DFS	BFS

DFS

```
#include <iostream>
#include <vector>
using namespace std;
const int N = 100;
vector<int> g[N];
bool visited[N];
vector<int> path;
// Simple DFS function to find the goal node
bool dfs(int vertex, int goalNode) {
  visited[vertex] = true;
  path.push_back(vertex);
  // Check if we reached the goal node
  if (vertex == goalNode) return true;
  // Visit all unvisited neighbors
  for (int child : g[vertex]) {
    if (!visited[child]) {
      if (dfs(child, goalNode)) return true; // Goal found, exit
    }
  }
  // Backtrack if goal not found in this path
  path.pop_back();
  return false;
}
int main() {
  int node, edge;
  cout << "Enter number of nodes and edges: ";
  cin >> node >> edge;
  cout << "Enter edges (u v):" << endl;
  for (int i = 0; i < edge; i++) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v);
    g[v].push_back(u);
```

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1e5 + 2;
bool vis[N];
vector<int> adj[N];
int main() {
  int n, m;
  cout << "Enter number of nodes and edges: ";
  cin >> n >> m;
  // Initialize visited array
  for (int i = 0; i < n + 1; i++) {
    vis[i] = false;
  }
  // Input edges
  cout << "Enter edges (u v):" << endl;
  for (int i = 0; i < m; i++) {
    int x, y;
    cin >> x >> y;
    adj[x].push_back(y);
    adj[y].push_back(x);
  }
  int start, goal;
  cout << "Enter start and goal nodes: ";
  cin >> start >> goal;
  // BFS traversal with level tracking
  queue<int> q;
  q.push(start);
  vis[start] = true;
  bool found = false;
  int level = 0;
  cout << "Level-wise traversal from " << start << " to " << goal <<
":" << endl;
  while (!q.empty()) {
    int size = q.size();  // Number of nodes at the current level
```

```
  }
  int goalNode;
  cout << "Enter goal node: ";
  cin >> goalNode;

  if (dfs(1, goalNode)) {  // Start DFS from node 1
    cout << "Path to goal node " << goalNode << ": ";
    for (int v : path) {
      cout << v << " ";
    }
    cout << endl;
  } else {
    cout << "Goal node " << goalNode << " not found in the graph." << endl;
  }
  return 0;
}
```

DLS

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

const int N = 100;
vector<int> g[N];
bool visited[N];
vector<int> path;

// DLS function with depth limit
bool dls(int vertex, int goalNode, int limit) {
  visited[vertex] = true;
  path.push_back(vertex);

  // Check if we reached the goal node
  if (vertex == goalNode) return true;

  // Stop recursion if the depth limit is reached
  if (limit <= 0) {
    path.pop_back();  // Backtrack
```

```
      cout << "Level " << level << ": ";
      for (int i = 0; i < size; i++) {
        int node = q.front();
        q.pop();
        cout << node << " ";
        // Check if we reached the goal node
        if (node == goal) {
          found = true;
        }
        // Traverse adjacent nodes
        for (int neighbor : adj[node]) {
          if (vis[neighbor]==false) {
            vis[neighbor] = true;
            q.push(neighbor);
          }
        }
      }
      cout << endl;
      level++;
      // Stop processing further levels once the goal is found
      if (found) break;
    }
  return 0;
}
```

IDS

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

const int N = 100;
vector<int> g[N];
bool visited[N];
vector<int> path;
```

```
    return false;
  }

  // Visit all unvisited neighbors with a reduced depth limit
  for (int child : g[vertex]) {
    if (!visited[child]) {
      if (dls(child, goalNode, limit - 1)) return true; // Goal
found
    }
  }

  // Backtrack if goal not found in this path
  path.pop_back();
  return false;
}

int main() {
  int node, edge;
  cout << "Enter number of nodes and edges: ";
  cin >> node >> edge;

  cout << "Enter edges (u v):" << endl;
  for (int i = 0; i < edge; i++) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v);
    g[v].push_back(u);
  }

  int goalNode, depthLimit;
  cout << "Enter goal node: ";
  cin >> goalNode;
  cout << "Enter depth limit: ";
  cin >> depthLimit;

  if (dls(1, goalNode, depthLimit)) {  // Start DLS from node
1 with depth limit
    cout << "Path to goal node " << goalNode << ": ";
    for (int v : path) {
      cout << v << " ";
    }
    cout << endl;
  } else {
    cout << "Goal node " << goalNode << " not found within
depth limit " << depthLimit << "." << endl;
  }

  return 0;
}
```

```
// Depth-Limited Search function
bool dls(int vertex, int goalNode, int limit) {
  visited[vertex] = true;
  path.push_back(vertex);

  // Check if we reached the goal node
  if (vertex == goalNode) return true;

  // Stop recursion if the depth limit is reached
  if (limit <= 0) {
    path.pop_back();  // Backtrack
    return false;
  }

  // Visit all unvisited neighbors with a reduced depth limit
  for (int child : g[vertex]) {
    if (!visited[child]) {
      if (dls(child, goalNode, limit - 1)) return true; // Goal
found
    }
  }

  // Backtrack if goal not found in this path
  path.pop_back();
  return false;
}

// Iterative Deepening Search (IDS)
bool ids(int start, int goalNode, int maxDepth) {
  for (int depth = 0; depth <= maxDepth; depth++) {
    fill(visited, visited + N, false);  // Reset visited array for
each depth
    path.clear();  // Clear path for each new depth level
    if (dls(start, goalNode, depth)) {
      return true;  // Goal found at this depth
    }
  }
  return false;  // Goal not found within maxDepth
}

int main() {
  int node, edge;
  cout << "Enter number of nodes and edges: ";
  cin >> node >> edge;

  cout << "Enter edges (u v):" << endl;
  for (int i = 0; i < edge; i++) {
    int u, v;
    cin >> u >> v;
    g[u].push_back(v);
    g[v].push_back(u);
```

	`` } int goalNode, maxDepth; cout << "Enter goal node: "; cin >> goalNode; cout << "Enter maximum depth for IDS: "; cin >> maxDepth; if (ids(1, goalNode, maxDepth)) { // Start IDS from node 1 cout << "Path to goal node " << goalNode << ": "; for (int v : path) { cout << v << " "; } cout << endl; } else { cout << "Goal node " << goalNode << " not found within maximum depth " << maxDepth << "." << endl; } return 0; } ```

UCS	A*

UCS

```cpp
#include <bits/stdc++.h>

using namespace std;

const int N = 1e5 + 2;

vector<pair<int, int>> adj[N];  // adj[node] = list of (neighbor, cost)

vector<int> parent(N, -1);      // Track path

vector<int> dist(N, INT_MAX);   // Distance from start node

// Function to print the path from start to goal

void printPath(int start, int goal) {

   vector<int> path;

   for (int v = goal; v != -1; v = parent[v]) {

      path.push_back(v);

   }

   reverse(path.begin(), path.end());

   cout << "Path from " << start << " to " << goal << " with minimum cost:\n";

   for (int node : path) {

      cout << node << " ";

   }

   cout << endl;

   cout << "Total cost: " << dist[goal] << endl;

}

void uniformCostSearch(int start, int goal) {

   priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

   pq.push({0, start});

   dist[start] = 0;

   while (!pq.empty()) {

      int cost = pq.top().first;

      int node = pq.top().second;

      pq.pop();

      // Stop if we reach the goal node with minimum cost

      if (node == goal) {

         printPath(start, goal);

         return;
```

A*

```cpp
;#include <iostream>
#include <vector>
#include <queue>
#include <map>
#include <cmath>
#include <algorithm>

using namespace std;

// Define the 8-puzzle state as a 3x3 vector
struct PuzzleState {
   vector<vector<int>> state;
   int x, y; // Position of the blank (0)
   int cost, level;
   string path;

   bool operator<(const PuzzleState& other) const {
      return (cost + level) > (other.cost + other.level);
   }

// Calculate Manhattan distance
int calculateManhattan(const vector<vector<int>>& current, const vector<vector<int>>& goal) {
   int distance = 0;
   for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++) {
         if (current[i][j] != 0) {
            for (int x = 0; x < 3; x++) {
               for (int y = 0; y < 3; y++) {
                  if (current[i][j] == goal[x][y]) {
                     distance += abs(i - x) + abs(j - y);
                  }
               }
            }
         }
      }
   }
   return distance;
}

// Check if the state is valid (within bounds)
bool isValid(int x, int y) {
   return x >= 0 && x < 3 && y >= 0 && y < 3;
}

// Print the 3x3 puzzle state
void printState(const vector<vector<int>>& state) {
   for (const auto& row : state) {
      for (int val : row) {
         cout << val << " ";
      }
      cout << endl;
   }
   cout << "- - -" << endl;
```

```
    }

    // Explore neighbors

    for (auto neighbor : adj[node]) {

      int nextNode = neighbor.first;

      int edgeCost = neighbor.second;

      int newCost = cost + edgeCost;

      // If a cheaper path is found, update the cost and path

      if (newCost < dist[nextNode]) {

        dist[nextNode] = newCost;

        parent[nextNode] = node;

        pq.push({newCost, nextNode});

      }

    }

  }

  cout << "No path found from " << start << " to " << goal << endl;

}

int main() {

  int n, m;

  cout << "Enter number of nodes and edges: ";

  cin >> n >> m;

  cout << "Enter edges (u v cost):" << endl;

  for (int i = 0; i < m; i++) {

    int u, v, cost;

    cin >> u >> v >> cost;

    adj[u].push_back({v, cost});

    adj[v].push_back({u, cost});  // For undirected graphs; remove
if directed

  }

  int start, goal;

  cout << "Enter start and goal nodes: ";

  cin >> start >> goal;

  uniformCostSearch(start, goal);

  return 0;
```

```
}

// Perform the A* algorithm
void solve8Puzzle(vector<vector<int>> start,
vector<vector<int>> goal) {
  // Define possible moves for the blank space
  int dx[] = {1, 0, -1, 0};
  int dy[] = {0, 1, 0, -1};

  priority_queue<PuzzleState> pq;
  map<vector<vector<int>>, bool> visited;

  int startX, startY;
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
      if (start[i][j] == 0) {
        startX = i;
        startY = j;
      }
    }
  }

  PuzzleState initial = {start, startX, startY,
calculateManhattan(start, goal), 0, ""};
  pq.push(initial);

  while (!pq.empty()) {
    PuzzleState current = pq.top();
    pq.pop();

    if (current.state == goal) {
      cout << "Solution found with path: " << current.path
<<" "<< endl; //0->down, 1->right, 2->up, 3->left
      printState(current.state);
      return;
    }

    if (visited[current.state]) {
      continue;
    }
    visited[current.state] = true;

    for (int i = 0; i < 4; i++) {
      int newX = current.x + dx[i];
      int newY = current.y + dy[i];

      if (isValid(newX, newY)) {
        vector<vector<int>> newState = current.state;
        swap(newState[current.x][current.y],
newState[newX][newY]);
```

```
}
```

```
            if (!visited[newState]) {
               int newCost = calculateManhattan(newState, goal);
               pq.push({newState, newX, newY, newCost, current.level + 1, current.path + to_string(i)});
            }
         }
      }
   }

   cout << "No solution found." << endl;
}

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

   vector<vector<int>> goal = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 0}
   };

   solve8Puzzle(start, goal);
   return 0;
}
```

N Queens Backtrack

```
#include <iostream>

#include <vector>

using namespace std;

int N;

vector<vector<char>> board;  // Chessboard represented as a 2D grid

// Function to print one solution

void printSolution() {

   cout << "One solution for " << N << "-Queens problem:\n";

   for (int i = 0; i < N; i++) {

      for (int j = 0; j < N; j++) {

         cout << board[i][j] << " ";
```

N Queens Genetic

```
#include <iostream>

#include <vector>

#include <algorithm>

#include <ctime>

#include <cstdlib>

using namespace std;

const int N = 4; // Number of queens

const int POP_SIZE = 100; // Population size

const int MAX_GEN = 1000; // Maximum generations

const double MUTATION_RATE = 0.05; // Mutation rate
```

`` } cout << endl; } } // Function to check if a queen placement is safe bool isSafe(int row, int col) { // Check the column for (int i = 0; i < row; i++) { if (board[i][col] == 'Q') return false; } // Check the upper-left diagonal for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) { if (board[i][j] == 'Q') return false; } // Check the upper-right diagonal for (int i = row - 1, j = col + 1; i >= 0 && j < N; i--, j++) { if (board[i][j] == 'Q') return false; } return true; } // Recursive function to solve the N-Queens problem row by row bool solve(int row) { if (row == N) { // All queens are placed successfully printSolution(); return true; } for (int col = 0; col < N; col++) { if (isSafe(row, col)) { board[row][col] = 'Q'; // Place the queen if (solve(row + 1)) { // Try to place queens in the next row return true; // Stop when a solution is found } ```	``` // Chromosome structure representing a solution (queen positions in each row) struct Chromosome { vector<int> genes; int fitness; Chromosome() : genes(N), fitness(0) { // Initialize chromosome with random queen positions for (int i = 0; i < N; ++i) { genes[i] = rand() % N; } } // Calculate the fitness of the chromosome (number of non-attacking pairs) void calculateFitness() { fitness = 0; for (int i = 0; i < N; ++i) { for (int j = i + 1; j < N; ++j) { // Check for non-attacking pairs if (genes[i] != genes[j] && abs(genes[i] - genes[j]) != abs(i - j)) { fitness++; } } } } }; // Genetic Algorithm functions Chromosome crossover(const Chromosome &parent1, const Chromosome &parent2) { Chromosome child; int crossoverPoint = rand() % N; for (int i = 0; i < N; ++i) { child.genes[i] = (i < crossoverPoint) ? parent1.genes[i] : parent2.genes[i]; ```

```
        board[row][col] = '.';  // Backtrack: remove the queen
      }
    }
    return false;  // No solution found in this configuration
}
int main() {
    cout << "Enter the number of queens: ";
    cin >> N;
    // Initialize the board with empty spaces (.)
    board = vector<vector<char>>(N, vector<char>(N, '.'));
    // Try to find one solution
    if (!solve(0)) {
      cout << "No solution found for " << N << "-Queens problem.\n";
    }
    return 0;
}
```

```
    }
    return child;
}
void mutate(Chromosome &chromosome) {
    if ((double) rand() / RAND_MAX < MUTATION_RATE) {
      int pos = rand() % N;
      chromosome.genes[pos] = rand() % N;
    }
}
// Function to select a parent using tournament selection
Chromosome selectParent(const vector<Chromosome> &population) {
    int tournamentSize = 5;
    Chromosome best = population[rand() % POP_SIZE];
    for (int i = 1; i < tournamentSize; ++i) {
      Chromosome contender = population[rand() % POP_SIZE];
      if (contender.fitness > best.fitness) {
        best = contender;
      }
    }
    return best;
}
int main() {
    srand(time(0));
    vector<Chromosome> population(POP_SIZE);

    // Initialize population and calculate fitness
    for (auto &chromosome : population) {
      chromosome.calculateFitness();
    }
    int generation = 0;
    Chromosome bestSolution;
    // Genetic algorithm loop
```

	`` while (generation < MAX_GEN) { sort(population.begin(), population.end(), [](const Chromosome &a, const Chromosome &b) { return a.fitness > b.fitness; }); if (population[0].fitness == (N * (N - 1)) / 2) { // Max fitness for non-attacking pairs bestSolution = population[0]; break; } vector<Chromosome> newPopulation; // Selection and crossover to create a new population for (int i = 0; i < POP_SIZE; ++i) { Chromosome parent1 = selectParent(population); Chromosome parent2 = selectParent(population); Chromosome child = crossover(parent1, parent2); mutate(child); child.calculateFitness(); newPopulation.push_back(child); } population = newPopulation; generation++; } // Print the solution if (bestSolution.fitness == (N * (N - 1)) / 2) { cout << "Solution found in generation " << generation << ":\n"; for (int i = 0; i < N; ++i) { for (int j = 0; j < N; ++j) { if (j == bestSolution.genes[i]) { cout << "Q "; } else { cout << ". "; } ```

```
      }
      cout << endl;
    }
  } else {
    cout << "No solution found.\n";
  }
  return 0;
}
```

Tic Tac Toe Minimax	Tic Tac Toe Alpha Beta

Tic Tac Toe Minimax

```
#include <iostream>
#include <vector>
#include <climits>
using namespace std;
const int SIZE = 3;  // Board size (3x3 for Tic-Tac-Toe)
vector<vector<char>> board(SIZE, vector<char>(SIZE, ' '));  // Initialize empty board


// Function to display the board
void displayBoard() {
  cout << "\n";
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      cout << " " << board[i][j] << " ";
      if (j < SIZE - 1) cout << "|";
    }
    cout << "\n";
    if (i < SIZE - 1) cout << "---|---|---\n";
  }
  cout << "\n";
}
// Function to check for a winner
```

Tic Tac Toe Alpha Beta

```
#include <bits/stdc++.h>
using namespace std;
const int SIZE = 4; // Board size
const int WINNING_LENGTH = 4; // Winning length (4 in a row)
const char HUMAN = 'X';
const char COMPUTER = 'O';
const char EMPTY = '_';


// Function to print the board
void printBoard(const vector<vector<char>>& board) {
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      cout << board[i][j] << " ";
    }
    cout << endl;
  }
}
// Check if a player has won
bool isGameOver(const vector<vector<char>>& board, char player) {
  // Check rows and columns
  for (int i = 0; i < SIZE; i++) {
```

```
char checkWinner() {
  // Check rows and columns
  for (int i = 0; i < SIZE; i++) {
    if (board[i][0] == board[i][1] && board[i][1] == board[i][2] && board[i][0] != ' ') return board[i][0];  // Row check
    if (board[0][i] == board[1][i] && board[1][i] == board[2][i] && board[0][i] != ' ') return board[0][i];  // Column check
  }
  // Check diagonals
  if (board[0][0] == board[1][1] && board[1][1] == board[2][2] && board[0][0] != ' ') return board[0][0];  // Top-left to bottom-right
  if (board[0][2] == board[1][1] && board[1][1] == board[2][0] && board[0][2] != ' ') return board[0][2];  // Top-right to bottom-left
  return ' ';  // No winner
}

// Function to check if the board is full (draw condition)
bool isBoardFull() {
  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      if (board[i][j] == ' ') return false;
    }
  }
  return true;
}
// Minimax algorithm to find the best move for the AI
int minimax(int depth, bool isMaximizingPlayer) {
  char winner = checkWinner();
  if (winner == 'X') return -1;  // Player wins
  if (winner == 'O') return 1;   // AI wins
  if (isBoardFull()) return 0;   // Draw
  if (isMaximizingPlayer) {
    int best = INT_MIN;  // Maximize AI's score
    for (int i = 0; i < SIZE; i++) {
```

```
    for (int j = 0; j <= SIZE - WINNING_LENGTH; j++) {
      bool winRow = true, winCol = true;
      for (int k = 0; k < WINNING_LENGTH; k++) {
        if (board[i][j + k] != player) winRow = false;
        if (board[j + k][i] != player) winCol = false;
      }
      if (winRow || winCol) return true;
    }
  }
  // Check diagonals
  for (int i = 0; i <= SIZE - WINNING_LENGTH; i++) {
    for (int j = 0; j <= SIZE - WINNING_LENGTH; j++) {
      bool winDiag1 = true, winDiag2 = true;
      for (int k = 0; k < WINNING_LENGTH; k++) {
        if (board[i + k][j + k] != player) winDiag1 = false;
        if (board[i + k][j + WINNING_LENGTH - 1 - k] != player) winDiag2 = false;
      }
      if (winDiag1 || winDiag2) return true;
    }
  }
  return false;
}
// Evaluate board state
int evaluate(const vector<vector<char>>& board) {
  if (isGameOver(board, COMPUTER)) return 10;
  if (isGameOver(board, HUMAN)) return -10;
  return 0;
}
// Check if there are moves left
bool isMovesLeft(const vector<vector<char>>& board) {
  for (const auto& row : board)
    for (char cell : row)
```

```
        for (int j = 0; j < SIZE; j++) {
          if (board[i][j] == ' ') {
            board[i][j] = 'O';  // AI's move
            best = max(best, minimax(depth + 1, false));
            board[i][j] = ' ';  // Undo move
          }
        }
      }
      return best;
    } else {
      int best = INT_MAX;  // Minimize player's score
      for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
          if (board[i][j] == ' ') {
            board[i][j] = 'X';  // Player's move
            best = min(best, minimax(depth + 1, true));
            board[i][j] = ' ';  // Undo move
          }
        }
      }
      return best;
    }
}
// Function to find the best move for AI using Minimax
pair<int, int> findBestMove() {
  int bestVal = INT_MIN;
  pair<int, int> bestMove = {-1, -1};

  for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++) {
      if (board[i][j] == ' ') {
        board[i][j] = 'O';  // AI's move
```

```
      if (cell == EMPTY) return true;
  return false;
}
// Minimax algorithm with alpha-beta pruning
int minimax(vector<vector<char>>& board, int depth, bool isMax,
int alpha, int beta) {
  int score = evaluate(board);
  if (score == 10 || score == -10 || depth == 0 ||
!isMovesLeft(board)) return score;


  if (isMax) {
    int best = INT_MIN;
    for (int i = 0; i < SIZE; i++) {
      for (int j = 0; j < SIZE; j++) {
        if (board[i][j] == EMPTY) {
          board[i][j] = COMPUTER;
          best = max(best, minimax(board, depth - 1, false, alpha,
beta));
          board[i][j] = EMPTY;
          alpha = max(alpha, best);
          if (beta <= alpha) return best;
        }
      }
    }
    return best;
  } else {
    int best = INT_MAX;
    for (int i = 0; i < SIZE; i++) {
      for (int j = 0; j < SIZE; j++) {
        if (board[i][j] == EMPTY) {
          board[i][j] = HUMAN;
          best = min(best, minimax(board, depth - 1, true, alpha,
beta));
          board[i][j] = EMPTY;
```

`int moveVal = minimax(0, false);` `board[i][j] = ' ';  // Undo move` `if (moveVal > bestVal) {` `bestMove = {i, j};` `bestVal = moveVal;` `}` `}` `}` `}` `return bestMove;` `}` `// Main game loop with AI` `void playGame() {` `char currentPlayer = 'X';  // Human starts first` `while (true) {` `displayBoard();` `if (currentPlayer == 'X') {` `// Player's move` `int row, col;` `cout << "Player X, enter your move (row and column): ";` `cin >> row >> col;` `// Validate input` `if (row < 1 || row > SIZE || col < 1 || col > SIZE || board[row - 1][col - 1] != ' ') {` `cout << "Invalid move. Try again.\n";` `continue;` `}` `board[row - 1][col - 1] = currentPlayer;` `} else {` `// AI's move` `cout << "AI (Player O) is making a move...\n";` `pair<int, int> bestMove = findBestMove();`	`beta = min(beta, best);` `if (beta <= alpha) return best;` `}` `}` `}` `return best;` `}` `}` `// Find the best move for the computer` `pair<int, int> findBestMove(vector<vector<char>>& board) {` `int bestValue = INT_MIN;` `pair<int, int> bestMove = {-1, -1};` `for (int i = 0; i < SIZE; i++) {` `for (int j = 0; j < SIZE; j++) {` `if (board[i][j] == EMPTY) {` `board[i][j] = COMPUTER;` `int moveValue = minimax(board, 3, false, INT_MIN, INT_MAX);` `board[i][j] = EMPTY;` `if (moveValue > bestValue) {` `bestMove = {i, j};` `bestValue = moveValue;` `}` `}` `}` `}` `return bestMove;` `}` `// Main function` `int main() {` `vector<vector<char>> board(SIZE, vector<char>(SIZE, EMPTY));` `printBoard(board);`

```
        board[bestMove.first][bestMove.second] = currentPlayer;
      }
      // Check for a winner
      char winner = checkWinner();
      if (winner != ' ') {
        displayBoard();
        cout << "Player " << winner << " wins!\n";
        break;
      }
      // Check for a draw
      if (isBoardFull()) {
        displayBoard();
        cout << "It's a draw!\n";
        break;
      }
      // Switch player
      currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
    }
}
int main() {
    cout << "Welcome to Tic-Tac-Toe! You are X and the AI is O.\n";
    playGame();
    return 0;
}
```

```
    while (true) {
      int row, col;
      cout << "Enter row and column (1-based index): ";
      cin >> row >> col;
      row--; col--; // Convert to 0-based indexing for internal processing
      if (row < 0 || col < 0 || row >= SIZE || col >= SIZE || board[row][col] != EMPTY) {
        cout << "Invalid move. Try again." << endl;
        continue;
      }

      board[row][col] = HUMAN;
      if (isGameOver(board, HUMAN)) {
        printBoard(board);
        cout << "You won!" << endl;
        break;
      }
      auto [bestRow, bestCol] = findBestMove(board);
      board[bestRow][bestCol] = COMPUTER;
      printBoard(board);
      if (isGameOver(board, COMPUTER)) {
        cout << "Computer won!" << endl;
        break;
      }
      if (!isMovesLeft(board)) {
        cout << "It's a tie!" << endl;
        break;
      }
    }
    return 0;
}
```