Artificial Intelligence Codebook

by MOHAMMAD EMRAN AHMED EMON

BSc in SOFTWARE ENGINEERING,

SHAHJALAL UNIVERSITY OF SCIENCE AND TECHNOLOGY, SYLHET.

Registration No: 2020831040

Session: 2020-21

1.nQueens

```
#include <iostream>
#include <vector>

using namespace std;

const int N = 8;

bool isSafe(vector<vector<int> >& board, int row, int col)
{
	for (int x = 0; x < col; x++)
		if (board[row][x] == 1)
			return false;
	for (int x = row, y = col; x >= 0 && y >= 0; x--, y--)
		if (board[x][y] == 1)
			return false;
	for (int x = row, y = col; x < N && y >= 0; x++, y--)
		if (board[x][y] == 1)
			return false;
	return true;
}

bool solveNQueens(vector<vector<int> >& board, int col)
{
	if (col == N) {
		for (int i = 0; i < N; i++) {
			for (int j = 0; j < N; j++)
				cout << board[i][j] << " ";
			cout << endl;
		}
		cout << endl;
		return true;
	}
	for (int i = 0; i < N; i++) {
		if (isSafe(board, i, col)) {
			board[i][col] = 1;
			if (solveNQueens(board, col + 1))
				return true;
			board[i][col] = 0;
		}
	}
	return false;
}

int main()
{
	vector<vector<int> > board(N, vector<int>(N, 0));
	if (!solveNQueens(board, 0))
		cout << "No solution found";
	return 0;
}
```

2.Minimax Basics:

```
// A simple C++ program to find
// maximum score that
// maximizing player can get.
#include<bits/stdc++.h>
using namespace std;

// Returns the optimal value a maximizer can obtain.
// depth is current depth in game tree.
// nodeIndex is index of current node in scores[].
// isMax is true if current move is
// of maximizer, else false
// scores[] stores leaves of Game tree.
// h is maximum height of Game tree
int minimax(int depth, int nodeIndex, bool isMax,
			int scores[], int h)
{
	// Terminating condition. i.e
	// leaf node is reached
	if (depth == h)
		return scores[nodeIndex];

	// If current move is maximizer,
	// find the maximum attainable
	// value
	if (isMax)
	return max(minimax(depth+1, nodeIndex*2, false,
scores, h),
			minimax(depth+1, nodeIndex*2 + 1,
false, scores, h));

	// Else (If current move is Minimizer), find the minimum
	// attainable value
	else
		return min(minimax(depth+1, nodeIndex*2,
true, scores, h),
			minimax(depth+1, nodeIndex*2 + 1,
true, scores, h));
}

// A utility function to find Log n in base 2
int log2(int n)
{
return (n==1)? 0 : 1 + log2(n/2);
}

// Driver code
int main()
{
	// The number of elements in scores must be
	// a power of 2.
	int scores[] = {3, 5, 2, 9, 12, 5, 23, 23};
	int n = sizeof(scores)/sizeof(scores[0]);
	int h = log2(n);
	int res = minimax(0, 0, true, scores, h);
	cout << "The optimal value is : " << res << endl;
	return 0;
}
```

3. Minimax Basics 2

```
// C++ program to find the next optimal move for
// a player
#include<bits/stdc++.h>
using namespace std;
struct Move
{
	int row, col;
};
char player = 'x', opponent = 'o';
// This function returns true if there are moves
// remaining on the board. It returns false if
// there are no moves left to play.
bool isMovesLeft(char board[3][3])
{
	for (int i = 0; i<3; i++)
		for (int j = 0; j<3; j++)
			if (board[i][j]=='_')
				return true;
	return false;
}
// This is the evaluation function as discussed
// in the previous article ( http://goo.gl/sJgv68 )
int evaluate(char b[3][3])
{
	// Checking for Rows for X or O victory.
	for (int row = 0; row<3; row++)
	{
		if (b[row][0]==b[row][1] &&
			b[row][1]==b[row][2])
		{
			if (b[row][0]==player)
				return +10;
			else if (b[row][0]==opponent)
				return -10;
		}
	}
	// Checking for Columns for X or O victory.
	for (int col = 0; col<3; col++)
	{
		if (b[0][col]==b[1][col] &&
			b[1][col]==b[2][col])
		{
			if (b[0][col]==player)
				return +10;

			else if (b[0][col]==opponent)
				return -10;
		}
	}
	// Checking for Diagonals for X or O victory.
	if (b[0][0]==b[1][1] && b[1][1]==b[2][2])
	{
		if (b[0][0]==player)
			return +10;
		else if (b[0][0]==opponent)
			return -10;
	}
```

4. Minimax Basics 3

```
#include<bits/stdc++.h>
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
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; j++) {
        if (board[i][j] == ' ') {
          board[i][j] = 'O';  // AI's move
```

```
	if (b[0][2]==b[1][1] && b[1][1]==b[2][0])
	{
		if (b[0][2]==player)
			return +10;
		else if (b[0][2]==opponent)
			return -10;
	}
	// Else if none of them have won then return 0
	return 0;
}
// This is the minimax function. It considers all
// the possible ways the game can go and returns
// the value of the board
int minimax(char board[3][3], int depth, bool isMax)
{
	int score = evaluate(board);
	// If Maximizer has won the game return his/her
	// evaluated score
	if (score == 10)
		return score;
	// If Minimizer has won the game return his/her
	// evaluated score
	if (score == -10)
		return score;
	// If there are no more moves and no winner then
	// it is a tie
	if (isMovesLeft(board)==false)
		return 0;
	// If this maximizer's move
	if (isMax)
	{
		int best = -1000;
		// Traverse all cells
		for (int i = 0; i<3; i++)
		{
			for (int j = 0; j<3; j++)
			{
				// Check if cell is empty
				if (board[i][j]=='_')
				{
					// Make the move
					board[i][j] = player;
					// Call minimax
recursively and choose
					// the maximum
value
					best = max( best,
minimax(board, depth+1, !isMax) );
					// Undo the move
					board[i][j] = '_';
				}
			}
		}
		return best;
	}
	// If this minimizer's move
	else
```

```
          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
        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;
      }
```

```
    {
        int best = 1000;

        // Traverse all cells
        for (int i = 0; i<3; i++)
        {
            for (int j = 0; j<3; j++)
            {
                // Check if cell is empty
                if (board[i][j]=='_')
                {
                    // Make the move
                    board[i][j] =
opponent;
                    // Call minimax
recursively and choose
                    // the minimum
value
                    best = min(best,
minimax(board, depth+1, !isMax));
                    // Undo the move
                    board[i][j] = '_';
                }
            }
        }
        return best;
    }
}
// This will return the best possible move for the player
Move findBestMove(char board[3][3])
{
    int bestVal = -1000;
    Move bestMove;
    bestMove.row = -1;
    bestMove.col = -1;

    // Traverse all cells, evaluate minimax function for
    // all empty cells. And return the cell with optimal
    // value.
    for (int i = 0; i<3; i++)
    {
        for (int j = 0; j<3; j++)
        {
            // Check if cell is empty
            if (board[i][j]=='_')
            {
                // Make the move
                board[i][j] = player;
                // compute evaluation
function for this
                // move.
                int moveVal = minimax(board,
0, false);
                // Undo the move
                board[i][j] = '_';
                // If the value of the current
move is
```

```
      board[row - 1][col - 1] = currentPlayer;
    } else {
      // AI's move
      cout << "AI (Player O) is making a move...\n";
      pair<int, int> bestMove = findBestMove();
      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;
}
```

7.BFS

```
#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
```

```
                                        // more than the best value,
 then update
                                        // best/
                                        if (moveVal > bestVal)
                                        {
                                                  bestMove.row = i;
                                                  bestMove.col = j;
                                                  bestVal = moveVal;
                                        }
                              }
                    }
          }
          printf("The value of the best Move is : %d\n\n",
                              bestVal);
          return bestMove;
}
// Driver code
int main()
{
          char board[3][3] =
          {
                    { 'x', 'o', 'x' },
                    { 'o', 'o', 'x' },
                    { '_', '_', '_' }
          };

          Move bestMove = findBestMove(board);
          printf("The Optimal Move is :\n");
          printf("ROW: %d COL: %d\n\n", bestMove.row,

          bestMove.col );
          return 0;
}
```

```
  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
    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;
}
```

5.MIniMax Alpha Beta Pruning

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

// Initial values of
// Alpha and Beta
const int MAX = 1000;
const int MIN = -1000;

// Returns optimal value for
// current player(Initially called
// for root and maximizer)
int minimax(int depth, int nodeIndex, bool maximizingPlayer, int
values[], int alpha, int beta)
{
          // Terminating condition. i.e
          // leaf node is reached
          if (depth == 3)
                    return values[nodeIndex];
```

6. MIniMax Alpha Beta Pruning 2

```
#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;
  }
}
```

```
        if (maximizingPlayer)
        {
                int best = MIN;
                // Recur for left and
                // right children
                for (int i = 0; i < 2; i++)
                {
                        int val = minimax(depth + 1, nodeIndex
* 2 + i, false, values, alpha, beta);
                        best = max(best, val);
                        alpha = max(alpha, best);
                        // Alpha Beta Pruning
                        if (beta <= alpha) break;
                }
                return best;
        }
        else {
                int best = MAX;
                // Recur for left and
                // right children
                for (int i = 0; i < 2; i++)
                {
                        int val = minimax(depth + 1, nodeIndex
* 2 + i, true, values, alpha, beta);
                        best = min(best, val);
                        beta = min(beta, best);
                        // Alpha Beta Pruning
                        if (beta <= alpha) break;
                }
                return best;
        }
}
// Driver Code
int main()
{
        int values[8] = { 3, 5, 6, 9, 1, 2, 0, -1 };
        cout <<"The optimal value is : "<< minimax(0, 0, true,
values, MIN, MAX);;
        return 0;
}
```

8.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
```

```
// 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++) {
    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)
      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) {
```

```
    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);
    }
    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;
}
```

9. A*

```
//8 puzzle
#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);
    }
};
```

```
                    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;
                    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);

    while (true) {
        int row, col;
```

```
// 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;
}
// 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();
```

```
      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;
      }

      pair<int, int> bestMove = findBestMove(board);
      int bestRow = bestMove.first;
      int bestCol = bestMove.second;

      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;
}
```

10. A*/2

```
//8puzzle print all states using Branch and Bound
#include <bits/stdc++.h>
using namespace std;
#define N 3
struct Node {
   Node* parent; int mat[N][N]; int x, y; int cost; int level;
};
int printMatrix(int mat[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++) printf("%d ", mat[i][j]);
      printf("\n");
   }
}
Node* newNode(int mat[N][N], int x, int y, int newX, int newY, int
level, Node* parent) {
   Node* node = new Node;
```

```
    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;
}
```

```
  node->parent = parent;
  memcpy(node->mat, mat, sizeof node->mat);
  swap(node->mat[x][y], node->mat[newX][newY]);
  node->cost = INT_MAX;
  node->level = level; node->x = newX;
  node->y = newY; return node;
}
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };
int calculateCost(int initial[N][N], int final[N][N]) {
  int count = 0;
  for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
    if (initial[i][j] && initial[i][j] != final[i][j]) count++;
  return count;
}
int isSafe(int x, int y) {
  return (x >= 0 && x < N && y >= 0 && y < N);
}
void printPath(Node* root) {
  if (root == NULL) return;
  printPath(root->parent);
  printMatrix(root->mat); printf("\n");
}
struct comp {
  bool operator()(const Node* lhs, const Node* rhs) const  {
    return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);
  }
};
void solve(int initial[N][N], int x, int y, int final[N][N]) {
  priority_queue<Node*, std::vector<Node*>, comp> pq;
  Node* root = newNode(initial, x, y, x, y, 0, NULL);
  root->cost = calculateCost(initial, final);
  pq.push(root);
  while (!pq.empty()) {
    Node* min = pq.top();
    pq.pop();
    if (min->cost == 0) {
      printPath(min); return;
    }
    for (int i = 0; i < 4; i++) {
      if (isSafe(min->x + row[i], min->y + col[i])) {
        Node* child = newNode(min->mat, min->x, min->y, min-
>x + row[i], min->y + col[i], min->level + 1, min);
        child->cost = calculateCost(child->mat, final);
        pq.push(child);
      }}}}
int main() {
  int initial[N][N] =
  {
    {1, 2, 3},
    {5, 6, 0},
    {7, 8, 4}
  };
  int final[N][N] =
  { {1, 2, 3},
    {5, 8, 6},
```

```
    {0, 7, 4} };
  int x = 1, y = 2;
  solve(initial, x, y, final);
  return 0;
}
```

11. UCS

```
#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;
    }
    // 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});
      }
    }
  }
```

12. 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
    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 << "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;
}
```

13. IDS

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

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

// 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;
```

```
    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;
}
```

14. **Best First Search**

```
// C++ program to implement Best First Search using priority
// queue
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pi;

vector<vector<pi> > graph;

// Function for adding edges to graph
void addedge(int x, int y, int cost)
{
    graph[x].push_back(make_pair(cost, y));
    graph[y].push_back(make_pair(cost, x));
}

// Function For Implementing Best First Search
// Gives output path having lowest cost
void best_first_search(int actual_Src, int target, int n)
{
    vector<bool> visited(n, false);
    // MIN HEAP priority queue
    priority_queue<pi, vector<pi>, greater<pi> > pq;
    // sorting in pq gets done by first value of pair
    pq.push(make_pair(0, actual_Src));
    int s = actual_Src;
    visited[s] = true;
    while (!pq.empty()) {
        int x = pq.top().second;
        // Displaying the path having lowest cost
        cout << x << " ";
        pq.pop();
        if (x == target)
            break;

        for (int i = 0; i < graph[x].size(); i++) {
```

```cpp
}

// 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;
}
```

```cpp
         if (!visited[graph[x][i].second]) {
            visited[graph[x][i].second] = true;
            pq.push(make_pair(graph[x][i].first,graph[x][i].second));
         }
      }
   }
}

// Driver code to test above methods
int main()
{
   // No. of Nodes
   int v = 14;
   graph.resize(v);

   // The nodes shown in above example(by alphabets) are
   // implemented using integers addedge(x,y,cost);
   addedge(0, 1, 3);
   addedge(0, 2, 6);
   addedge(0, 3, 5);
   addedge(1, 4, 9);
   addedge(1, 5, 8);
   addedge(2, 6, 12);
   addedge(2, 7, 14);
   addedge(3, 8, 7);
   addedge(8, 9, 5);
   addedge(8, 10, 6);
   addedge(9, 11, 1);
   addedge(9, 12, 10);
   addedge(9, 13, 2);

   int source = 0;
   int target = 9;

   // Function call
   best_first_search(source, target, v);

   return 0;
}
```

15. Genetic Algorithm

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>

using namespace std;
```

```cpp
int main() {
   srand(time(0));
   vector<Chromosome> population(POP_SIZE);

   // Initialize population and calculate fitness
   for (auto &chromosome : population) {
```

```
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

// 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];
  }
  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) {
```

```
    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;
}
```

<table>
<tr>
<td>
<pre>
  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;
}
</pre>
</td>
<td></td>
</tr>
</table>