**NON AI**

#include <bits/stdc++.h>

#include <vector>

using namespace std;

int turn;

int board[10] = {2, 2, 2, 2, 2, 2, 2, 2, 2, 2};

// X(1) odd turn = 3;

// O(0) even turn = 5;

void GO(int n)

{

    if (turn % 2 == 0)

    {

        board[n] = 5;

    }

    else

    {

        board[n] = 3;

    }

    cout << "Placing at " << n << endl;

    turn++;

}

int Make2()

{

    if (board[5] == 2)

    {

        return 5;

    }

    int x = 5;

    int arr[] = {2,4,6,8};

    while (x != 2)

    {

        x = rand() % 4;

    }

    cout << "randon no : " << arr[x] << endl;

    return arr[x];

}

int findBlank()

{

  for(int i = 1; i<=9; i++)

  {

    if(board[i] == 2)

    {

      return i;

    }

  }

  return -1;

}

void printBoard()

{

    cout << "-------------" << endl;

    cout <<"| ";

    for (int i = 1; i <= 3; i++)

    {

        if (board[i] == 5)

        {

            cout << "5" << " ";

            cout <<"| ";

        }

        else if (board[i] == 3)

        {

            cout << "3" << " ";

            cout <<"| ";

        }

        else

        {

            cout << board[i] << " ";

            cout <<"| ";

        }

    }

    cout << endl;

    cout << "-------------" << endl;

    cout <<"| ";

    for (int i = 4; i <= 6; i++)

    {

        if (board[i] == 5)

        {

            cout << "5"<< " ";

            cout <<"| ";

        }

        else if (board[i] == 3)

        {

            cout << "3" << " ";

            cout <<"| ";

        }

        else

        {

            cout << board[i] << " ";

            cout <<"| ";

        }

    }

    cout << endl;

    cout << "-------------" << endl;

    cout <<"| ";

    for (int i = 7; i <= 9; i++)

    {

        if (board[i] == 5)

        {

            cout << "5" << " ";

            cout <<"| ";

        }

        else if (board[i] == 3)

        {

            cout << "3" << " ";

            cout <<"| ";

        }

        else

        {

            cout << board[i] << " ";

            cout <<"| ";

        }

    }

    cout << endl;

    cout << "-------------" << endl;

}

int posswin(int p)

{

  int target;

  // x

  if(p == 1)

  {

    target = 18;

  }

  else

  {

    target = 50;

  }

  // row check

  for(int i=1; i<=7; i+=3)

  {

    if(board[i] \* board[i+1] \* board[i+2] == target)

    {

      if(board[i] == 2)

      {

        return i;

      }

      else if(board[i+1] == 2)

      {

        return i+1;

      }

      else

      {

        return i+2;

      }

    }

  }

  // col check

  for(int i=1; i<=3; i++)

  {

    if(board[i] \* board[i+3] \* board[i+6] == target)

    {

      if(board[i] == 2)

      {

        return i;

      }

      else if(board[i+3] == 2)

      {

        return i+3;

      }

      else

      {

        return i+6;

      }

    }

  }

   // left diagonal

    if (board[1] \* board[5] \* board[9] == target)

    {

        if (board[1] == 2)

        {

            return 1;

        }

        if (board[5] == 2)

        {

            return 5;

        }

        else

        {

            return 9;

        }

    }

    // right diagonal

    if (board[3] \* board[5] \* board[7] == target)

    {

        if (board[3] == 2)

        {

            return 3;

        }

        if (board[5] == 2)

        {

            return 5;

        }

        else

        {

            return 7;

        }

    }

    return 0;

}

void TicTacToe()

{

    switch(turn)

    {

      case 1:

        GO(turn);

        printBoard();

        break;

      case 2:

        if(board[5] == 2)

        {

          GO(5);

        }

        else

        {

          GO(1);

        }

        printBoard();

        break;

      case 3:

        if(board[9] == 2)

        {

          GO(9);

        }

        else

        {

          GO(5);

        }

        printBoard();

        break;

      case 4:

        if(posswin(1) != 0)

        {

          GO(posswin(1));

        }

        else

        {

          GO(Make2());

        }

        printBoard();

        break;

      case 5:

        if(posswin(1) != 0)

        {

          GO(posswin(1));

          cout << "X wins";

          return;

        }

        else if(posswin(0) != 0)

        {

          GO(posswin(0));

        }

        else if(board[7] == 2)

        {

          GO(7);

        }

        else

        {

          GO(3);

        }

        printBoard();

        break;

      case 6:

        if(posswin(0) != 0)

        {

          GO(posswin(0));

          cout << "O wins";

          return;

        }

        else if(posswin(1) != 0)

        {

          GO(posswin(1));

        }

        else

        {

          GO(Make2());

        }

        printBoard();

        break;

      case 7:

         if(posswin(1) != 0)

        {

          GO(posswin(1));

          cout << "X wins";

          return;

        }

        else if(posswin(0) != 0)

        {

          GO(posswin(0));

        }

        else

        {

          int i = findBlank();

          GO(i);

        }

        printBoard();

        break;

      case 8:

        if(posswin(0) != 0)

        {

          GO(posswin(0));

          cout << "O wins";

          return;

        }

        else if(posswin(1) != 0)

        {

          GO(posswin(1));

        }

        else

        {

          int i = findBlank();

          GO(i);

        }

        printBoard();

        break;

      case 9:

        if(posswin(1) != 0)

        {

          GO(posswin(1));

          cout << "X wins";

          return;

        }

        else if(posswin(0) != 0)

        {

          GO(posswin(0));

        }

        else

        {

          int i = findBlank();

          GO(i);

        }

        printBoard();

        break;

      default:

        break;

    }

}

int main()

{

  turn  = 1;

  cout << "Before : "<< endl;

  printBoard();

  while(turn <= 9)

  {

    TicTacToe();

  }

    return 0;

}

**NON AI H2C**

#include <iostream>

#include <vector>

using namespace std;

vector<int> board(10, 2);

int turn = 1;

int possWin(int player) {

    int winningMove = 0;

    for (int i = 1; i <= 9; i += 3) {

        int product = board[i] \* board[i + 1] \* board[i + 2];

        if (product == player \* player \* 2) {

            if (board[i] == 2) winningMove = i;

            else if (board[i + 1] == 2) winningMove = i + 1;

            else if (board[i + 2] == 2) winningMove = i + 2;

        }

    }

    for (int i = 1; i <= 3; i++) {

        int product = board[i] \* board[i + 3] \* board[i + 6];

        if (product == player \* player \* 2) {

            if (board[i] == 2) winningMove = i;

            else if (board[i + 3] == 2) winningMove = i + 3;

            else if (board[i + 6] == 2) winningMove = i + 6;

        }

    }

    if ((board[1] \* board[5] \* board[9] == player \* player \* 2) && (board[1] == 2 || board[5] == 2 || board[9]==2))

        winningMove = 1;

    else if ((board[3] \* board[5] \* board[7] == player \* player \* 2) && (board[3] == 2 || board[5] == 2 || board[7]==2))

        winningMove = 3;

    return winningMove;

}

void go(int n) {

    if (turn % 2 == 1)

        board[n] = 3;

    else

        board[n] = 5;

    turn++;

}

void makeMove() {

    int move = 0;

    if (board[5] == 2)

        move = 5;

    else {

        for (int i = 2; i <= 8; i += 2) {

            if (board[i] == 2) {

                move = i;

                break;

            }

        }

    }

    if (move == 0) {

        for (int i = 1; i <= 9; i++) {

            if (board[i] == 2) {

                move = i;

                break;

            }

        }

    }

    go(move);

}

void printBoard() {

    for (int i = 1; i <= 9; i++) {

        if (i % 3 != 0)

            cout << board[i] << " | ";

        else

            cout << board[i] << endl;

    }

}

int main() {

    while (turn <= 9) {

        cout << "Turn: " << turn << endl;

        printBoard();

        if (turn % 2 == 1){

            cout << "Player X's turn." << endl;

            cout<<"Choose your desired position(1-9) : ";

            int hmove;

            cin>>hmove;

            go(hmove);

        }

        else{

            cout << "Player O's turn." << endl;

            makeMove();

        }

        int winner = possWin((turn % 2 == 1) ? 3 : 5);

        if (winner != 0) {

            cout << endl;

            printBoard();

            if (turn % 2 == 1)

                cout << "Player X wins!" << endl;

            else

                cout << "Player O wins!" << endl;

            break;

        }

    }

    if (turn > 9) {

        cout << "The game ended in a draw!" << endl;

    }

    return 0;

}

**AI –**

#include <bits/stdc++.h>

#include <vector>

using namespace std;

int size = 3;

bool isHumanTurn = true;

bool hasGameStarted = false;

const char Human = 'X';

const char Computer = 'O';

vector<vector<char>> board(size, vector<char>(size, ' '));

bool isBoardFull()

{

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            if (isdigit(board[i][j]))

            {

                return false;

            }

        }

    }

    return true;

}

bool hasPlayerWon(char player)

{

    // row check

    for (int i = 0; i < size; i++)

    {

        if (board[i][0] == player && board[i][1] == player && board[i][2] == player)

        {

            return true;

        }

    }

    // col check

    for (int j = 0; j < size; j++)

    {

        if (board[0][j] == player && board[1][j] == player && board[2][j] == player)

        {

            return true;

        }

    }

    // digonals check

    if (board[0][0] == player && board[1][1] == player && board[2][2] == player)

    {

        return true;

    }

    if (board[0][2] == player && board[1][1] == player && board[2][0] == player)

    {

        return true;

    }

}

bool isGameOver()

{

    if (isBoardFull() || hasPlayerWon(Human) || hasPlayerWon(Computer))

    {

        return true;

    }

    return false;

}

bool isValidMove(int position)

{

    int row = (position - 1) / size;

    int col = (position - 1) % size;

    if (position < 1 || position > size \* size)

    {

        cout << "Invalid move" << endl;

        return false;

    }

    if (!isdigit(board[row][col]))

    {

        cout << "Already occupied" << endl;

        return false;

    }

    return true;

}

void makeHumanMove()

{

    int position;

    do

    {

        cout << "Enter the position : ";

        cin >> position;

    } while (!isValidMove(position));

    int row = (position - 1) / size;

    int col = (position - 1) % size;

    board[row][col] = Human;

}

int evaluteBoard()

{

    if (hasPlayerWon(Computer))

    {

        return 1;

    }

    else if (hasPlayerWon(Human))

    {

        return -1;

    }

    else

    {

        return 0;

    }

}

vector<int> minimax(int depth, char player)

{

    // row, col, score

    vector<int> bestMove = {-1, -1, 0};

    int bestScore;

    if (player == Computer)

    {

        bestScore = -10;

    }

    else

    {

        bestScore = 10;

    }

    if (isGameOver())

    {

        int score = evaluteBoard();

        return {-1, -1, score};

    }

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            if (isdigit(board[i][j]))

            {

                board[i][j] = player;

                vector<int> currentMove = minimax(depth + 1, (player == Computer) ? Human : Computer);

                int currentScore = currentMove[2];

                board[i][j] = ('0' + i \* size + j + 1);

                // cout << board[i][j] << endl;

                if (player == Computer)

                {

                    if (currentScore > bestScore)

                    {

                        bestScore = currentScore;

                        bestMove[0] = i;

                        bestMove[1] = j;

                    }

                }

                else

                {

                    if (currentScore < bestScore)

                    {

                        bestScore = currentScore;

                        bestMove[0] = i;

                        bestMove[1] = j;

                    }

                }

            }

        }

    }

    bestMove[2] = bestScore;

    return bestMove;

}

void makeComputerMove()

{

    vector<int> bestMove = minimax(0, Computer);

    int position = bestMove[0] \* size + bestMove[1] + 1;

    board[bestMove[0]][bestMove[1]] = Computer;

    cout << "Computer chose position : " << position << endl;

}

void printBoard()

{

    cout << "-------------" << endl;

    for (int i = 0; i < size; i++)

    {

        cout << "| ";

        for (int j = 0; j < size; j++)

        {

            if (isHumanTurn && board[i][j] != Human && board[i][j] != Computer)

            {

                cout << (hasGameStarted ? ' ' : board[i][j]) << " | ";

            }

            else

            {

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

            }

        }

        cout << endl

             << "-------------" << endl;

    }

}

void printResult()

{

    if (hasPlayerWon(Human))

    {

        cout << "Congratulations! You won!" << endl;

    }

    else if (hasPlayerWon(Computer))

    {

        cout << "Computer wins!" << endl;

    }

    else

    {

        cout << "It's a draw!" << endl;

    }

}

void TicTacToe()

{

    char val = '1';

    for (int i = 0; i < size; i++)

    {

        for (int j = 0; j < size; j++)

        {

            board[i][j] = val++;

        }

    }

    while (!isGameOver())

    {

        printBoard();

        if (isHumanTurn)

        {

            makeHumanMove();

        }

        else

        {

            makeComputerMove();

        }

        isHumanTurn = !isHumanTurn;

        hasGameStarted = true;

    }

    printBoard();

    printResult();

}

int main()

{

    TicTacToe();

    return 0;

}

**BFS WATER JUG**

// working

#include <bits/stdc++.h>

#include <vector>

#include <map>

#include <queue>

using namespace std;

void printSolution(map<pair<int, int>, pair<int, int>> mp, pair<int, int> u)

{

    if (u.first == 0 && u.second == 0)

    {

        cout << 0 << " " << 0 << endl;

        return;

    }

    printSolution(mp, mp[u]);

    cout << u.first << " " << u.second << endl;

}

void bfs(int a, int b, int target, int &count)

{

    map<pair<int, int>, bool> visited;

    bool canSolve = false;

    map<pair<int, int>, pair<int, int>> mp;

    queue<pair<int, int>> q;

    q.push({0, 0});

    while (!q.empty())

    {

        auto u = q.front();

        q.pop();

        if (visited[u] == true)

        {

            continue;

        }

        if (u.first > a || u.second > b || u.first < 0 || u.second < 0)

        {

            continue;

        }

        visited[{u.first, u.second}] = true;

        cout << u.first << " " << u.second << endl;

        count++;

        // if (u.first == target) // 2nd solution

        if (u.first == target || u.second == target)

        {

            canSolve = true;

            cout << "Solved"<< endl;

            printSolution(mp, u);

            if (u.first == target)

            {

                if (u.second != 0)

                {

                    cout << u.first << " " << 0 << endl;

                    count++;

                }

            }

            else

            {

                if (u.first != 0)

                {

                    cout << 0 << " " << u.second << endl;

                    count++;

                }

                cout << u.second << " " << 0 << endl;

                    count++;

            }

            return;

        }

        // fill jug a

        if (visited[{a, u.second}] != true)

        {

            q.push({a, u.second});

            mp[{a, u.second}] = u;

        }

        // fill jug b

        if (visited[{u.first, b}] != true)

        {

            q.push({u.first, b});

            mp[{u.first, b}] = u;

        }

        // empty a

        if (visited[{0, u.second}] != true)

        {

            q.push({0, u.second});

            mp[{0, u.second}] = u;

        }

        // empty b

        if (visited[{u.first, 0}] != true)

        {

            q.push({u.first, 0});

            mp[{u.first, 0}] = u;

        }

        // transfer a to b

        int x = b - u.second;

        if (u.first > x)

        {

            int c = u.first - x;

            if (visited[{c, b}] != true)

            {

                q.push({c, b});

                mp[{c, b}] = u;

            }

        }

        else

        {

            int c = u.first + u.second;

            if (visited[{0, c}] != true)

            {

                q.push({0, c});

                mp[{0, c}] = u;

            }

        }

        // tranfer b to a

        x = a - u.first;

        if (u.second >= x)

        {

            int c = u.second - x;

            if (visited[{a, c}] != true)

            {

                q.push({a, c});

                mp[{a, c}] = u;

            }

        }

        else

        {

            int c = u.first + u.second;

            if (visited[{c, 0}] != true)

            {

                q.push({c, 0});

                mp[{c, 0}] = u;

            }

        }

    }

    if (!canSolve)

    {

        cout << "No solution";

    }

}

int main()

{

    int a = 4;

    int b = 3;

    int target = 2;

    int count =0;

    cout << "Water jug problem using BFS :" << endl;

    bfs(a, b, target, count);

    cout << "Number of states visited :" << count ;

    return 0;

}

**DFS WATER JUG**

#include <iostream>

#include <map>

using namespace std;

void printSolution(map<pair<int, int>, pair<int, int>> mp, pair<int, int> u) {

    if (u.first == 0 && u.second == 0) {

        cout << 0 << " " << 0 << endl;

        return;

    }

    printSolution(mp, mp[u]);

    cout << u.first << " " << u.second << endl;

}

bool solveRecursive(int a, int b, int target, int curj1, int curj2, map<pair<int, int>, bool>& visited, map<pair<int, int>, pair<int, int>>& mp, int &count) {

    if (curj1 > a || curj2 > b || curj1 < 0 || curj2 < 0) {

        return false;

    }

    if (visited[{curj1, curj2}]) {

        return false;

    }

    visited[{curj1, curj2}] = true;

    cout << curj1 << " " << curj2 << endl;

    count++;

    if (curj1 == target || curj2 == target) {

        cout << "Solved" << endl;

        printSolution(mp, {curj1, curj2});

        if (curj1 == target) {

            if (curj2 != 0) {

                cout << curj1 << " " << 0 << endl;

                count++;

            }

        } else {

            if(curj1 != 0) {

                cout << 0 << " " << curj2 << endl;

                count++;

            }

            cout << curj2 << " " << 0 << endl;

            count++;

        }

        return true;

    }

    // Fill jug1

    if (!visited[{a, curj2}]) {

        mp[{a, curj2}] = {curj1, curj2};

        if (solveRecursive(a, b, target, a, curj2, visited, mp, count)) {

            return true;

        }

    }

    // Fill jug2

    if (!visited[{curj1, b}]) {

        mp[{curj1, b}] = {curj1, curj2};

        if (solveRecursive(a, b, target, curj1, b, visited, mp, count)) {

            return true;

        }

    }

    // Empty jug1

    if (!visited[{0, curj2}]) {

        mp[{0, curj2}] = {curj1, curj2};

        if (solveRecursive(a, b, target, 0, curj2, visited, mp, count)) {

            return true;

        }

    }

    // Empty jug2

    if (!visited[{curj1, 0}]) {

        mp[{curj1, 0}] = {curj1, curj2};

        if (solveRecursive(a, b, target, curj1, 0, visited, mp, count)) {

            return true;

        }

    }

    // Transfer from jug1 to jug2

    int x = b - curj2;

    if (curj1 >= x) {

        int c = curj1 - x;

        if (!visited[{c, b}]) {

            mp[{c, b}] = {curj1, curj2};

            if (solveRecursive(a, b, target, c, b, visited, mp, count)) {

                return true;

            }

        }

    } else {

        int c = curj1 + curj2;

        if (!visited[{0, c}]) {

            mp[{0, c}] = {curj1, curj2};

            if (solveRecursive(a, b, target, 0, c, visited, mp, count)) {

                return true;

            }

        }

    }

    // Transfer from jug2 to jug1

    x = a - curj1;

    if (curj2 >= x) {

        int c = curj2 - x;

        if (!visited[{a, c}]) {

            mp[{a, c}] = {curj1, curj2};

            if (solveRecursive(a, b, target, a, c, visited, mp, count)) {

                return true;

            }

        }

    } else {

        int c = curj1 + curj2;

        if (!visited[{c, 0}]) {

            mp[{c, 0}] = {curj1, curj2};

            if (solveRecursive(a, b, target, c, 0, visited, mp, count)) {

                return true;

            }

        }

    }

    return false;

}

int main() {

    int a = 4;

    int b = 3;

    int target = 2;

    int count = 0;

    map<pair<int, int>, bool> visited;

    map<pair<int, int>, pair<int, int>> mp;

    cout << "Water jug problem using recursive DFS :" << endl;

    bool res = solveRecursive(a, b, target, 0, 0, visited, mp, count);

    if (!res) {

        cout << "No solution";

    }

    cout << "Number of states visited: " << count << endl;

    return 0;

}

**DEPTH LIMIT WATER JUG**

#include <iostream>

#include <map>

using namespace std;

void printSolution(map<pair<int, int>, pair<int, int>> mp, pair<int, int> u) {

    if (u.first == 0 && u.second == 0) {

        cout << 0 << " " << 0 << endl;

        return;

    }

    printSolution(mp, mp[u]);

    cout << u.first << " " << u.second << endl;

}

bool solveDLS(int a, int b, int target, int curj1, int curj2, map<pair<int, int>, bool>& visited, map<pair<int, int>, pair<int, int>>& mp, int depth, int maxDepth, int &count) {

    if (depth > maxDepth) {

        return false; // Reached depth limit

    }

    if (curj1 > a || curj2 > b || curj1 < 0 || curj2 < 0) {

        return false;

    }

    if (visited[{curj1, curj2}]) {

        return false;

    }

    visited[{curj1, curj2}] = true;

    cout << curj1 << " " << curj2 << endl;

    count++;

    if (curj1 == target || curj2 == target) {

        cout << "Solved" << endl;

        printSolution(mp, {curj1, curj2});

        if (curj1 == target) {

            if (curj2 != 0) {

                cout << curj1 << " " << 0 << endl;

                count++;

            }

        } else {

            if(curj1 != 0) {

                cout << 0 << " " << curj2 << endl;

                count++;

            }

            cout << curj2 << " " << 0 << endl;

            count++;

        }

        return true;

    }

    // Fill jug1

    if (!visited[{a, curj2}]) {

        mp[{a, curj2}] = {curj1, curj2};

        if (solveDLS(a, b, target, a, curj2, visited, mp, depth + 1, maxDepth, count)) {

            return true;

        }

    }

    // Fill jug2

    if (!visited[{curj1, b}]) {

        mp[{curj1, b}] = {curj1, curj2};

        if (solveDLS(a, b, target, curj1, b, visited, mp, depth + 1, maxDepth, count)) {

            return true;

        }

    }

    // Empty jug1

    if (!visited[{0, curj2}]) {

        mp[{0, curj2}] = {curj1, curj2};

        if (solveDLS(a, b, target, 0, curj2, visited, mp, depth + 1, maxDepth, count)) {

            return true;

        }

    }

    // Empty jug2

    if (!visited[{curj1, 0}]) {

        mp[{curj1, 0}] = {curj1, curj2};

        if (solveDLS(a, b, target, curj1, 0, visited, mp, depth + 1, maxDepth, count)) {

            return true;

        }

    }

    // Transfer from jug1 to jug2

    int x = b - curj2;

    if (curj1 >= x) {

        int c = curj1 - x;

        if (!visited[{c, b}]) {

            mp[{c, b}] = {curj1, curj2};

            if (solveDLS(a, b, target, c, b, visited, mp, depth + 1, maxDepth, count)) {

                return true;

            }

        }

    } else {

        int c = curj1 + curj2;

        if (!visited[{0, c}]) {

            mp[{0, c}] = {curj1, curj2};

            if (solveDLS(a, b, target, 0, c, visited, mp, depth + 1, maxDepth, count)) {

                return true;

            }

        }

    }

    // Transfer from jug2 to jug1

    x = a - curj1;

    if (curj2 >= x) {

        int c = curj2 - x;

        if (!visited[{a, c}]) {

            mp[{a, c}] = {curj1, curj2};

            if (solveDLS(a, b, target, a, c, visited, mp, depth + 1, maxDepth, count)) {

                return true;

            }

        }

    } else {

        int c = curj1 + curj2;

        if (!visited[{c, 0}]) {

            mp[{c, 0}] = {curj1, curj2};

            if (solveDLS(a, b, target, c, 0, visited, mp, depth + 1, maxDepth, count)) {

                return true;

            }

        }

    }

    return false;

}

int main() {

    int a = 4;

    int b = 3;

    int target = 2;

    int count = 0;

    int maxDepth = 10; // Depth limit

    map<pair<int, int>, bool> visited;

    map<pair<int, int>, pair<int, int>> mp;

    cout << "Water jug problem using Depth Limited Search (DLS):" << endl;

    bool res = solveDLS(a, b, target, 0, 0, visited, mp, 0, maxDepth, count);

    if (!res) {

        cout << "No solution" << endl;

    }

    cout << "Number of states visited: " << count << endl;

    return 0;

}

**ITERATIVE WATER JUG**

#include <iostream>

#include <map>

using namespace std;

void printSolution(map<pair<int, int>, pair<int, int>> mp, pair<int, int> u)

{

    if (u.first == 0 && u.second == 0)

    {

        cout << 0 << " " << 0 << endl;

        return;

    }

    printSolution(mp, mp[u]);

    cout << u.first << " " << u.second << endl;

}

bool solveDLS(int a, int b, int target, int curj1, int curj2, map<pair<int, int>, bool> &visited, map<pair<int, int>, pair<int, int>> &mp, int depth, int maxDepth, int &count)

{

    if (depth > maxDepth)

    {

        return false; // Reached depth limit

    }

    if (curj1 > a || curj2 > b || curj1 < 0 || curj2 < 0)

    {

        return false;

    }

    if (visited[{curj1, curj2}])

    {

        return false;

    }

    visited[{curj1, curj2}] = true;

    cout << curj1 << " " << curj2 << endl;

    count++;

    if (curj1 == target || curj2 == target)

    {

        cout << "Solved" << endl;

        printSolution(mp, {curj1, curj2});

        if (curj1 == target)

        {

            if (curj2 != 0)

            {

                cout << curj1 << " " << 0 << endl;

                count++;

            }

        }

        else

        {

            if (curj1 != 0)

            {

                cout << 0 << " " << curj2 << endl;

                count++;

            }

            cout << curj2 << " " << 0 << endl;

            count++;

        }

        return true;

    }

    // Fill jug1

    if (!visited[{a, curj2}])

    {

        mp[{a, curj2}] = {curj1, curj2};

        if (solveDLS(a, b, target, a, curj2, visited, mp, depth + 1, maxDepth, count))

        {

            return true;

        }

    }

    // Fill jug2

    if (!visited[{curj1, b}])

    {

        mp[{curj1, b}] = {curj1, curj2};

        if (solveDLS(a, b, target, curj1, b, visited, mp, depth + 1, maxDepth, count))

        {

            return true;

        }

    }

    // Empty jug1

    if (!visited[{0, curj2}])

    {

        mp[{0, curj2}] = {curj1, curj2};

        if (solveDLS(a, b, target, 0, curj2, visited, mp, depth + 1, maxDepth, count))

        {

            return true;

        }

    }

    // Empty jug2

    if (!visited[{curj1, 0}])

    {

        mp[{curj1, 0}] = {curj1, curj2};

        if (solveDLS(a, b, target, curj1, 0, visited, mp, depth + 1, maxDepth, count))

        {

            return true;

        }

    }

    // Transfer from jug1 to jug2

    int x = b - curj2;

    if (curj1 >= x)

    {

        int c = curj1 - x;

        if (!visited[{c, b}])

        {

            mp[{c, b}] = {curj1, curj2};

            if (solveDLS(a, b, target, c, b, visited, mp, depth + 1, maxDepth, count))

            {

                return true;

            }

        }

    }

    else

    {

        int c = curj1 + curj2;

        if (!visited[{0, c}])

        {

            mp[{0, c}] = {curj1, curj2};

            if (solveDLS(a, b, target, 0, c, visited, mp, depth + 1, maxDepth, count))

            {

                return true;

            }

        }

    }

    // Transfer from jug2 to jug1

    x = a - curj1;

    if (curj2 >= x)

    {

        int c = curj2 - x;

        if (!visited[{a, c}])

        {

            mp[{a, c}] = {curj1, curj2};

            if (solveDLS(a, b, target, a, c, visited, mp, depth + 1, maxDepth, count))

            {

                return true;

            }

        }

    }

    else

    {

        int c = curj1 + curj2;

        if (!visited[{c, 0}])

        {

            mp[{c, 0}] = {curj1, curj2};

            if (solveDLS(a, b, target, c, 0, visited, mp, depth + 1, maxDepth, count))

            {

                return true;

            }

        }

    }

    return false;

}

bool solveIDDFS(int a, int b, int target, map<pair<int, int>, bool> &visited, map<pair<int, int>, pair<int, int>> &mp, int maxDepth, int &count)

{

    for (int depth = 0; depth <= maxDepth; depth++)

    {

        if (solveDLS(a, b, target, 0, 0, visited, mp, 0, depth, count))

        {

            return true; // Solution found

        }

        // Reset visited and mp for next iteration

        visited.clear();

        mp.clear();

    }

    return false; // Solution not found within max depth

}

int main()

{

    int a = 4;

    int b = 3;

    int target = 2;

    int maxDepth = 10;

    int count = 0;

    map<pair<int, int>, bool> visited;

    map<pair<int, int>, pair<int, int>> mp;

    cout << "Water jug problem using Iterative Deepening Depth First Search (IDDFS):" << endl;

    bool res = solveIDDFS(a, b, target, visited, mp, maxDepth, count);

    if (!res)

    {

        cout << "No solution" << endl;

    }

    cout << "Number of states visited: " << count << endl;

    return 0;

}

**DFS N QUEENS**

#include <bits/stdc++.h>

#include <vector>

using namespace std;

printSolution(vector<vector<int>> &board)

{

    int n = board.size();

    // for (int i = 0; i < n; i++)

    // {

    //     for (int j = 0; j < n; j++)

    //         if (board[i][j] == 1)

    //         {

    //             cout << "Q" << i << "\t";

    //         }

    //         else

    //         {

    //             cout << board[i][j] << "\t";

    //         }

    //     cout << endl;

    // }

    // cout << endl;

    for (int i = 0; i < n; i++) {

        for (int j = 0; j < n; j++) {

            if (board[i][j] == 1) {

                cout << "Q" << i << "\t";

            } else if (board[i][j] == -1) {

                cout << "X" << " "; // Mark the position being checked as 'X'

            } else {

                cout << board[i][j] << "\t";

            }

        }

        cout << endl;

    }

    cout << endl;

    // vector<int> ans(n, 0);

    // for (int i = 0; i < n; i++)

    // {

    //     for (int j = 0; j < n; j++)

    //     {

    //         if (board[i][j] == 1)

    //         {

    //             ans[i] = j+1;

    //         }

    //     }

    // }

    // for (int i = 0; i < n; i++)

    // {

    //     cout << ans[i] << " ";

    // }

}

bool isSafe(vector<vector<int>> &board, int row, int col)

{

    int n = board.size();

    // check for row

    for (int j = 0; j < col; j++)

    {

        if (board[row][j] == 1)

        {

            // cout << "Not safe at (" << row << ", " << col << "):" << endl;

            // printSolution(board);

             // Print the board when a position is not safe

            cout << "Position (" << row << ", " << col << ") is not safe because of row conflict with (" << row << ", " << j << "):" << endl;

            board[row][col] = -1; // Temporarily mark the position

            printSolution(board);

            board[row][col] = 0; // Unmark the position

            return false;

        }

    }

    // check for upper left

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

    {

        if (board[i][j] == 1)

        {

            // cout << "Not safe at (" << row << ", " << col << "):" << endl;

            // printSolution(board);

            // Print the board when a position is not safe

            cout << "Position (" << row << ", " << col << ") is not safe because of upper left diagonal conflict with (" << i << ", " << j << "):" << endl;

            board[row][col] = -1; // Temporarily mark the position

            printSolution(board);

            board[row][col] = 0; // Unmark the position

            return false;

        }

    }

    // check for lower left

    for (int i = row, j = col; i < n && j >= 0; i++, j--)

    {

        if (board[i][j] == 1)

        {

            // cout << "Not safe at (" << row << ", " << col << "):" << endl;

            // printSolution(board);

            // Print the board when a position is not safe

            cout << "Position (" << row << ", " << col << ") is not safe because of lower left diagonal conflict with (" << i << ", " << j << "):" << endl;

            board[row][col] = -1; // Temporarily mark the position

            printSolution(board);

            board[row][col] = 0; // Unmark the position

            return false;

        }

    }

}

bool solveQueens(vector<vector<int>> &board, int col)

{

    int n = board.size();

    if (col >= n)

    {

        cout << "Final result : " << endl;

        printSolution(board);

        return true;

    }

    for (int i = 0; i < n; i++)

    {

        if (isSafe(board, i, col))

        {

            board[i][col] = 1;

            cout << "Trying to place queen at (" << i << ", " << col << "):" << endl;

            printSolution(board);

            if (solveQueens(board, col + 1))

            {

                return true;

            }

            cout << "Backtracking from (" << i << ", " << col << "):" << endl;

            printSolution(board);

            board[i][col] = 0;

        }

    }

    return false;

}

int main()

{

    int n;

    cout << "Enter the number of queens : ";

    cin >> n;

    vector<vector<int>> board(n, vector<int>(n, 0));

    if (!solveQueens(board, 0))

    {

        cout << "No solution";

    }

    return 0;

}

**BFS MNC**

#include <bits/stdc++.h>

#include <vector>

#include <unordered\_set>

#include <queue>

using namespace std;

using State = vector<vector<int>>;

// check valid state or not

bool isValid(State &state)

{

    int ml = state[0][0];

    int cl = state[0][1];

    int bl = state[0][2];

    int mr = state[1][0];

    int cr = state[1][1];

    int br = state[1][2];

    if (ml < 0 || cl < 0 || mr < 0 || cr < 0)

    {

        return false;

    }

    if (ml > 0 && ml < cl || mr > 0 && mr < cr)

    {

        return false;

    }

    return true;

}

vector<State> getsuccessor(State &state)

{

    vector<State> Successors;

    // boat is on left

    if (state[0][2] == 1)

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] - i, state[0][1] - j, 0},

                        {state[1][0] + i, state[1][1] + j, 1}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    else

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] + i, state[0][1] + j, 1},

                        {state[1][0] - i, state[1][1] - j, 0}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    return Successors;

}

vector<State> bfs(State &intial, State &goal)

{

    if (!isValid(intial))

    {

        return {};

    }

    unordered\_set<int> visited;

    queue<vector<State>> q;

    q.push({intial});

    while (!q.empty())

    {

        vector<State> path = q.front();

        q.pop();

        State currentState = path.back();

        int hash = currentState[0][0] \* 10000 + currentState[0][1] \* 1000 + currentState[0][2] \* 100 + currentState[1][0] \* 10 + currentState[1][1];

        if (visited.find(hash) != visited.end())

        {

            continue;

        }

        visited.insert(hash);

        if (currentState == goal)

        {

            return path;

        }

        for (auto &successor : getsuccessor(currentState))

        {

            vector<State> newPath = path;

            newPath.push\_back(successor);

            q.push(newPath);

        }

    }

    return {};

}

// void printState(State &state)

// {

//     cout << "Left Side: (";

//     for (int i = 0; i < state[0][0]; ++i) cout << "M";

//     cout << ",";

//     for (int i = 0; i < state[0][1]; ++i) cout << "C";

//     cout << ", " << (state[0][2] == 1 ? "Boat" : "No Boat") << ") ";

//     cout << "Right Side: (";

//     for (int i = 0; i < state[1][0]; ++i) cout << "M";

//     cout << ",";

//     for (int i = 0; i < state[1][1]; ++i) cout << "C";

//     cout << ", " << (state[1][2] == 1 ? "Boat" : "No Boat") << ")";

// }

void printSolution(vector<State> &solution)

{

    for (int i = 0; i < solution.size(); i++)

    {

        cout << "At depth " << i << ": Left Side ("

             << solution[i][0][0] << ", " << solution[i][0][1] << ", " << solution[i][0][2]

             << "), Right Side : ("

             << solution[i][1][0] << ", " << solution[i][1][1] << ", " << solution[i][1][2]

             << ")" << endl;

    }

    //  for (int i = 0; i < solution.size(); i++)

    // {

    //     cout << "At depth " << i << ": ";

    //     printState(solution[i]);

    //     cout << endl;

    // }

}

int main()

{

    State initial = {{3, 3, 1}, {0, 0, 0}};

    State goal = {{0, 0, 0}, {3, 3, 1}};

    vector<State> solution = bfs(initial, goal);

    cout << "Using bfs" << endl;

    if (solution.empty())

    {

        cout << "No solution";

    }

    else

    {

        printSolution(solution);

    }

    return 0;

}

**DFS MNC**

#include <bits/stdc++.h>

#include <vector>

#include <unordered\_set>

using namespace std;

using State = vector<vector<int>>;

bool isValid(State &state)

{

    int ml = state[0][0];

    int cl = state[0][1];

    int bl = state[0][2];

    int mr = state[1][0];

    int cr = state[1][1];

    int br = state[1][2];

    if (ml < 0 || cl < 0 || mr < 0 || cr < 0)

    {

        return false;

    }

    if (ml > 0 && ml < cl || mr > 0 && mr < cr)

    {

        return false;

    }

    return true;

}

vector<State> getSuccessor(State &state)

{

    vector<State> Successors;

    // boat is on left

    if (state[0][2] == 1)

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] - i, state[0][1] - j, 0},

                        {state[1][0] + i, state[1][1] + j, 1}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    else

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] + i, state[0][1] + j, 1},

                        {state[1][0] - i, state[1][1] - j, 0}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    return Successors;

}

bool dfs(State &current, State &goal, unordered\_set<int> &visited, vector<State> &path)

{

    if (current == goal)

    {

        return true;

    }

    for (auto &successor : getSuccessor(current))

    {

        int hash = successor[0][0] \* 10000 + successor[0][1] \* 1000 + successor[0][2] \* 100 + successor[1][0] \* 10 + successor[1][1];

        if (visited.find(hash) == visited.end())

        {

            visited.insert(hash);

            path.push\_back(successor);

            if (dfs(successor, goal, visited, path))

            {

                return true;

            }

            path.pop\_back();

        }

    }

    return false;

}

void printSolution(vector<State> &solution)

{

    for (int i = 0; i < solution.size(); i++)

    {

        cout << "At depth " << i << ": Left Side ("

             << solution[i][0][0] << ", " << solution[i][0][1] << ", " << solution[i][0][2]

             << "), Right Side : ("

             << solution[i][1][0] << ", " << solution[i][1][1] << ", " << solution[i][1][2]

             << ")" << endl;

    }

}

int main()

{

    State initial = {{3, 3, 1}, {0, 0, 0}};

    State goal = {{0, 0, 0}, {3, 3, 1}};

    vector<State> path;

    unordered\_set<int> visited;

    path.push\_back(initial);

    int hash = initial[0][0] \* 10000 + initial[0][1] \* 1000 +

    initial[0][2] \* 100 + initial[1][0] \* 10 + initial[1][1];

    visited.insert(hash);

    cout << "Using dfs :" << endl;

    if (dfs(initial, goal, visited,path))

    {

        printSolution(path);

    }

    else

    {

        cout << "No solution";

    }

    return 0;

}

**DEPTH LIMIT MNC**

#include <bits/stdc++.h>

#include <vector>

#include <unordered\_set>

using namespace std;

using State = vector<vector<int>>;

bool isValid(State &state)

{

    int ml = state[0][0];

    int cl = state[0][1];

    int bl = state[0][2];

    int mr = state[1][0];

    int cr = state[1][1];

    int br = state[1][2];

    if (ml < 0 || cl < 0 || mr < 0 || cr < 0)

    {

        return false;

    }

    if (ml > 0 && ml < cl || mr > 0 && mr < cr)

    {

        return false;

    }

    return true;

}

vector<State> getSuccessor(State &state)

{

    vector<State> Successors;

    // boat is on left

    if (state[0][2] == 1)

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] - i, state[0][1] - j, 0},

                        {state[1][0] + i, state[1][1] + j, 1}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    else

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] + i, state[0][1] + j, 1},

                        {state[1][0] - i, state[1][1] - j, 0}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    return Successors;

}

bool dfs(State &current, State &goal, unordered\_set<int> &visited, vector<State> &path, int depth, int limit, int &count)

{

    if (depth > limit)

    {

        return false;

    }

    if (current == goal)

    {

        return true;

    }

    for (auto &successor : getSuccessor(current))

    {

        int hash = successor[0][0] \* 10000 + successor[0][1] \* 1000 + successor[0][2] \* 100 + successor[1][0] \* 10 + successor[1][1];

        if (visited.find(hash) == visited.end())

        {

            visited.insert(hash);

            count++;

            path.push\_back(successor);

            if (dfs(successor, goal, visited, path, depth + 1, limit, count))

            {

                return true;

            }

            path.pop\_back();

        }

    }

    return false;

}

void printSolution(vector<State> &solution)

{

    for (int i = 0; i < solution.size(); i++)

    {

        cout << "At depth " << i << ": Left Side ("

             << solution[i][0][0] << ", " << solution[i][0][1] << ", " << solution[i][0][2]

             << "), Right Side : ("

             << solution[i][1][0] << ", " << solution[i][1][1] << ", " << solution[i][1][2]

             << ")" << endl;

    }

}

int main()

{

    State initial = {{3, 3, 1}, {0, 0, 0}};

    State goal = {{0, 0, 0}, {3, 3, 1}};

    vector<State> path;

    unordered\_set<int> visited;

    path.push\_back(initial);

    int hash = initial[0][0] \* 10000 + initial[0][1] \* 1000 +

               initial[0][2] \* 100 + initial[1][0] \* 10 + initial[1][1];

    visited.insert(hash);

    cout << "Using dfs :" << endl;

    int limit = 15; // Set the depth limit

    int depth = 0;

    int count = 0;

    if (dfs(initial, goal, visited, path, depth, limit, count))

    {

        cout << "Number of states visited: " << count << endl;

        printSolution(path);

    }

    else

    {

        cout << "No solution found within depth limit of " << limit << endl;

    }

    return 0;

}

**ITERATIVE MNC**

#include <bits/stdc++.h>

#include <vector>

#include <unordered\_set>

using namespace std;

using State = vector<vector<int>>;

bool isValid(State &state)

{

    int ml = state[0][0];

    int cl = state[0][1];

    int bl = state[0][2];

    int mr = state[1][0];

    int cr = state[1][1];

    int br = state[1][2];

    if (ml < 0 || cl < 0 || mr < 0 || cr < 0)

    {

        return false;

    }

    if (ml > 0 && ml < cl || mr > 0 && mr < cr)

    {

        return false;

    }

    return true;

}

vector<State> getSuccessor(State &state)

{

    vector<State> Successors;

    // boat is on left

    if (state[0][2] == 1)

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] - i, state[0][1] - j, 0},

                        {state[1][0] + i, state[1][1] + j, 1}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    else

    {

        for (int i = 0; i <= 2; ++i)

        {

            for (int j = 0; j <= 2; ++j)

            {

                if (i + j > 0 && i + j <= 2)

                {

                    State successor = {

                        {state[0][0] + i, state[0][1] + j, 1},

                        {state[1][0] - i, state[1][1] - j, 0}};

                    if (isValid(successor))

                    {

                        Successors.push\_back(successor);

                    }

                }

            }

        }

    }

    return Successors;

}

bool dls(State &current, State &goal, unordered\_set<int> &visited, vector<State> &path, int depth, int limit, int &count)

{

    if (depth > limit)

    {

        return false;

    }

    if (current == goal)

    {

        return true;

    }

    for (auto &successor : getSuccessor(current))

    {

        int hash = successor[0][0] \* 10000 + successor[0][1] \* 1000 + successor[0][2] \* 100 + successor[1][0] \* 10 + successor[1][1];

        if (visited.find(hash) == visited.end())

        {

            visited.insert(hash);

            count++;

            path.push\_back(successor);

            if (dls(successor, goal, visited, path, depth + 1, limit, count))

            {

                return true;

            }

            path.pop\_back();

        }

    }

    return false;

}

bool iddfs(State &initial, State &goal, int maxDepth, vector<State> &solution, int &count)

{

    for (int limit = 0; limit <= maxDepth; ++limit)

    {

        unordered\_set<int> visited;

        vector<State> path;

        path.push\_back(initial);

        int hash = initial[0][0] \* 10000 + initial[0][1] \* 1000 +

                   initial[0][2] \* 100 + initial[1][0] \* 10 + initial[1][1];

        visited.insert(hash);

        // cout << "Depth limit: " << limit << endl;

        if (dls(initial, goal, visited, path, 0, limit, count))

        {

            solution = path;

            return true;

        }

    }

    return false;

}

void printSolution(vector<State> &solution)

{

    for (int i = 0; i < solution.size(); i++)

    {

        cout << "At depth " << i << ": Left Side ("

             << solution[i][0][0] << ", " << solution[i][0][1] << ", " << solution[i][0][2]

             << "), Right Side : ("

             << solution[i][1][0] << ", " << solution[i][1][1] << ", " << solution[i][1][2]

             << ")" << endl;

    }

}

int main()

{

    State initial = {{3, 3, 1}, {0, 0, 0}};

    State goal = {{0, 0, 0}, {3, 3, 1}};

    vector<State> solution;

    cout << "Using IDDFS:" << endl;

    int maxDepth = 15; // Set the maximum depth limit

    int count  = 0;

    if (iddfs(initial, goal, maxDepth, solution, count))

    {

        printSolution(solution);

    }

    else

    {

        cout << "No solution found within depth limit of " << maxDepth << endl;

    }

    cout << "Number of states visited: " << count << endl;

    return 0;

}

**BFS 8 PUZZLE**

#include <bits/stdc++.h>

#include <vector>

using namespace std;

void findEmpty(vector<vector<int>> &board, int &x, int &y)

{

    for (int i = 0; i < board.size(); i++)

    {

        for (int j = 0; j < board.size(); j++)

        {

            if (board[i][j] == 0)

            {

                x = i;

                y = j;

                return;

            }

        }

    }

}

vector<pair<vector<vector<int>>, string>> findAdj(vector<vector<int>> &state)

{

    vector<pair<vector<vector<int>>, string>> neighbours;

    int x = -1, y = -1;

    findEmpty(state, x, y);

    int dx[] = {0, 0, -1, 1};

    int dy[] = {1, -1, 0, 0};

    string moves[] = {"right", "left", "up", "down"};

    for (int i = 0; i < 4; i++)

    {

        int newX = x + dx[i];

        int newY = y + dy[i];

        if (newX >= 0 && newX < state.size() && newY >= 0 && newY < state.size())

        {

            vector<vector<int>> neighbourState = state;

            swap(neighbourState[x][y], neighbourState[newX][newY]);

            neighbours.push\_back(make\_pair(neighbourState, moves[i]));

        }

    }

    return neighbours;

}

vector<pair<vector<vector<int>>, string>> bfs(vector<vector<int>> &initial, vector<vector<int>> &goal)

{

    unordered\_set<string> visited;

    queue<vector<vector<int>>> q;

    int count = 0;

    queue<vector<pair<vector<vector<int>>, string>>> paths;

    q.push(initial);

    paths.push({});

    while (!q.empty())

    {

        auto current = q.front();

        q.pop();

        auto path = paths.front();

        paths.pop();

        if (current == goal)

        {

            cout << "Number of stated visited : " << count << endl;

            return path;

        }

        string vis = "";

        for (auto &row : current)

        {

            for (int num : row)

            {

                vis += to\_string(num);

            }

        }

        if (visited.find(vis) != visited.end())

        {

            continue;

        }

        visited.insert(vis);

        count++;

        for (auto neighbourPair : findAdj(current))

        {

            auto neighbour = neighbourPair.first;

            auto move = neighbourPair.second;

            string neigh = "";

            for (auto &row : neighbour)

            {

                for (int num : row)

                {

                    neigh += to\_string(num);

                }

            }

            if (visited.find(neigh) == visited.end())

            {

                q.push(neighbour);

                auto newPath = path;

                newPath.push\_back(make\_pair(neighbour, move));

                paths.push(newPath);

            }

        }

    }

    return {};

}

int main()

{

    vector<vector<int>> initial = {{2, 8, 3},

                                   {1, 6, 4},

                                   {7, 0, 5}};

    vector<vector<int>> goal = {{1, 2, 3},

                                {8, 0, 4},

                                {7, 6, 5}};

    // vector<vector<int>> initial = {{1, 2, 3},

    //                                {8, 0, 4},

    //                                {7, 6, 5}};

    // vector<vector<int>> goal = {{2, 8, 1},

    //                             {0, 4, 3},

    //                             {7, 6, 5}};

    vector<pair<vector<vector<int>>, string>> result = bfs(initial, goal);

    // vector<pair<vector<vector<int>>, string>> solution = result.first;

    // int statesExplored = result.second;

    if (!result.empty())

    {

        cout << "Solution found:" << endl;

        for (auto &row : initial)

        {

            for (auto &num : row)

            {

                cout << num << " ";

            }

            cout << endl;

        }

        cout << endl;

        for (auto stateActionPair : result)

        {

            auto state = stateActionPair.first;

            auto action = stateActionPair.second;

            cout << "Action: " << action << endl;

            for (auto &row : state)

            {

                for (int num : row)

                {

                    cout << num << " ";

                }

                cout << endl;

            }

            cout << endl;

        }

    }

    else

    {

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

    }

    return 0;

}

**DFS 8 PUZZLE**

#include <bits/stdc++.h>

using namespace std;

void findEmpty(vector<vector<int>> &board, int &x, int &y) {

    for (int i = 0; i < board.size(); i++) {

        for (int j = 0; j < board.size(); j++) {

            if (board[i][j] == 0) {

                x = i;

                y = j;

                return;

            }

        }

    }

}

vector<pair<vector<vector<int>>, string>> findAdj(vector<vector<int>> &state) {

    vector<pair<vector<vector<int>>, string>> neighbours;

    int x = -1, y = -1;

    findEmpty(state, x, y);

    int dx[] = {0, 0, -1, 1};

    int dy[] = {1, -1, 0, 0};

    string moves[] = {"right", "left", "up", "down"};

    for (int i = 0; i < 4; i++) {

        int newX = x + dx[i];

        int newY = y + dy[i];

        if (newX >= 0 && newX < state.size() && newY >= 0 && newY < state.size()) {

            vector<vector<int>> neighbourState = state;

            swap(neighbourState[x][y], neighbourState[newX][newY]);

            neighbours.push\_back(make\_pair(neighbourState, moves[i]));

        }

    }

    return neighbours;

}

pair<vector<pair<vector<vector<int>>, string>>, int> dfs(vector<vector<int>> &initial, vector<vector<int>> &goal) {

    unordered\_set<string> visited;

    stack<vector<vector<int>>> s;

    s.push(initial);

    stack<vector<pair<vector<vector<int>>, string>>> paths;

    paths.push({});

    int statesExplored = 0;

    while (!s.empty()) {

        auto current = s.top();

        s.pop();

        auto path = paths.top();

        paths.pop();

        if (current == goal) {

            return make\_pair(path, statesExplored);

        }

        string vis = "";

        for (auto &row : current) {

            for (int num : row) {

                vis += to\_string(num);

            }

        }

        if (visited.find(vis) != visited.end()) {

            continue;

        }

        visited.insert(vis);

        statesExplored++;

        for (auto neighbourPair : findAdj(current)) {

            auto neighbour = neighbourPair.first;

            auto move = neighbourPair.second;

            string neigh = "";

            for (auto &row : neighbour) {

                for (int num : row) {

                    neigh += to\_string(num);

                }

            }

            if (visited.find(neigh) == visited.end()) {

                s.push(neighbour);

                auto newPath = path;

                newPath.push\_back(make\_pair(neighbour, move));

                paths.push(newPath);

            }

        }

    }

    return make\_pair(vector<pair<vector<vector<int>>, string>>(), statesExplored);

}

int main() {

    vector<vector<int>> initial = {{2, 8, 3},

                                   {1, 6, 4},

                                   {7, 0, 5}};

    vector<vector<int>> goal = {{1, 2, 3},

                                {8, 0, 4},

                                {7, 6, 5}};

    pair<vector<pair<vector<vector<int>>, string>>, int> result = dfs(initial, goal);

    vector<pair<vector<vector<int>>, string>> solution = result.first;

    int statesExplored = result.second;

    if (!solution.empty()) {

        cout << "Solution found:" << endl;

        for (auto &row : initial) {

            for (auto &num : row) {

                cout << num << " ";

            }

            cout << endl;

        }

        cout << endl;

        for (auto stateActionPair : solution) {

            auto state = stateActionPair.first;

            auto action = stateActionPair.second;

            cout << "Action: " << action << endl;

            for (auto &row : state) {

                for (int num : row) {

                    cout << num << " ";

                }

                cout << endl;

            }

            cout << endl;

        }

    } else {

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

    }

    cout << "Number of states explored: " << statesExplored << endl;

    return 0;

}

**DFS1 8 PUZZLE**

#include <iostream>

#include <vector>

#include <unordered\_set>

#include <cstring>

using namespace std;

#define N 3

struct Node {

    Node\* parent;

    int board[N][N];

    int x, y;

    string action;

    int depth;

};

void findEmpty(int board[N][N], int& x, int& y) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            if (board[i][j] == 0) {

                x = i;

                y = j;

                return;

            }

        }

    }

}

Node\* newNode(int board[N][N], int x, int y, int newX, int newY, string action, int depth, Node\* parent) {

    Node\* node = new Node;

    memcpy(node->board, board, sizeof(node->board));

    node->x = newX;

    node->y = newY;

    swap(node->board[x][y], node->board[newX][newY]);

    node->depth = depth;

    node->action = action;

    node->parent = parent;

    return node;

}

bool isEqual(int mat1[N][N], int mat2[N][N]) {

    for (int i = 0; i < N; i++)

        for (int j = 0; j < N; j++)

            if (mat1[i][j] != mat2[i][j])

                return false;

    return true;

}

void printBoard(int board[N][N]) {

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

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

        }

        cout << endl;

    }

    cout << endl;

}

void printPath(Node\* node) {

    if (node == nullptr)

        return;

    printPath(node->parent);

    if (!node->action.empty()) {

        cout << "Move: " << node->action << endl;

    }

    printBoard(node->board);

}

bool dfs(Node\* node, int goal[N][N], unordered\_set<string>& visited, int maxDepth) {

    if (node == nullptr || node->depth > maxDepth)

        return false;

    if (isEqual(node->board, goal)) {

        printPath(node);

        cout << "Depth: " << node->depth << endl;

        return true;

    }

    string state;

    for (int i = 0; i < N; i++) {

        for (int j = 0; j < N; j++) {

            state += to\_string(node->board[i][j]);

        }

    }

    if (visited.find(state) != visited.end()) {

        return false;

    }

    visited.insert(state);

    int dx[] = {0, 0, -1, 1};

    int dy[] = {1, -1, 0, 0};

    string moves[] = {"right", "left", "up", "down"};

    for (int i = 0; i < 4; i++) {

        int newX = node->x + dx[i];

        int newY = node->y + dy[i];

        if (newX >= 0 && newX < N && newY >= 0 && newY < N) {

            Node\* adj = newNode(node->board, node->x, node->y, newX, newY, moves[i], node->depth + 1, node);

            if (dfs(adj, goal, visited, maxDepth)) {

                // delete adj; // Free memory after exploring

                return true;

            }

            // delete adj;

        }

    }

    return false;

}

void solve(int board[N][N], int goal[N][N], int x, int y, int maxDepth) {

    unordered\_set<string> visited;

    Node\* root = newNode(board, x, y, x, y, "", 0, nullptr);

    if (!dfs(root, goal, visited, maxDepth)) {

        cout << "No solution within the specified depth limit." << endl;

    }

}

int main() {

    int initial[N][N] = {{2, 8, 3},

                         {1, 6, 4},

                         {7, 0, 5}};

    int goal[N][N] = {{1, 2, 3},

                      {8, 0, 4},

                      {7, 6, 5}};

    int x, y;

    findEmpty(initial, x, y);

    int maxDepth = 10; // Set maximum depth for DFS

    solve(initial, goal, x, y, maxDepth);

    return 0;

}

**BEST FIRST 8 PUZZLE**

// new with struct - working

#include<bits/stdc++.h>

#include <cstring>

#include <iostream>

#include<vector>

#include <algorithm>

#include <queue>

using namespace std;

#define N 3

struct Node

{

    Node\* parent;

    int board[N][N];

    int depth;

    int h;

    int x, y;

    string action;

};

void findEmpty(int board[N][N], int &x, int &y)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (board[i][j] == 0)

            {

                x = i;

                y = j;

                return;

            }

        }

    }

}

int heuristics(int current[N][N], int goal[N][N])

{

    int h = 0;

    for (int i = 0; i <N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (current[i][j] != 0 && current[i][j] != goal[i][j])

            {

                h++;

            }

        }

    }

    return h;

}

bool isGoalState(int board[N][N], int goal[N][N])

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (board[i][j] != goal[i][j])

                return false;

        }

    }

    return true;

}

void printBoard(int board[N][N])

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

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

        }

        cout << endl;

    }

    cout << endl;

}

void printPath(Node \*node)

{

    if (node == nullptr)

        return;

    printPath(node->parent);

    if (!node->action.empty())

    {

        cout << "Move: " << node->action << endl;

    }

    printBoard(node->board);

}

Node \*newNode(int board[N][N], int x, int y, int newX, int newY, int depth, Node\* parent, string action)

{

    Node \* node = new Node;

    memcpy(node->board, board, sizeof(node->board));

    node->x = newX;

    node->y = newY;

    swap(node->board[x][y], node->board[newX][newY]);

    node->depth = depth;

    node->parent = parent;

    node->action = action;

    return node;

}

struct Comparator

{

    bool operator()(Node \*n1, Node \*n2)

    {

        return (n1->h > n2->h);

    }

};

void solve(int board[N][N], int goal[N][N], int x, int y)

{

    priority\_queue<Node\*, vector<Node\*>, Comparator>

        pq;

    int statesExplored = 0;

    int count = 0;

    Node\* root = newNode(board, x,y,x,y,0,nullptr,"");

    root->h = heuristics(board, goal);

    pq.push(root);

    cout << "h : " << root->h << endl;

    printBoard(board);

    count++;

    while (!pq.empty())

    {

        Node \*current = pq.top();

        pq.pop();

        statesExplored++;

        cout << "Expanding node : " << endl;

        printBoard(current->board);

        findEmpty(current->board, x, y);

        if (isGoalState(current->board, goal))

        {

            cout << "Goal state reached" << endl;

            printPath(current);

            cout << "Depth: " << current->depth << endl;

            cout << "Cost: " << current->depth -1 << endl;

            cout << "Visited: " << count << endl;

            cout << "Number of states explored: " << statesExplored << endl;

            return;

        }

        int dx[] = {0, 0, -1, 1};

        int dy[] = {1, -1, 0, 0};

        string moves[] = {"right", "left", "up", "down"};

        for (int i = 0; i < 4; i++)

        {

            int newX = current->x + dx[i];

            int newY = current->y + dy[i];

            if (newX >= 0 && newX < N && newY >= 0 && newY < N)

            {

                Node \*adj = newNode(current->board, current->x, current->y, newX, newY, current->depth+1, current, moves[i]);

                adj->h =  heuristics(adj->board, goal);

                pq.push(adj);

                cout << "h : " << adj->h << endl;

                printBoard(current->board);

                count++;

            }

        }

    }

    cout << "No solution";

    return;

}

int main()

{

     int initial[N][N] = {{2, 8, 3},

                         {1, 6, 4},

                         {7, 0, 5}};

    int goal[N][N] = {{1, 2, 3},

                      {8, 0, 4},

                      {7, 6, 5}};

    int x,y;

    findEmpty(initial,x,y);

    solve(initial, goal, x,y);

    return 0;

}

**A\* 8 PUZZLE**

// new with struct - working

#include<bits/stdc++.h>

#include <cstring>

#include <iostream>

#include<vector>

#include <algorithm>

#include <queue>

using namespace std;

#define N 3

struct Node

{

    Node\* parent;

    int board[N][N];

    int depth;

    int h;

    int x, y;

    string action;

};

void findEmpty(int board[N][N], int &x, int &y)

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (board[i][j] == 0)

            {

                x = i;

                y = j;

                return;

            }

        }

    }

}

int heuristics(int current[N][N], int goal[N][N])

{

    int h = 0;

    for (int i = 0; i <N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (current[i][j] != 0 && current[i][j] != goal[i][j])

            {

                h++;

            }

        }

    }

    return h;

}

bool isGoalState(int board[N][N], int goal[N][N])

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

            if (board[i][j] != goal[i][j])

                return false;

        }

    }

    return true;

}

void printBoard(int board[N][N])

{

    for (int i = 0; i < N; i++)

    {

        for (int j = 0; j < N; j++)

        {

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

        }

        cout << endl;

    }

    cout << endl;

}

void printPath(Node \*node)

{

    if (node == nullptr)

        return;

    printPath(node->parent);

    if (!node->action.empty())

    {

        cout << "Move: " << node->action << endl;

    }

    printBoard(node->board);

}

Node \*newNode(int board[N][N], int x, int y, int newX, int newY, int depth, Node\* parent, string action)

{

    Node \* node = new Node;

    memcpy(node->board, board, sizeof(node->board));

    node->x = newX;

    node->y = newY;

    swap(node->board[x][y], node->board[newX][newY]);

    node->depth = depth;

    node->parent = parent;

    node->action = action;

    return node;

}

struct Comparator

{

    bool operator()(Node \*n1, Node \*n2)

    {

        return (n1->h > n2->h);

    }

};

void solve(int board[N][N], int goal[N][N], int x, int y)

{

    priority\_queue<Node\*, vector<Node\*>, Comparator>

        pq;

    int statesExplored = 0;

    int count = 0;

    Node\* root = newNode(board, x,y,x,y,0,nullptr,"");

    root->h = heuristics(board, goal);

    pq.push(root);

    cout << "h : " << root->h << endl;

    printBoard(board);

    count++;

    while (!pq.empty())

    {

        Node \*current = pq.top();

        pq.pop();

        statesExplored++;

        cout << "Expanding node : " << endl;

        printBoard(current->board);

        findEmpty(current->board, x, y);

        if (isGoalState(current->board, goal))

        {

            cout << "Goal state reached" << endl;

            printPath(current);

            cout << "Depth: " << current->depth << endl;

            cout << "Cost: " << current->depth -1 << endl;

            cout << "Visited: " << count << endl;

            cout << "Number of states explored: " << statesExplored << endl;

            return;

        }

        int dx[] = {0, 0, -1, 1};

        int dy[] = {1, -1, 0, 0};

        string moves[] = {"right", "left", "up", "down"};

        for (int i = 0; i < 4; i++)

        {

            int newX = current->x + dx[i];

            int newY = current->y + dy[i];

            if (newX >= 0 && newX < N && newY >= 0 && newY < N)

            {

                Node \*adj = newNode(current->board, current->x, current->y, newX, newY, current->depth+1, current, moves[i]);

                adj->h = adj->depth + heuristics(adj->board, goal);

                pq.push(adj);

                cout << "h : " << adj->h << endl;

                printBoard(current->board);

                count++;

            }

        }

    }

    cout << "No solution";

    return;

}

int main()

{

     int initial[N][N] = {{2, 8, 3},

                         {1, 6, 4},

                         {7, 0, 5}};

    int goal[N][N] = {{1, 2, 3},

                      {8, 0, 4},

                      {7, 6, 5}};

    int x,y;

    findEmpty(initial,x,y);

    solve(initial, goal, x,y);

    return 0;

}

**BEST FIRST ROUTE**

// working

#include <bits/stdc++.h>

#include <vector>

#include <string>

#include <map>

#include <queue>

using namespace std;

class Node

{

public:

    string name;

    int heuristic;

    vector<pair<Node \*, int>> adj;

    Node(string n, int h) : name(n), heuristic(h) {}

};

vector<string> bestFirst(Node \*start, Node \*goal, map<Node \*, int> &distances)

{

    priority\_queue<pair<int, Node \*>, vector<pair<int, Node \*>>,

                   greater<pair<int, Node \*>>>

        pq;

    map<Node \*, int> visited;

    map<Node \*, Node \*> parent;

    pq.push({start->heuristic, start});

    distances[start] = 0;

    while (!pq.empty())

    {

        Node \*current = pq.top().second;

        int currentH = pq.top().first;

        pq.pop();

        cout << current->name << " " << currentH << endl;

        if (current == goal)

        {

            vector<string> path;

            int distance = distances[current];

            while (current != start)

            {

                path.push\_back(current->name);

                current = parent[current];

            }

            path.push\_back(start->name);

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

            cout << "Distance : " << distance << endl;

            return path;

        }

        visited[current] = currentH;

        for (auto &edge : current->adj)

        {

            Node \*neighbourNode = edge.first;

            int neighbourH = neighbourNode->heuristic;

            int edgeWeight = edge.second;

            int newDistance = distances[current] + edgeWeight;

            if (visited.find(neighbourNode) == visited.end())

            {

                pq.push({neighbourH, neighbourNode});

                parent[neighbourNode] = current;

                distances[neighbourNodea] = newDistance;

            }

        }

    }

    cout << "Not reachable" << endl;

    return vector<string>();

}

int main()

{

    Node \*n1 = new Node("Arad", 366);

    Node \*n2 = new Node("Zerind", 374);

    Node \*n3 = new Node("Oradea", 380);

    Node \*n4 = new Node("Sibiu", 253);

    Node \*n5 = new Node("Timisoara", 329);

    Node \*n6 = new Node("Lugoj", 244);

    Node \*n7 = new Node("Mehadia", 241);

    Node \*n8 = new Node("Craiova", 160);

    Node \*n9 = new Node("Drobeta", 242);

    Node \*n10 = new Node("Eforie", 161);

    Node \*n11 = new Node("Fagaras", 176);

    Node \*n12 = new Node("Giurgiu", 77);

    Node \*n13 = new Node("Bucharest", 0);

    Node \*n14 = new Node("Hirsova", 151);

    Node \*n15 = new Node("Iasi", 226);

    Node \*n16 = new Node("Neamt", 234);

    Node \*n17 = new Node("Pitesti", 98);

    Node \*n18 = new Node("Rimnicu Vilcea", 193);

    Node \*n19 = new Node("Vaslui", 199);

    Node \*n20 = new Node("Urziceni", 80);

    n1->adj = {{n2, 75}, {n4, 140}, {n5, 118}};

    n2->adj = {{n1, 75}, {n3, 71}};

    n3->adj = {{n2, 71}, {n4, 151}};

    n4->adj = {{n1, 140}, {n11, 99}, {n3, 151}, {n18, 80}};

    n5->adj = {{n1, 118}, {n6, 111}};

    n6->adj = {{n5, 111}, {n7, 70}};

    n7->adj = {{n6, 70}, {n9, 75}};

    n8->adj = {{n9, 120}, {n18, 146}, {n17, 138}};

    n9->adj = {{n7, 75}, {n8, 120}};

    n10->adj = {{n14, 86}};

    n11->adj = {{n4, 99}, {n13, 211}};

    n12->adj = {{n13, 90}};

    n13->adj = {{n12, 90}, {n17, 101}, {n20, 85}};

    n14->adj = {{n10, 86}, {n20, 98}};

    n15->adj = {{n16, 87}, {n19, 92}};

    n16->adj = {{n15, 87}};

    n17->adj = {{n18, 97}, {n13, 101}, {n8, 138}};

    n18->adj = {{n4, 80}, {n17, 97}, {n8, 146}};

    n19->adj = {{n15, 92}, {n20, 152}};

    n20->adj = {{n19, 142}, {n14, 98}, {n13, 85}};

    map<Node \*, int> distances;

    cout << "Using Best first search algorithm - " << endl;

    vector<string> path = bestFirst(n1, n13, distances);

    // Print the path

    cout << "Path: ";

    for (const auto &city : path)

    {

        cout << city << " -> ";

    }

    cout << endl;

    return 0;

}

**ROUTE A\***

// working

#include <bits/stdc++.h>

#include <vector>

#include <string>

#include <map>

#include <queue>

using namespace std;

class Node

{

public:

    string name;

    int heuristic;

    vector<pair<Node \*, int>> adj;

    Node(string n, int h) : name(n), heuristic(h) {}

};

vector<string> bestFirst(Node \*start, Node \*goal, map<Node \*, int> &distances)

{

    priority\_queue<pair<int, Node \*>, vector<pair<int, Node \*>>,

                   greater<pair<int, Node \*>>>

        pq;

    map<Node \*, int> visited;

    map<Node \*, Node \*> parent;

    pq.push({start->heuristic, start});

    distances[start] = 0;

    while (!pq.empty())

    {

        Node \*current = pq.top().second;

        int currentH = pq.top().first;

        pq.pop();

        cout << current->name << " " << currentH << endl;

        if (current == goal)

        {

            vector<string> path;

            int distance = distances[current];

            while (current != start)

            {

                path.push\_back(current->name);

                current = parent[current];

            }

            path.push\_back(start->name);

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

            cout << "Distance : " << distance << endl;

            return path;

        }

        visited[current] = currentH;

        for (auto &edge : current->adj)

        {

            Node \*neighbourNode = edge.first;

            int neighbourH = neighbourNode->heuristic;

            int edgeWeight = edge.second;

            int newDistance = distances[current] + edgeWeight;

            int totalCost = newDistance + neighbourNode->heuristic;

            cout << totalCost << endl;

            if (visited.find(neighbourNode) == visited.end())

            {

                pq.push({totalCost, neighbourNode});

                parent[neighbourNode] = current;

                distances[neighbourNode] = newDistance;

            }

        }

    }

    cout << "Not reachable" << endl;

    return vector<string>();

}

int main()

{

    Node \*n1 = new Node("Arad", 366);

    Node \*n2 = new Node("Zerind", 374);

    Node \*n3 = new Node("Oradea", 380);

    Node \*n4 = new Node("Sibiu", 253);

    Node \*n5 = new Node("Timisoara", 329);

    Node \*n6 = new Node("Lugoj", 244);

    Node \*n7 = new Node("Mehadia", 241);

    Node \*n8 = new Node("Craiova", 160);

    Node \*n9 = new Node("Drobeta", 242);

    Node \*n10 = new Node("Eforie", 161);

    Node \*n11 = new Node("Fagaras", 176);

    Node \*n12 = new Node("Giurgiu", 77);

    Node \*n13 = new Node("Bucharest", 0);

    Node \*n14 = new Node("Hirsova", 151);

    Node \*n15 = new Node("Iasi", 226);

    Node \*n16 = new Node("Neamt", 234);

    Node \*n17 = new Node("Pitesti", 98);

    Node \*n18 = new Node("Rimnicu Vilcea", 193);

    Node \*n19 = new Node("Vaslui", 199);

    Node \*n20 = new Node("Urziceni", 80);

    n1->adj = {{n2, 75}, {n4, 140}, {n5, 118}};

    n2->adj = {{n1, 75}, {n3, 71}};

    n3->adj = {{n2, 71}, {n4, 151}};

    n4->adj = {{n1, 140}, {n11, 99}, {n3, 151}, {n18, 80}};

    n5->adj = {{n1, 118}, {n6, 111}};

    n6->adj = {{n5, 111}, {n7, 70}};

    n7->adj = {{n6, 70}, {n9, 75}};

    n8->adj = {{n9, 120}, {n18, 146}, {n17, 138}};

    n9->adj = {{n7, 75}, {n8, 120}};

    n10->adj = {{n14, 86}};

    n11->adj = {{n4, 99}, {n13, 211}};

    n12->adj = {{n13, 90}};

    n13->adj = {{n12, 90}, {n17, 101}, {n20, 85}};

    n14->adj = {{n10, 86}, {n20, 98}};

    n15->adj = {{n16, 87}, {n19, 92}};

    n16->adj = {{n15, 87}};

    n17->adj = {{n18, 97}, {n13, 101}, {n8, 138}};

    n18->adj = {{n4, 80}, {n17, 97}, {n8, 146}};

    n19->adj = {{n15, 92}, {n20, 152}};

    n20->adj = {{n19, 142}, {n14, 98}, {n13, 85}};

    map<Node \*, int> distances;

    cout << "Using Best first search algorithm - " << endl;

    vector<string> path = bestFirst(n1, n13, distances);

    // Print the path

    cout << "Path: ";

    for (const auto &city : path)

    {

        cout << city << " -> ";

    }

    cout << endl;

    return 0;

}

**HILL CLIMBING**

#include <iostream>

#include <vector>

#include <cmath>

#include <limits>

using namespace std;

struct Point {

    int x, y;

};

double distance(Point a, Point b) {

    return sqrt((a.x - b.x) \* (a.x - b.x) + (a.y - b.y) \* (a.y - b.y));

}

// Function to generate neighbors of a given point

vector<Point> getNeighbors(Point current, int gridSize) {

    vector<Point> neighbors;

    if (current.x + 1 < gridSize) {

        neighbors.push\_back({current.x + 1, current.y});

    }

    if (current.x - 1 >= 0) {

        neighbors.push\_back({current.x - 1, current.y});

    }

    if (current.y + 1 < gridSize) {

        neighbors.push\_back({current.x, current.y + 1});

    }

    if (current.y - 1 >= 0) {

        neighbors.push\_back({current.x, current.y - 1});

    }

    return neighbors;

}

// Function to perform hill climbing

vector<Point> hillClimbing(Point start, Point goal, int gridSize, const vector<vector<int>> &grid) {

    vector<Point> path;

    Point current = start;

    while (current.x != goal.x || current.y != goal.y) {

        double minDistance = distance(current, goal);

        Point nextMove = current;

        // Generate neighbors

        vector<Point> neighbors = getNeighbors(current, gridSize);

        cout << "Distance to goal : " << minDistance << endl;

        // Evaluate neighbors

        cout << "Exploring " << current.x << " " << current.y << endl;

        for (const auto &neighbor : neighbors) {

            if (grid[neighbor.x][neighbor.y] == 1) {

                cout << "obstacle : " << neighbor.x <<" " << neighbor.y << endl;

                continue; // Skip obstacles

            }

            double dist = distance(neighbor, goal);

            cout << "neighbours : " << neighbor.x << " " << neighbor.y << " distance : " << dist << endl;

            if (dist < minDistance) {

                minDistance = dist;

                nextMove = neighbor;

            }

        }

        // If no better move is found, break (stuck in a local minimum)

        if (nextMove.x == current.x && nextMove.y == current.y) {

            cout << "Stuck in local minimum at (" << current.x << ", " << current.y << ")" << endl;

            break;

        }

        // Move to the next best position

        current = nextMove;

        path.push\_back(current);

    }

    return path;

}

void printGrid(vector<vector<int>> grid)

{

    int gridSize = grid.size();

    for (int i = 0; i < gridSize; i++)

    {

        for (int j = 0; j < gridSize; j++)

        {

            cout << grid[i][j] << " ";

        }

        cout << endl;

    }

    cout << endl;

}

int main() {

    Point start = {0, 0};

    Point goal = {4, 4};

    int gridSize = 5;

    // Create a grid with obstacles

    vector<vector<int>> grid(gridSize, vector<int>(gridSize, 0));

    grid[2][1] = 1; // Obstacle at (2,1)

    grid[2][2] = 1; // Obstacle at (2,2)

    grid[2][3] = 1; // Obstacle at (2,3)

    printGrid(grid);

    vector<Point> path = hillClimbing(start, goal, gridSize, grid);

    cout << "Path: ";

    for (const auto &point : path) {

        cout << "(" << point.x << ", " << point.y << ") ";

    }

    cout << endl;

    return 0;

}

**CSP N QUEENS**

#include <bits/stdc++.h>

#include <vector>

using namespace std;

printSolution(vector<vector<int>> &board)

{

    int n = board.size();

    for (int i = 0; i < n; i++)

    {

        for (int j = 0; j < n; j++)

            if (board[i][j] == 1)

            {

                cout << "Q" << i << "\t";

            }

            else

            {

                cout << board[i][j] << "\t";

            }

        cout << endl;

    }

    cout << endl;

    // vector<int> ans(n, 0);

    // for (int i = 0; i < n; i++)

    // {

    //     for (int j = 0; j < n; j++)

    //     {

    //         if (board[i][j] == 1)

    //         {

    //             ans[i] = j+1;

    //         }

    //     }

    // }

    // for (int i = 0; i < n; i++)

    // {

    //     cout << ans[i] << " ";

    // }

}

bool isSafe(vector<vector<int>> &board, int row, int col)

{

    int n = board.size();

    // check for row

    for (int j = 0; j < col; j++)

    {

        if (board[row][j] == 1)

        {

            return false;

        }

    }

    // check for upper left

    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)

    {

        if (board[i][j] == 1)

        {

            return false;

        }

    }

    // check for lower left

    for (int i = row, j = col; i < n && j >= 0; i++, j--)

    {

        if (board[i][j] == 1)

        {

            return false;

        }

    }

}

void solveQueens(vector<vector<int>> &board, int col, int &count)

{

    int n = board.size();

    if (col >= n)

    {

       return;

    }

    for (int i = 0; i < n; i++)

    {

        if (isSafe(board, i, col))

        {

            board[i][col] = 1;

           if (col == n - 1)

           {

                cout << "Solution " << count+1 << endl;

                printSolution(board);

                count++;

           }

            solveQueens(board, col + 1, count);

            board[i][col] = 0;

        }

    }

}

int main()

{

    int n;

    cout << "Enter the number of queens : ";

    cin >> n;

    int count  = 0;

    vector<vector<int>> board(n, vector<int>(n, 0));

    solveQueens(board,0, count);

    if (count == 0)

    {

        cout << "No solution";

    }

    return 0;

}

**CSP CRYPTARTHMATIC**

// single solution

// only 1 valid solution

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <string>

#include <cmath>

#include <set>

using namespace std;

int evaluateTerm(const string &term, const unordered\_map<char, int> &mp)

{

    int value = 0;

    for (char ch : term)

    {

        value = value \* 10 + mp.at(ch);

    }

    return value;

}

bool backtrack(int i, const set<char> &uniqueLetters, unordered\_map<char, int> &mp, vector<bool> &used, const vector<string> &words, const string &result)

{

    // cout << i << endl;

    if (i == uniqueLetters.size())

    {

        int value1 = evaluateTerm(words[0], mp);

        int value2 = evaluateTerm(words[1], mp);

        int value3 = evaluateTerm(result, mp);

        return value1 + value2 == value3;

    }

    char currentLetter = \*next(uniqueLetters.begin(), i);

    for (int digit = 0; digit <= 9; digit++)

    {

        if (!used[digit])

        {

            mp[currentLetter] = digit;

            used[digit] = true;

            if (backtrack(i + 1, uniqueLetters, mp, used, words, result))

            {

                return true;

            }

            used[digit] = false;

            mp[currentLetter] = -1; // Reset the assignment

        }

    }

    return false;

}

void solve(const vector<string> &words, const string &result)

{

    unordered\_map<char, int> mp;

    set<char> uniqueLetters;

    for (const string &s : words)

    {

        for (char ch : s)

        {

            if (isalpha(ch) && uniqueLetters.find(ch) == uniqueLetters.end())

            {

                uniqueLetters.insert(ch);

            }

        }

    }

    for (char ch : result)

    {

        if (isalpha(ch) && uniqueLetters.find(ch) == uniqueLetters.end())

        {

            uniqueLetters.insert(ch);

        }

    }

    vector<bool> used(10, false); // Indicates if a digit is used

    if (backtrack(0, uniqueLetters, mp, used, words, result))

    {

        cout << "Solution found:" << endl;

        for (char letter : uniqueLetters)

        {

            cout << letter << " = " << mp[letter] << endl;

        }

    }

    else

    {

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

    }

}

int main()

{

    string s1, s2, result;

    cout << "Enter string 1 : ";

    cin >> s1;

    cout << "Enter string 2 : ";

    cin >> s2;

    cout << "Enter result string : ";

    cin >> result;

    vector<string> words = {s1, s2};

    solve(words, result);

    return 0;

}

**CSP CRYPARTHMATIC 1**

// multiple solutions

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <string>

#include <cmath>

#include <set>

using namespace std;

int evaluateTerm(const string &term, const unordered\_map<char, int> &mp) {

    int value = 0;

    for (char ch : term) {

        value = value \* 10 + mp.at(ch);

    }

    return value;

}

bool backtrack(int i, const set<char> &uniqueLetters, unordered\_map<char, int> &mp, vector<bool> &used, const vector<string> &words, const string &result, vector<unordered\_map<char, int>> &solutions) {

    if (i == uniqueLetters.size()) {

        int value1 = evaluateTerm(words[0], mp);

        int value2 = evaluateTerm(words[1], mp);

        int value3 = evaluateTerm(result, mp);

        if (value1 + value2 == value3) {

            solutions.push\_back(mp);

            return false;  // Continue searching for other solutions

        }

        return false;

    }

    char currentLetter = \*next(uniqueLetters.begin(), i);

    for (int digit = 0; digit <= 9; digit++) {

        if (!used[digit]) {

            mp[currentLetter] = digit;

            used[digit] = true;

            backtrack(i + 1, uniqueLetters, mp, used, words, result, solutions);

            used[digit] = false;

            mp[currentLetter] = -1; // Reset the assignment

        }

    }

    return false;

}

void solve(const vector<string> &words, const string &result) {

    unordered\_map<char, int> mp;

    set<char> uniqueLetters;

    for (const string &s : words) {

        for (char ch : s) {

            if (isalpha(ch)) {

                uniqueLetters.insert(ch);

            }

        }

    }

    for (char ch : result) {

        if (isalpha(ch)) {

            uniqueLetters.insert(ch);

        }

    }

    vector<bool> used(10, false); // Indicates if a digit is used

    vector<unordered\_map<char, int>> solutions;

    backtrack(0, uniqueLetters, mp, used, words, result, solutions);

    if (!solutions.empty()) {

        cout << "Number of solutions : " << solutions.size() << endl;

        cout << "Solutions found:" << endl;

        for (const auto &solution : solutions) {

            for (char letter : uniqueLetters) {

                cout << letter << " = " << solution.at(letter) << ", ";

            }

            cout << endl;

        }

    } else {

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

    }

}

int main() {

    string s1, s2, result;

    cout << "Enter string 1: ";

    cin >> s1;

    cout << "Enter string 2: ";

    cin >> s2;

    cout << "Enter result string: ";

    cin >> result;

    vector<string> words = {s1, s2};

    solve(words, result);

    return 0;

}

**AO STAR CPP**

#include <bits/stdc++.h>

using namespace std;

struct Condition {

    vector<string> AND;

    vector<string> OR;

};

struct Node {

    string name;

    int heuristic;

    Condition condition;

};

map<string, int> calculateCost(map<string, int>& heuristics, Condition& condition, int weight) {

    map<string, int> cost;

    if (!condition.AND.empty()) {

        int totalCost = 0;

        for (const string& node : condition.AND) {

            totalCost += heuristics[node] + weight;

        }

        string path = "AND " + accumulate(condition.AND.begin(), condition.AND.end(), string(),

                                          [](const string& a, const string& b) { return a + (a.length() > 0 ? " AND " : "") + b; });

        cost[path] = totalCost;

    }

    if (!condition.OR.empty()) {

        int minCost = INT\_MAX;

        string bestNode;

        for (const string& node : condition.OR) {

            int currentCost = heuristics[node] + weight;

            if (currentCost < minCost) {

                minCost = currentCost;

                bestNode = node;

            }

        }

        cost["OR " + bestNode] = minCost;

    }

    return cost;

}

map<string, map<string, int>> updateCost(map<string, int>& heuristics, map<string, Condition>& conditions, int weight) {

    vector<string> mainNodes;

    for (const auto& pair : conditions) {

        mainNodes.push\_back(pair.first);

    }

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

    map<string, map<string, int>> leastCost;

    for (const string& key : mainNodes) {

        Condition& condition = conditions[key];

        map<string, int> cost = calculateCost(heuristics, condition, weight);

        heuristics[key] = min\_element(cost.begin(), cost.end(),

                                      [](const pair<string, int>& a, const pair<string, int>& b) { return a.second < b.second; })

                              ->second;

        leastCost[key] = cost;

    }

    return leastCost;

}

string shortestPath(const string& start, map<string, map<string, int>>& updatedCost, map<string, int>& heuristics) {

    string path = start;

    if (updatedCost.find(start) != updatedCost.end()) {

        auto& costs = updatedCost[start];

        auto minCost = min\_element(costs.begin(), costs.end(),

                                   [](const pair<string, int>& a, const pair<string, int>& b) { return a.second < b.second; });

        string nextPath = minCost->first;

        stringstream ss(nextPath);

        string segment;

        vector<string> nodes;

        while (getline(ss, segment, ' ')) {

            if (segment != "AND" && segment != "OR") {

                nodes.push\_back(segment);

            }

        }

        if (nodes.size() == 1) {

            path += "<--" + shortestPath(nodes[0], updatedCost, heuristics);

        } else {

            path += "<--(" + nextPath + ") [" + shortestPath(nodes[0], updatedCost, heuristics) + " + " +

                    shortestPath(nodes[1], updatedCost, heuristics) + "]";

        }

    }

    return path;

}

int main() {

    map<string, int> heuristics = {

        {"A", -1}, {"B", 5}, {"C", 2}, {"D", 4}, {"E", 7}, {"F", 9},

        {"G", 3},  {"H", 0}, {"I", 0}, {"J", 0}};

    map<string, Condition> conditions;

    conditions["A"].OR = {"B"};

    conditions["A"].AND = {"C", "D"};

    conditions["B"].OR = {"E", "F"};

    conditions["C"].OR = {"G"};

    conditions["C"].AND = {"H", "I"};

    conditions["D"].OR = {"J"};

    int weight = 1;

    cout << "Updated Cost :" << endl;

    map<string, map<string, int>> updatedCost = updateCost(heuristics, conditions, weight);

    for (const auto& pair : updatedCost) {

        cout << pair.first << ": ";

        for (const auto& cost : pair.second) {

            cout << cost.first << " => " << cost.second << ", ";

        }

        cout << endl;

    }

    cout << string(75, '\*') << endl;

    cout << "Shortest Path :" << endl;

    cout << shortestPath("A", updatedCost, heuristics) << endl;

    return 0;

}

**AO STAR**

import java.util.\*;

public class trial {

    public static Map<String, Integer> Cost(Map<String, Integer> H,

            Map<String, List<String>> condition, int weight) {

        Map<String, Integer> cost = new HashMap<>();

        if (condition.containsKey("AND")) {

            List<String> AND\_nodes = condition.get("AND");

            String Path\_A = String.join(" AND ", AND\_nodes);

            int PathA = AND\_nodes.stream()

                    .mapToInt(

                            node -> H.get(node) + weight)

                    .sum();

            cost.put(Path\_A, PathA);

        }

        if (condition.containsKey("OR")) {

            List<String> OR\_nodes = condition.get("OR");

            String Path\_B = String.join(" OR ", OR\_nodes);

            int PathB = OR\_nodes.stream()

                    .mapToInt(

                            node -> H.get(node) + weight)

                    .min()

                    .getAsInt();

            cost.put(Path\_B, PathB);

        }

        return cost;

    }

    public static Map<String, Map<String, Integer>> UpdateCost(

            Map<String, Integer> H,

            Map<String, Map<String, List<String>>> Conditions,

            int weight) {

        List<String> Main\_nodes = new ArrayList<>(Conditions.keySet());

        Collections.reverse(Main\_nodes);

        Map<String, Map<String, Integer>> least\_cost = new HashMap<>();

        for (String key : Main\_nodes) {

            Map<String, List<String>> condition = Conditions.get(key);

            System.out.printf("%s: %s >>> %s%n", key,

                    condition,

                    Cost(H, condition, weight));

            Map<String, Integer> c = Cost(H, condition, weight);

            H.put(key, Collections.min(c.values()));

            least\_cost.put(key, Cost(H, condition, weight));

        }

        return least\_cost;

    }

    public static String ShortestPath(

            String Start,

            Map<String, Map<String, Integer>> Updated\_cost,

            Map<String, Integer> H) {

        String Path = Start;

        if (Updated\_cost.containsKey(Start)) {

            int Min\_cost = Collections.min(

                    Updated\_cost.get(Start).values());

            List<String> key = new ArrayList<>(

                    Updated\_cost.get(Start).keySet());

            List<Integer> values = new ArrayList<>(

                    Updated\_cost.get(Start).values());

            int Index = values.indexOf(Min\_cost);

            List<String> Next = Arrays.asList(key.get(Index).split(" "));

            if (Next.size() == 1) {

                Start = Next.get(0);

                Path += "<--"

                        + ShortestPath(Start, Updated\_cost,

                                H);

            } else {

                Path += "<--(" + key.get(Index) + ") ";

                Start = Next.get(0);

                Path += "["

                        + ShortestPath(Start, Updated\_cost,

                                H)

                        + " + ";

                Start = Next.get(Next.size() - 1);

                Path += ShortestPath(Start, Updated\_cost, H)

                        + "]";

            }

        }

        return Path;

    }

    public static void main(String[] args) {

        Map<String, Integer> H = new HashMap<>();

        H.put("A", -1);

        H.put("B", 5);

        H.put("C", 2);

        H.put("D", 4);

        H.put("E", 7);

        H.put("F", 9);

        H.put("G", 3);

        H.put("H", 0);

        H.put("I", 0);

        H.put("J", 0);

        Map<String, Map<String, List<String>>> Conditions = new HashMap<>();

        Map<String, List<String>> aConditions = new HashMap<>();

        aConditions.put("OR", Arrays.asList("B"));

        aConditions.put("AND", Arrays.asList("C", "D"));

        Conditions.put("A", aConditions);

        Map<String, List<String>> bConditions = new HashMap<>();

        bConditions.put("OR", Arrays.asList("E", "F"));

        Conditions.put("B", bConditions);

        Map<String, List<String>> cConditions = new HashMap<>();

        cConditions.put("OR", Arrays.asList("G"));

        cConditions.put("AND", Arrays.asList("H", "I"));

        Conditions.put("C", cConditions);

        Map<String, List<String>> dConditions = new HashMap<>();

        dConditions.put("OR", Arrays.asList("J"));

        Conditions.put("D", dConditions);

        // weight

        int weight = 1;

        // Updated cost

        System.out.println("Updated Cost :");

        Map<String, Map<String, Integer>> Updated\_cost = UpdateCost(H, Conditions, weight);

        System.out.println("\*".repeat(75));

        System.out.println("Shortest Path :");

        System.out.println(

                ShortestPath("A", Updated\_cost, H));

    }

}