), but usually guaranteed to find solution if it exists.

Observation: DFS with backtracking is the standard approach for Sudoku because it uses much less memory and can easily prune invalid paths early. BFS is rarely used because of huge space requirements.

DFS

```
#include <iostream>

#include <vector>


using namespace std;


// Check if placing 'num' at board[row][col] is valid
```

```
bool isValid(vector<vector<int>>& board, int row, int col, int num) {
    for (int i = 0; i < 9; i++) {
        // Check row and column
        if (board[row][i] == num || board[i][col] == num)
            return false;
        // Check 3x3 subgrid
        int subRow = 3 * (row / 3) + i / 3;
        int subCol = 3 * (col / 3) + i % 3;
        if (board[subRow][subCol] == num)
            return false;
    }
    return true;
}

// DFS + Backtracking solver
bool solveSudoku(vector<vector<int>>& board) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            if (board[row][col] == 0) { // empty cell
                for (int num = 1; num <= 9; num++) {
                    if (isValid(board, row, col, num)) {
                        board[row][col] = num;
                        if (solveSudoku(board))
                            return true;
                        board[row][col] = 0; // backtrack
                    }
                }
                return false; // no valid number found
            }
        }
    }
    return true; // solved
```

```
}

// Utility function to print the board
void printBoard(const vector<vector<int>>& board) {
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    vector<vector<int>> board = {
        {5,3,0,0,7,0,0,0,0},
        {6,0,0,1,9,5,0,0,0},
        {0,9,8,0,0,0,0,6,0},
        {8,0,0,0,6,0,0,0,3},
        {4,0,0,8,0,3,0,0,1},
        {7,0,0,0,2,0,0,0,6},
        {0,6,0,0,0,0,2,8,0},
        {0,0,0,4,1,9,0,0,5},
        {0,0,0,0,8,0,0,7,9}
    };

    if (solveSudoku(board)) {
        cout << "Solved Sudoku:\n";
        printBoard(board);
    } else {
        cout << "No solution exists.\n";
    }
```

```
    return 0;
}
```

BFS

```
#include <bits/stdc++.h>
using namespace std;

using Board = vector<vector<char>>;

// Check if placing ch at (row, col) is valid
bool isValid(const Board &board, int row, int col, char ch) {
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == ch) return false; // row
        if (board[i][col] == ch) return false; // column
    }

    int boxRow = (row / 3) * 3;
    int boxCol = (col / 3) * 3;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            if (board[boxRow + i][boxCol + j] == ch)
                return false;

    return true;
}

// Check if board is complete
bool isSolved(const Board &board) {
    for (int i = 0; i < 9; i++)
        for (int j = 0; j < 9; j++)
            if (board[i][j] == '.')
                return false;
```

```
    return true;
}

// BFS Sudoku Solver
bool solveSudokuBFS(Board &board) {
    queue<Board> q;
    q.push(board);

    while (!q.empty()) {
        Board curr = q.front();
        q.pop();

        if (isSolved(curr)) {
            board = curr;
            return true;
        }

        // find first empty cell
        int row = -1, col = -1;
        bool found = false;
        for (int i = 0; i < 9 && !found; i++) {
            for (int j = 0; j < 9; j++) {
                if (curr[i][j] == '.') {
                    row = i;
                    col = j;
                    found = true;
                    break;
                }
            }
        }

        // try all digits
```

```
        for (char ch = '1'; ch <= '9'; ch++) {
            if (isValid(curr, row, col, ch)) {
                Board next = curr;
                next[row][col] = ch;
                q.push(next);
            }
        }
    }
    return false;
}

// Helper to print board
void printBoard(const Board &board) {
    for (auto &row : board) {
        for (char c : row)
            cout << c << " ";
        cout << "\n";
    }
}

int main() {
    Board board = {
        {'5','3','.','.','7','.','.','.','.'},
        {'6','.','.','1','9','5','.','.','.'},
        {'.','9','8','.','.','.','.','.','6','.'},
        {'8','.','.','.','.','6','.','.','.','3'},
        {'4','.','.','8','.','3','.','.','1'},
        {'7','.','.','.','.','2','.','.','.','6'},
        {'.','6','.','.','.','.','.','2','8','.'},
        {'.','.','.','4','1','9','.','.','5'},
        {'.','.','.','.','8','.','.','7','9'}
    };
```

```
    if (solveSudokuBFS(board)) {
        cout << "Solved Sudoku:\n";
        printBoard(board);
    } else {
        cout << "No solution found.\n";
    }
}
```

3. N-Queens

Problem: Place N queens on N×N chessboard such that no two threaten each other.

Parameter	BFS	DFS
Time Complexity	O(N^N), branching factor = N (choices per row), depth = N. Explores all partial boards level by level.	O(N^N), same as BFS; DFS explores one board configuration at a time.
Space Complexity	O(N^N), stores all partial boards at each level. Extremely high for large N.	O(N), only need stack for recursion + board array. Very efficient.
Failure Conditions	BFS may fail for large N due to memory explosion.	DFS may fail if not using backtracking/pruning, but backtracking ensures correct solution.

Observation: DFS with backtracking is the standard for N-Queens. BFS is impractical except for very small N.

DFS

```
#include <iostream>
#include <vector>
using namespace std;

int N;
vector<int> board;

bool isSafe(int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row))
            return false;
    }
    return true;
}

void dfsNQueen(int row) {
    if (row == N) {
        for (int i = 0; i < N; i++)
            cout << board[i] << " ";
        cout << endl;
```

```
        return;
    }

    for (int col = 0; col < N; col++) {
        if (isSafe(row, col)) {
            board[row] = col;
            dfsNQueen(row + 1);
        }
    }
}

int main() {
    cout << "Enter N: ";
    cin >> N;
    board.resize(N);
    dfsNQueen(0);
    return 0;
}
```

BFS

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

bool isSafe(const vector<int>& board, int row, int col) {
    for (int i = 0; i < row; i++) {
        if (board[i] == col || abs(board[i] - col) == abs(i - row))
            return false;
    }
    return true;
}

void bfsNQueen(int N) {
    queue<vector<int>> q;
    q.push({});

    while (!q.empty()) {
        vector<int> curr = q.front();
        q.pop();

        int row = curr.size();
        if (row == N) {
            for (int x : curr) cout << x << " ";
            cout << endl;
            continue;
        }

        for (int col = 0; col < N; col++) {
            if (isSafe(curr, row, col)) {
                vector<int> next = curr;
                next.push_back(col);
```

```
                q.push(next);
            }
        }
    }
}

int main() {
    int N;
    cout << "Enter N: ";
    cin >> N;
    bfsNQueen(N);
    return 0;
}
```