**AI Assignment 2**

Sharvari Deshmukh

TY CSA 73

**Problem Statement**- Implementation of Uninformed strategies.

1. **Water Jug Problem:**

**Theory-**

The Water Jug problem typically involves two jugs of different capacities and a target quantity that needs to be measured. The objective is to find a sequence of operations (e.g., filling, emptying, pouring) that leads to obtaining the desired quantity in one of the jugs.

1. **BFS:** Starting from the initial state, BFS explores all possible states reachable from it in a level-by-level manner until the goal state is found. It maintains a queue data structure to store the states to be explored next.

#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> pii;

void printpath(map<pii, pii> mp, pii u)

{

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

    {

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

        return;

    }

    printpath(mp, mp[u]);

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

}

void BFS(int a, int b, int target)

{

    map<pii, int> m;

    bool isSolvable = false;

    map<pii, pii> mp;

    queue<pii> q;

    q.push(make\_pair(0, 0));

    while (!q.empty())

    {

        auto u = q.front();

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

        q.pop();

        if (m[u] == 1)

            continue;

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

            continue;

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

        m[{u.first, u.second}] = 1;

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

        {

            isSolvable = true;

            printpath(mp, u);

            if (u.first == target)

            {

                if (u.second != 0)

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

            }

            else

            {

                if (u.first != 0)

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

            }

            return;

        }

        // completely fill the jug 2

        if (m[{u.first, b}] != 1)

        {

            q.push({u.first, b});

            mp[{u.first, b}] = u;

        }

        // completely fill the jug 1

        if (m[{a, u.second}] != 1)

        {

            q.push({a, u.second});

            mp[{a, u.second}] = u;

        }

        // transfer jug 1 -> jug 2

        int d = b - u.second;

        if (u.first >= d)

        {

            int c = u.first - d;

            if (m[{c, b}] != 1)

            {

                q.push({c, b});

                mp[{c, b}] = u;

            }

        }

        else

        {

            int c = u.first + u.second;

            if (m[{0, c}] != 1)

            {

                q.push({0, c});

                mp[{0, c}] = u;

            }

        }

        // transfer jug 2 -> jug 1

        d = a - u.first;

        if (u.second >= d)

        {

            int c = u.second - d;

            if (m[{a, c}] != 1)

            {

                q.push({a, c});

                mp[{a, c}] = u;

            }

        }

        else

        {

            int c = u.first + u.second;

            if (m[{c, 0}] != 1)

            {

                q.push({c, 0});

                mp[{c, 0}] = u;

            }

        }

        // empty the jug 2

        if (m[{u.first, 0}] != 1)

        {

            q.push({u.first, 0});

            mp[{u.first, 0}] = u;

        }

        // empty the jug 1

        if (m[{0, u.second}] != 1)

        {

            q.push({0, u.second});

            mp[{0, u.second}] = u;

        }

    }

    if (!isSolvable)

        cout << "No solution";

}

int main()

{

    int Jug1,Jug2,target;

    cout<<"\nEnter jug 1 capacity : ";

    cin>>Jug1;

    cout<<"\nEnter jug 2 capacity : ";

    cin>>Jug2;

    cout<<"\nEnter target : ";

    cin>>target;

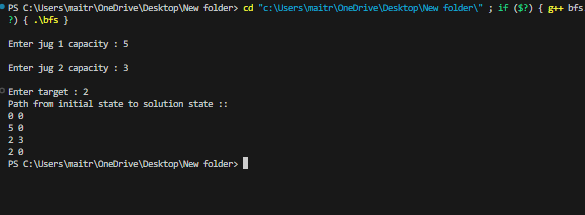
    cout << "Path from initial state "

            "to solution state ::\n";

    BFS(Jug1, Jug2, target);

    return 0;

}



1. **DFS:**  Starting from the initial state, DFS recursively explores each branch of the search tree until it reaches a leaf node or the goal state. It uses a stack data structure to keep track of the states to be explored, prioritizing deep exploration over breadth.

#include <bits/stdc++.h>

using namespace std;

typedef pair<int, int> pii;

void printpath(map<pii, pii> mp, pii u)

{

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

    {

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

        return;

    }

    printpath(mp, mp[u]);

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

}

void DFS(int a, int b, int target)

{

    map<pii, int> m;

    bool isSolvable = false;

    map<pii, pii> mp;

    stack<pii> stk;

    stk.push(make\_pair(0, 0));

    while (!stk.empty())

    {

        auto u = stk.top();

        stk.pop();

        if (m[u] == 1)

            continue;

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

            continue;

        m[{u.first, u.second}] = 1;

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

        {

            isSolvable = true;

            printpath(mp, u);

            if (u.first == target)

            {

                if (u.second != 0)

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

            }

            else

            {

                if (u.first != 0)

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

            }

            return;

        }

        // completely fill the jug 2

        if (m[{u.first, b}] != 1)

        {

            stk.push({u.first, b});

            mp[{u.first, b}] = u;

        }

        // completely fill the jug 1

        if (m[{a, u.second}] != 1)

        {

            stk.push({a, u.second});

            mp[{a, u.second}] = u;

        }

        // transfer jug 1 -> jug 2

        int d = b - u.second;

        if (u.first >= d)

        {

            int c = u.first - d;

            if (m[{c, b}] != 1)

            {

                stk.push({c, b});

                mp[{c, b}] = u;

            }

        }

        else

        {

            int c = u.first + u.second;

            if (m[{0, c}] != 1)

            {

                stk.push({0, c});

                mp[{0, c}] = u;

            }

        }

        // transfer jug 2 -> jug 1

        d = a - u.first;

        if (u.second >= d)

        {

            int c = u.second - d;

            if (m[{a, c}] != 1)

            {

                stk.push({a, c});

                mp[{a, c}] = u;

            }

        }

        else

        {

            int c = u.first + u.second;

            if (m[{c, 0}] != 1)

            {

                stk.push({c, 0});

                mp[{c, 0}] = u;

            }

        }

        // empty the jug 2

        if (m[{u.first, 0}] != 1)

        {

            stk.push({u.first, 0});

            mp[{u.first, 0}] = u;

        }

        // empty the jug 1

        if (m[{0, u.second}] != 1)

        {

            stk.push({0, u.second});

            mp[{0, u.second}] = u;

        }

    }

    if (!isSolvable)

        cout << "No solution";

}

int main()

{

    int Jug1,Jug2,target;

    cout<<"\nEnter jug 1 capacity : ";

    cin>>Jug1;

    cout<<"\nEnter jug 2 capacity : ";

    cin>>Jug2;

    cout<<"\nEnter target : ";

    cin>>target;

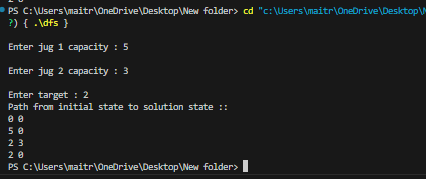
    cout << "Path from initial state "

            "to solution state ::\n";

    DFS(Jug1, Jug2, target);

    return 0;

}



1. **Missionaries & Cannibals Problem:**

**Theory-**

Missionaries and Cannibals is a standard problem in which 3 missionaries and 3 cannibals want to cross from the left bank of a river to the right bank of the river. There is a boat on the left bank, but it only carries at most two people at a time (and can never cross with zero people). If cannibals ever outnumber missionaries on either bank, the cannibals will eat the missionaries. A state can be represented by a triple, (m c b), where m is the number of missionaries on the left, c is the number of cannibals on the left, and b indicates whether the boat is on the left bank or right bank. The initial state is (3 3 L) and the goal state is (0 0 R).

A**. BFS:**

function BFS(initial\_state, goal\_state):

  frontier = Queue()

  explored = Set()

  frontier.enqueue(initial\_state)

  while not frontier.isEmpty():

    state = frontier.dequeue()

    explored.add(state)

    if state == goal\_state:

      return reconstruct\_path(state)  # Function to trace back solution path

    for successor in generate\_valid\_successors(state):

      if successor not in explored:

        frontier.enqueue(successor)

  return "No solution found"

function generate\_valid\_successors(state):

  # Implement logic to generate all possible valid successor states based on the current state information

  # (e.g., number of missionaries and cannibals on each side, boat position)

  # Ensure generated states follow the safety rule (missionaries not outnumbered by cannibals)

  # Return a list of valid successor states

function reconstruct\_path(state):

  # Implement logic to backtrack and reconstruct the solution path from the final state

  # Return the solution path as a sequence of states

1. **DFS:**

function DFS(initial\_state, goal\_state):

  frontier = Stack()

  explored = Set()

  frontier.push(initial\_state)

  while not frontier.isEmpty():

    state = frontier.pop()

    explored.add(state)

    if state == goal\_state:

      return reconstruct\_path(state)  # Function to trace back solution path

    for successor in generate\_valid\_successors(state):

      if successor not in explored:

        frontier.push(successor)

  return "No solution found"

function generate\_valid\_successors(state):

  # Implement logic to generate all possible valid successor states based on the current state information

  # (e.g., number of missionaries and cannibals on each side, boat position)

  # Ensure generated states follow the safety rule (missionaries not outnumbered by cannibals)

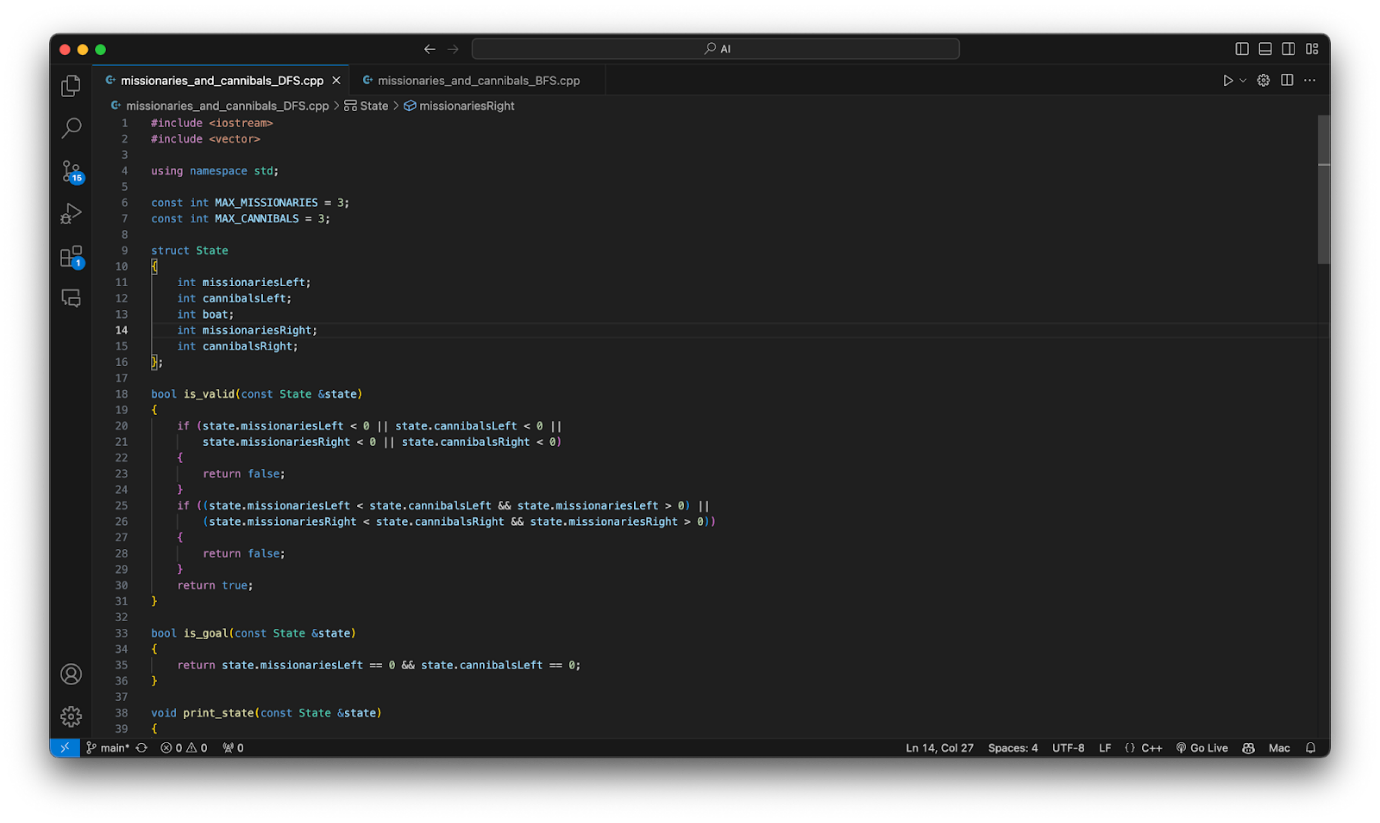
  # Return a list of valid successor states

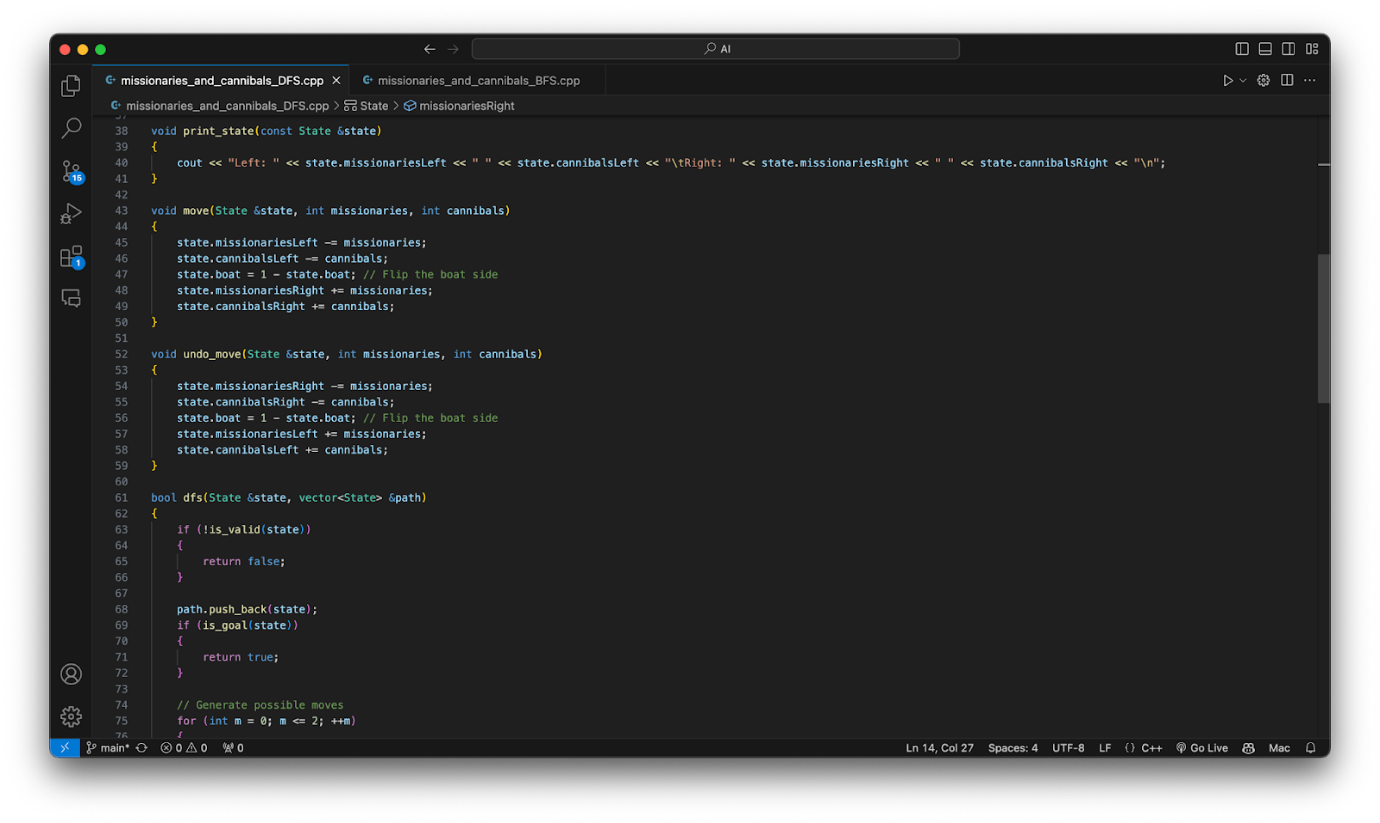
function reconstruct\_path(state):

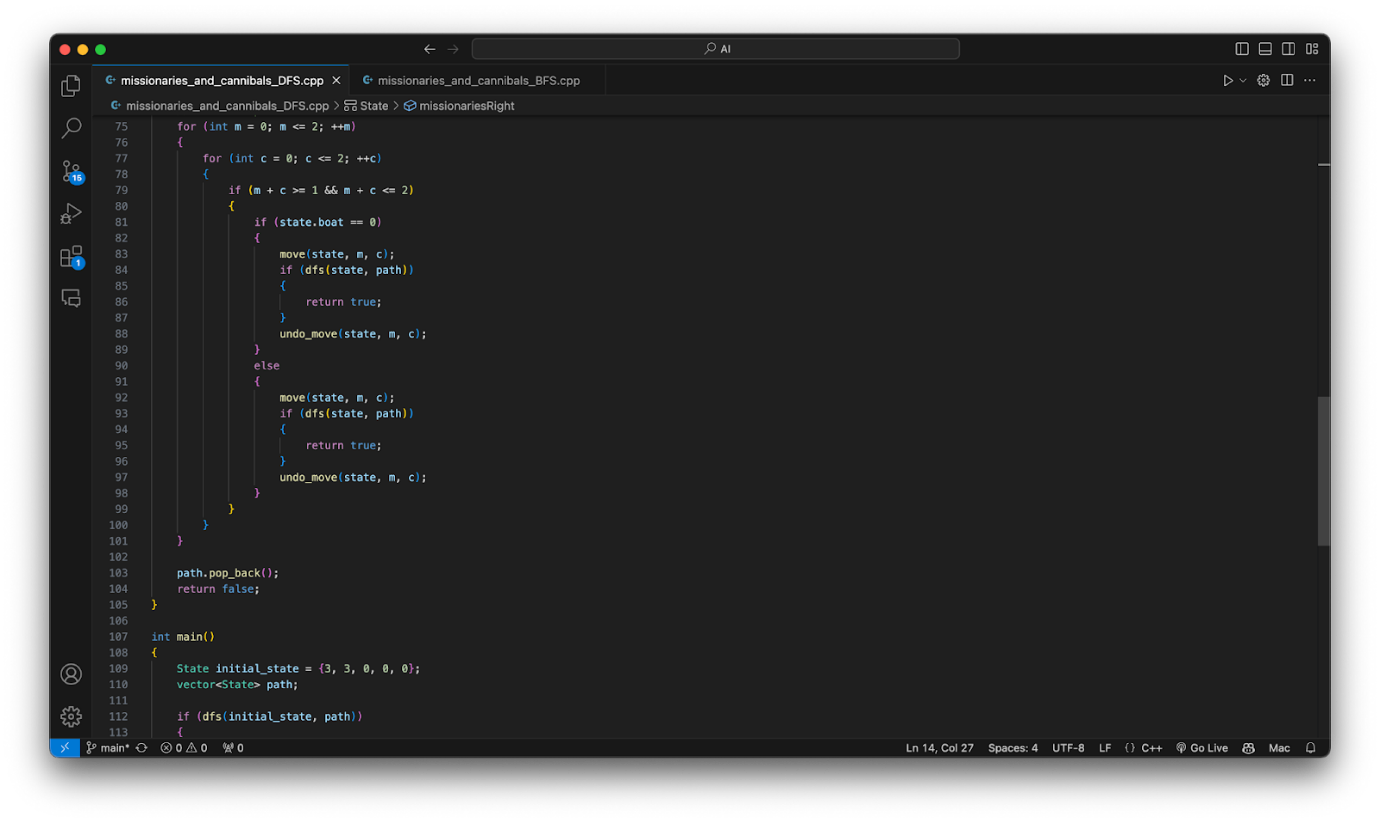
  # Implement logic to backtrack and reconstruct the solution path from the final state

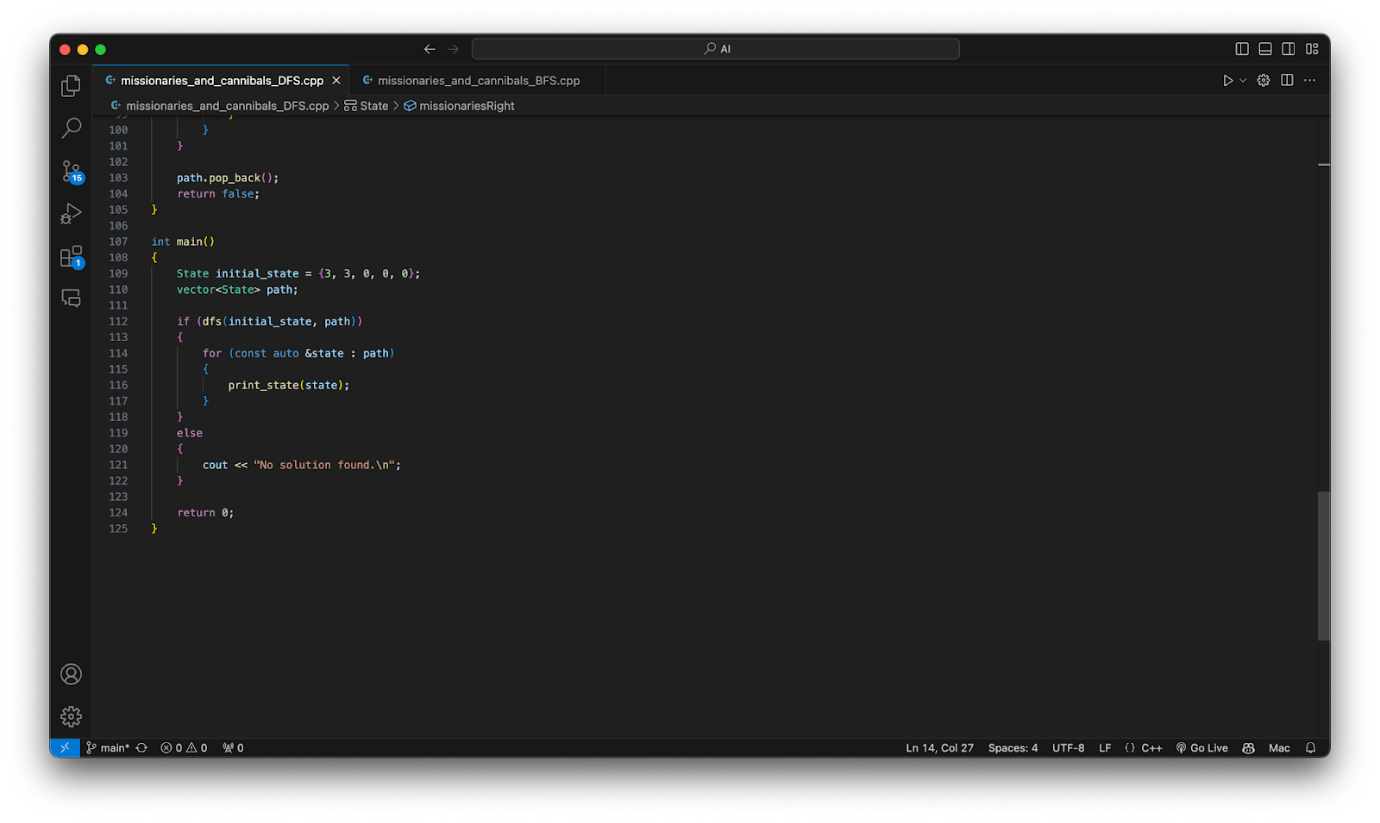
  # Return the solution path as a sequence of states

**Code:- DFS**



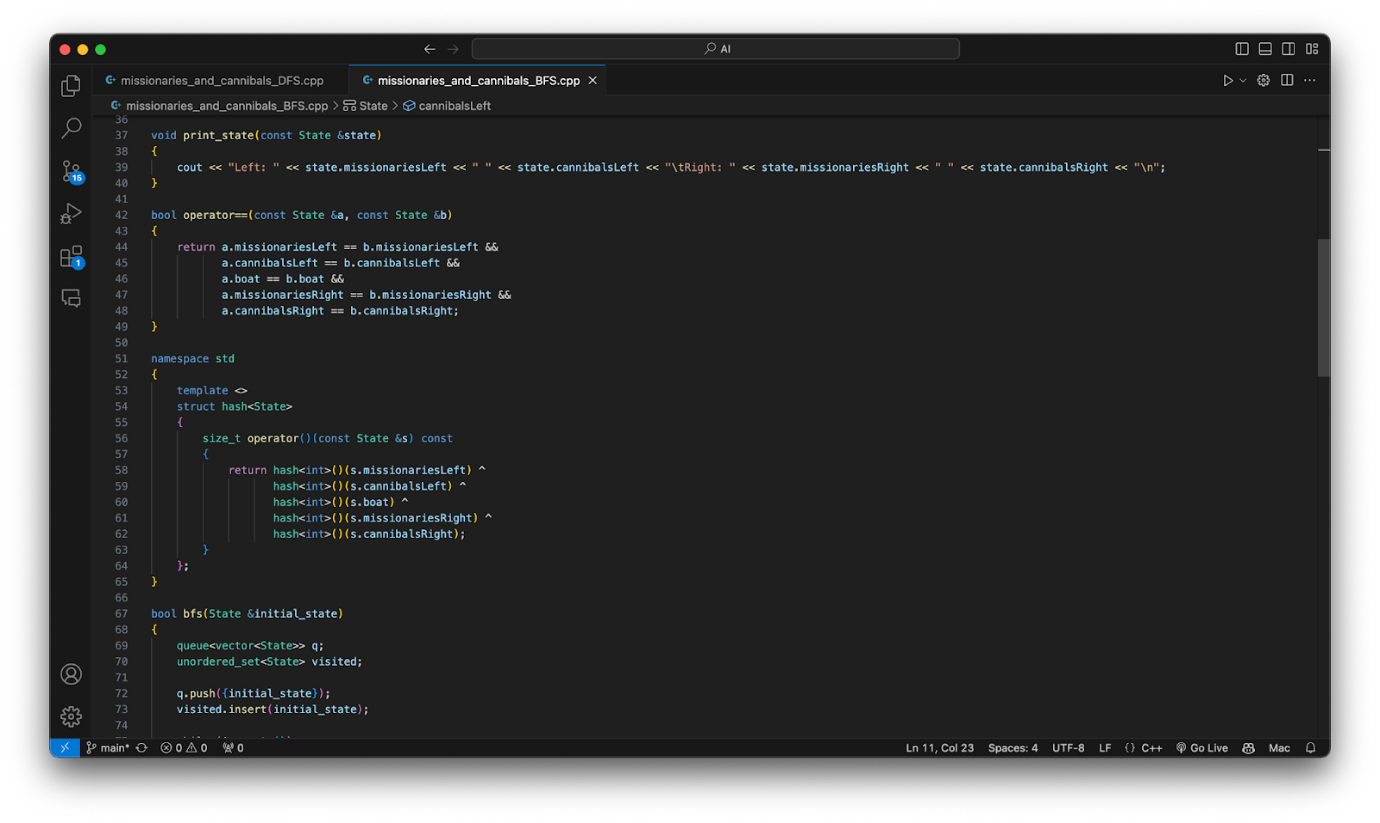




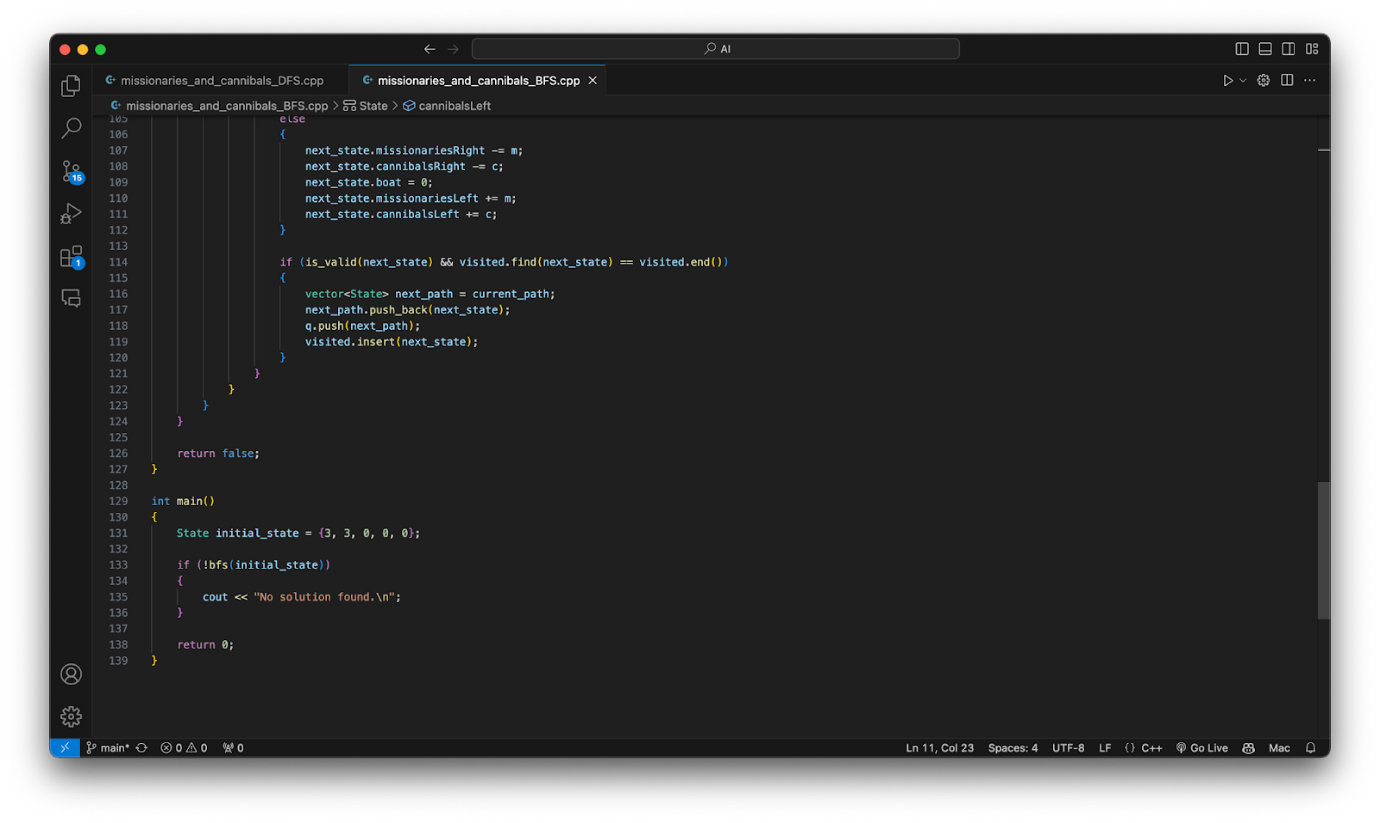


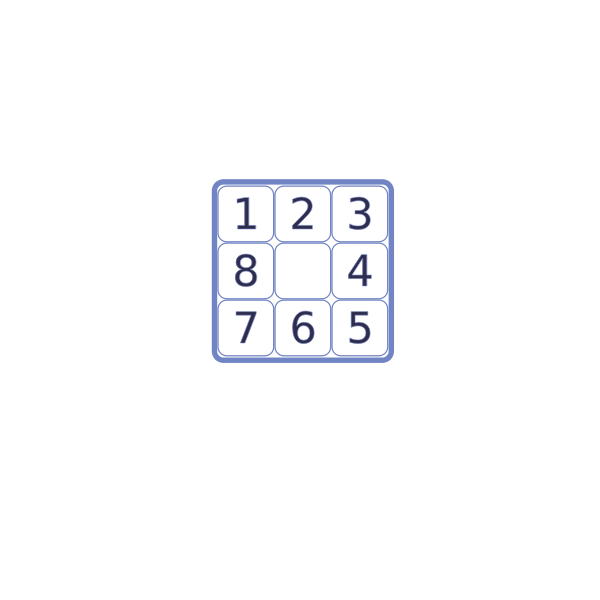
**BFS**-







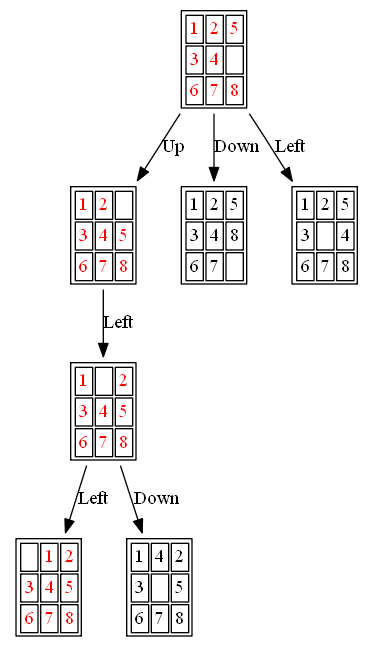


1. **8-Puzzle Problem**:

**Theory: -**

* An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares).
* One of the squares is empty.
* The object is to move squares around into different positions and display the numbers in the "goal state".
* The image to the left can be considered an unsolved initial state of the "3 x 3" 8 puzzle.

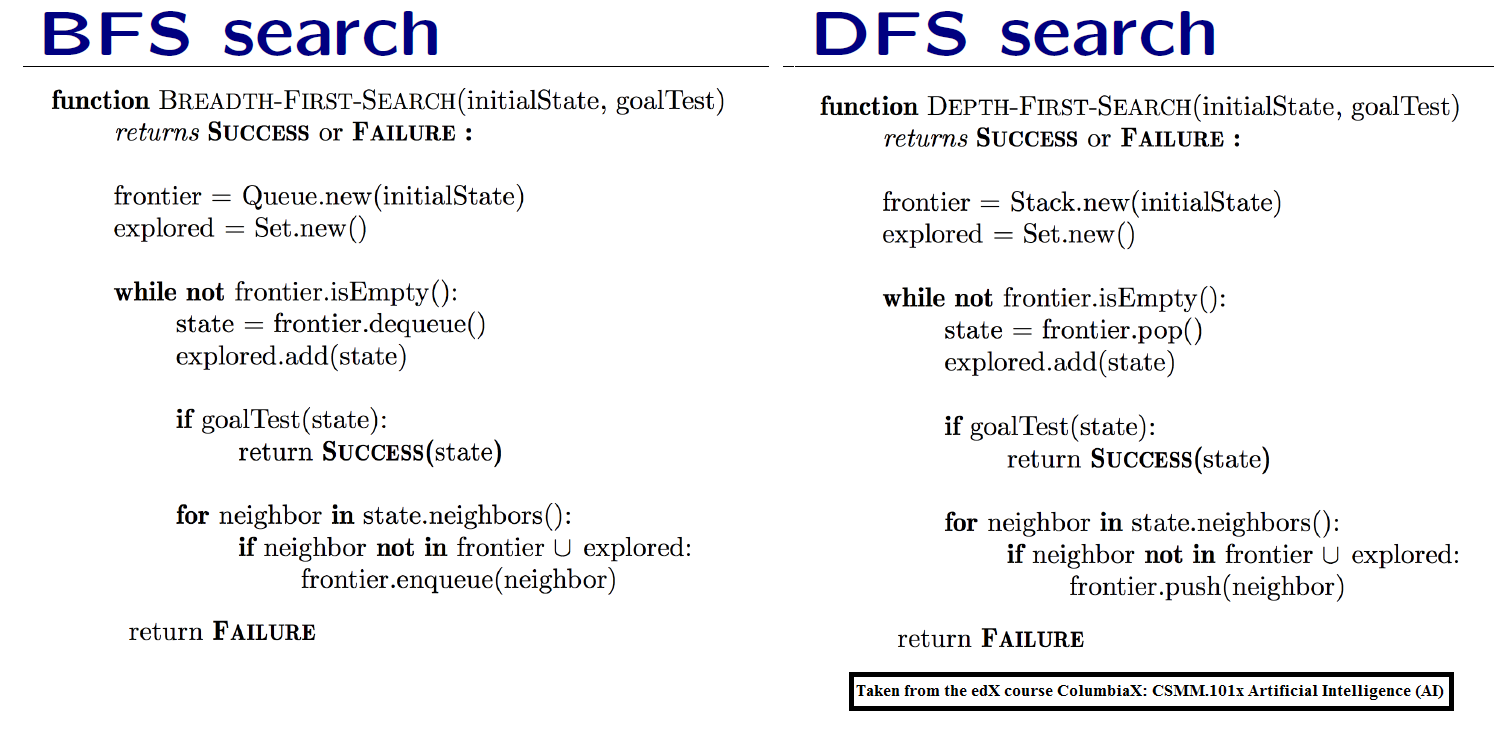
**Approach:-**



* 1. **BFS:**

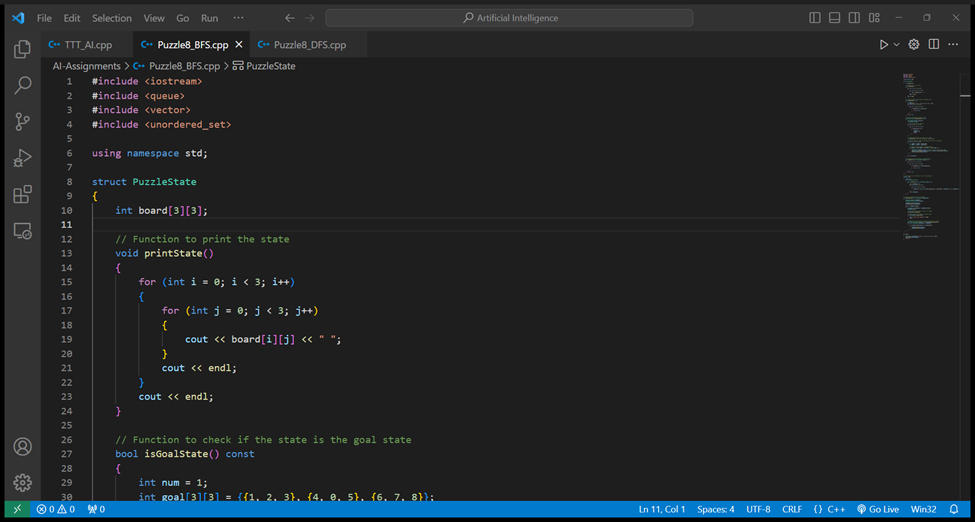
In **Breadth-First Search Uninformed Algorithm** searches begin by visiting the root node of the search tree, given by the initial state. Among other book-keeping details, three major things happen in sequence to visit a node:

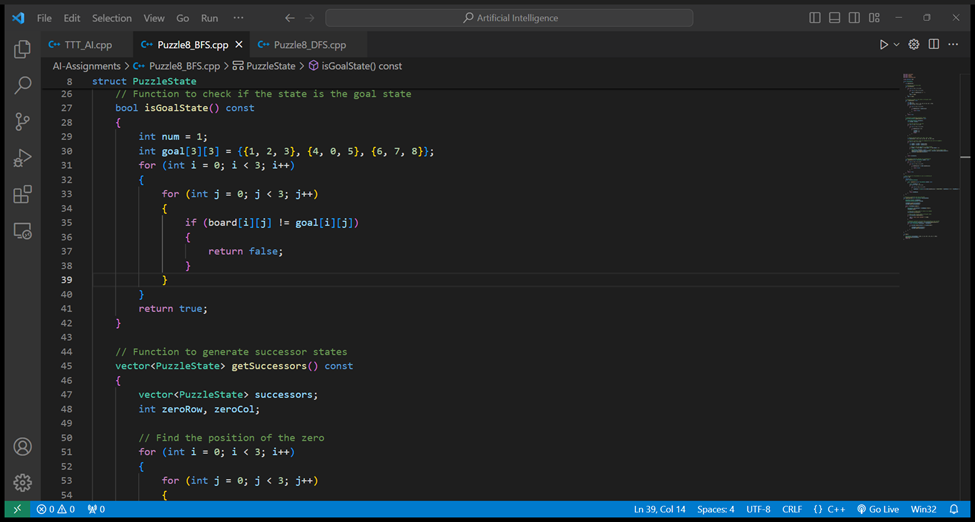
1. First, we remove a node from the **Queue**.
2. Second, we check the state against the goal state to determine if a solution has been found.
3. Finally, if the result of the check is negative, we then expand the node. To expand a given node, we generate successor nodes adjacent to the current node and add them to the Queue. Note that if these successor nodes are already in the Queue, or have already been visited, then they should not be added to the Queue again.

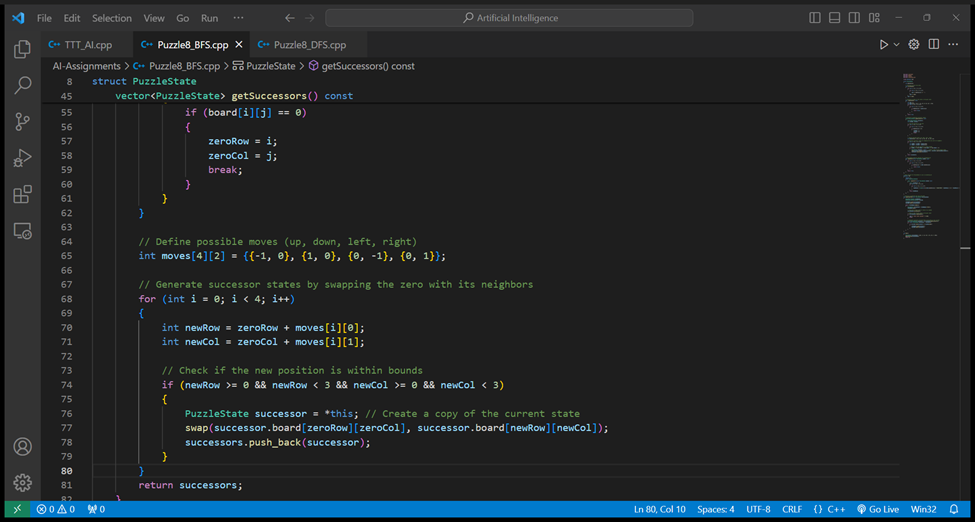


* Completeness:
  + BFS is complete when the branching factor is finite, and the depth of the search space is finite. It will find a solution if one exists.
* Optimality:
  + BFS is optimal for finding the shortest path in an unweighted graph. It explores all nodes at the current depth level before moving on to the next depth level, ensuring it finds the shallowest path first.
* Time Complexity:
  + In the worst case, the time complexity of BFS is *O*(*bd*), where *b* is the branching factor and *d* is the depth of the solution. In the case of a finite branching factor and a reasonably shallow solution, BFS is efficient.
* Space Complexity:
  + The space complexity of BFS is *O*(*bd*) due to the need to store all nodes at the current depth level in the queue. This can lead to significant memory requirements, especially for deep solutions.

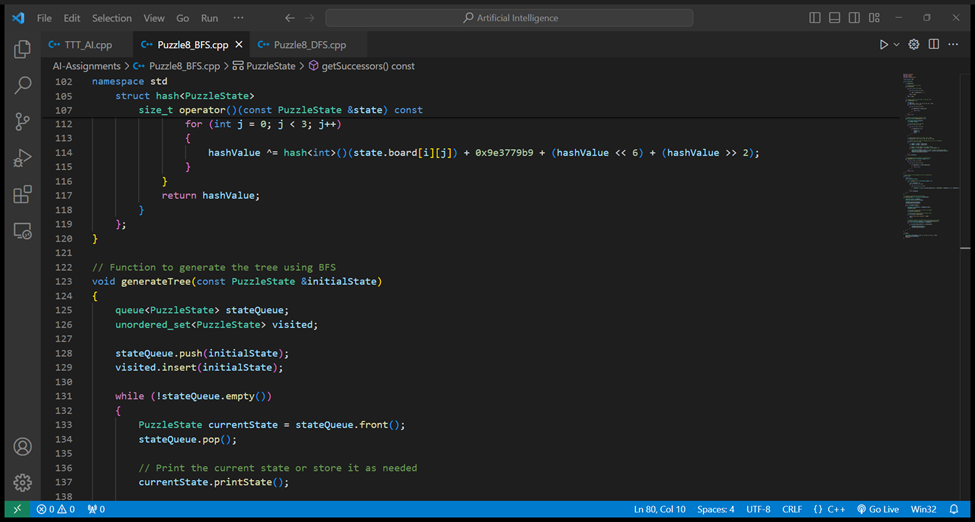
Code:-

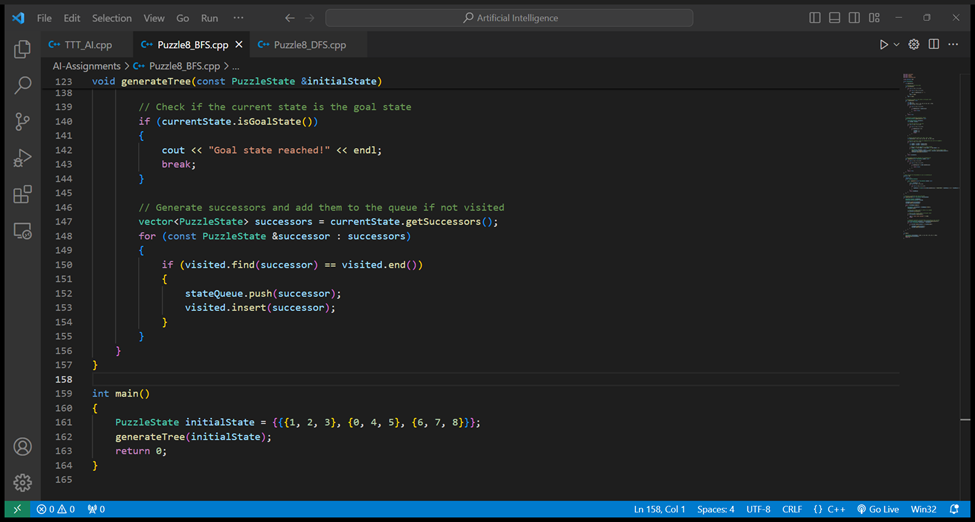


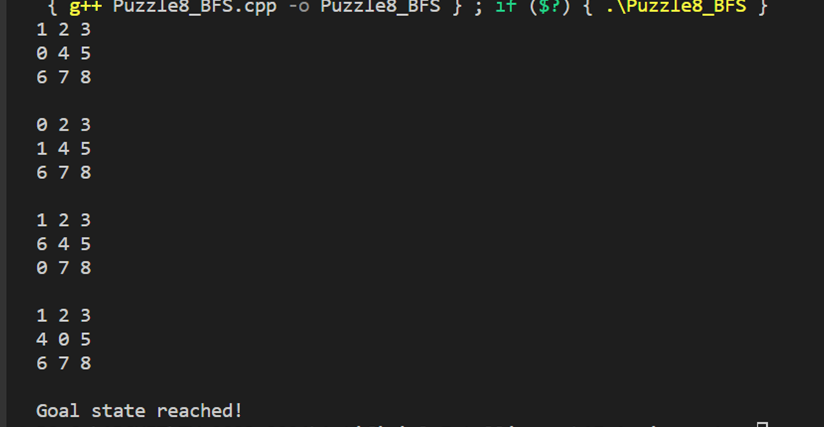








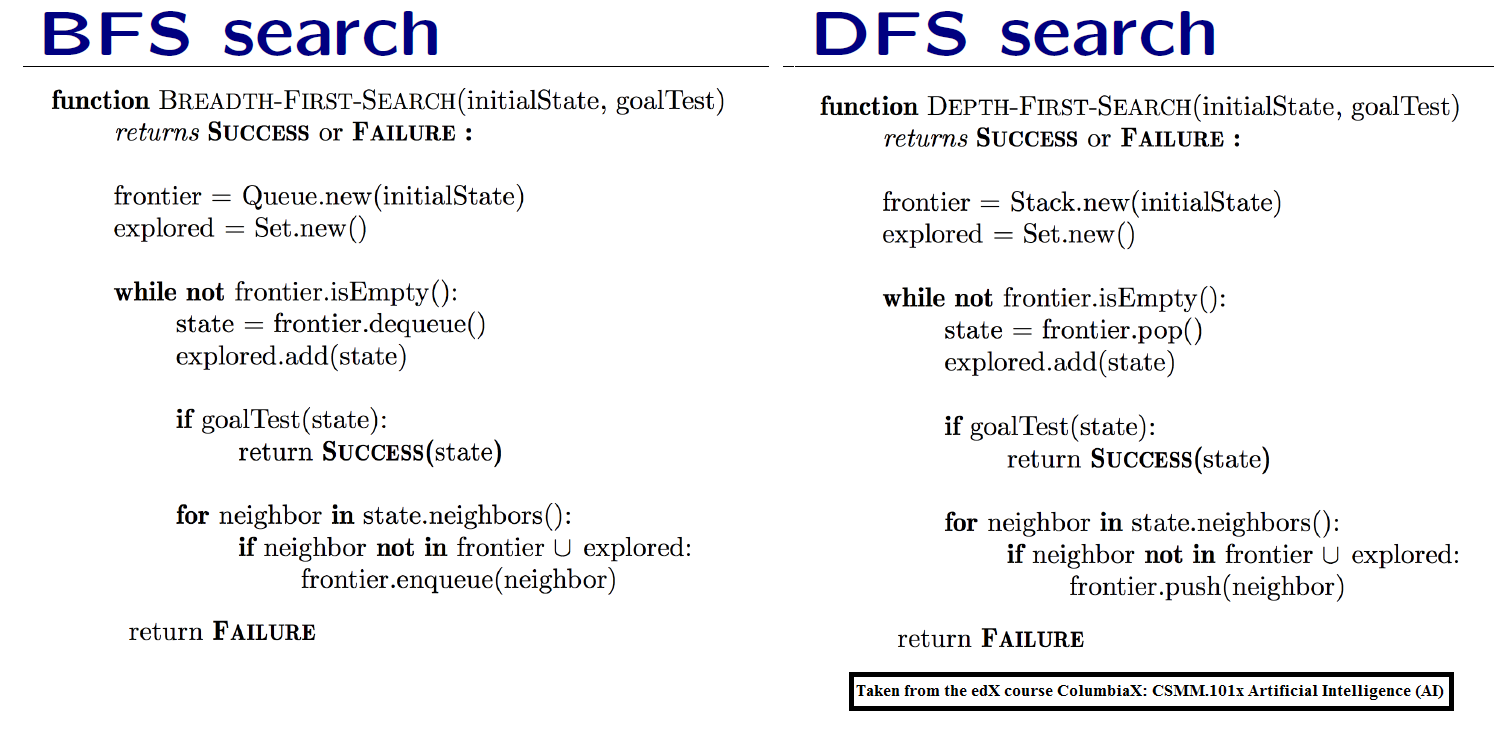




* 1. DFS:

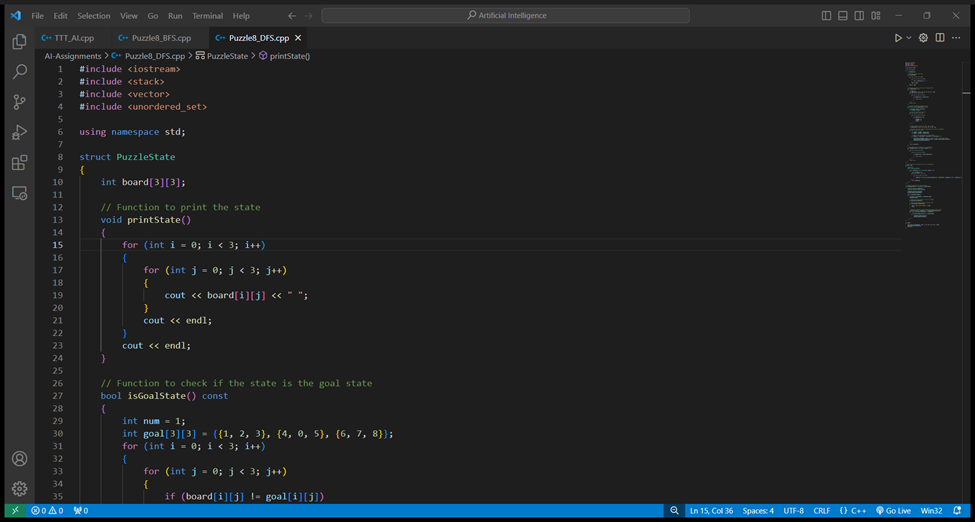
**In Depth First Search Uninformed Algorithm** searches begin by visiting the root node of the search tree, given by the initial state. Among other book-keeping details, three major things happen in sequence to visit a node:

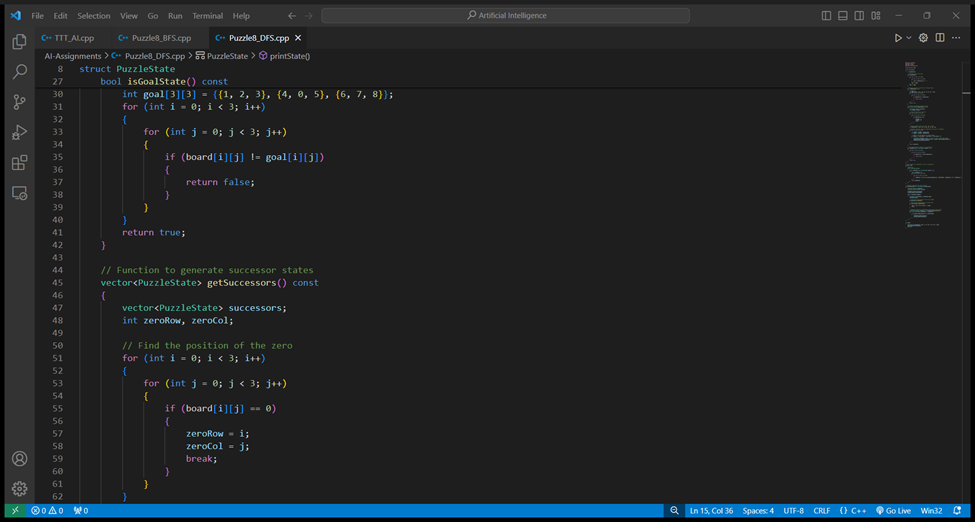
1. First, we remove a node from the Stack.
2. Second, we check the state against the goal state to determine if a solution has been found.
3. Finally, if the result of the check is negative, we then expand the node. To expand a given node, we generate successor nodes adjacent to the current node and add them to the Stack. Note that if these successor nodes are already in the Stack, or have already been visited, then they should not be added to the Queue again.

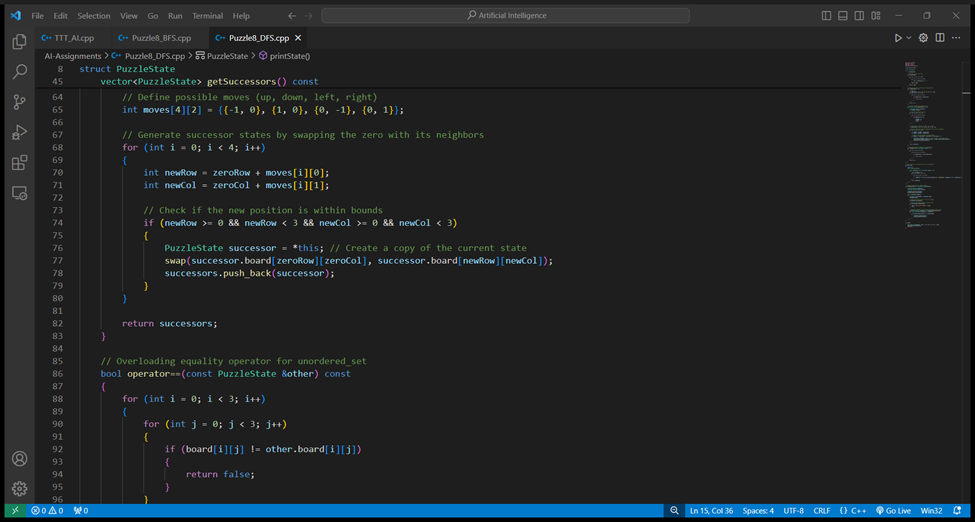


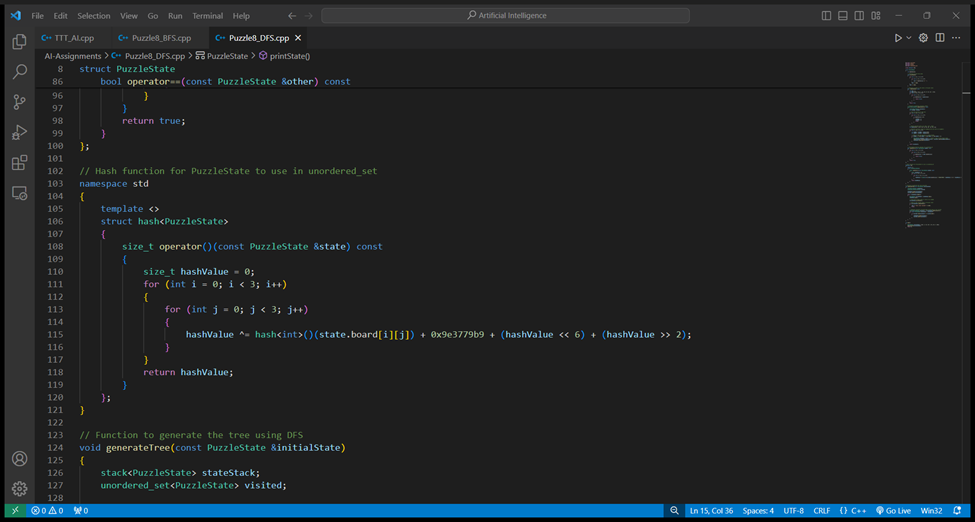
1. Completeness:
   * DFS is complete if the search space is finite. However, it may not find a solution if the graph contains cycles, as it can get stuck in an infinite loop.
2. Optimality:
   * DFS is not guaranteed to find the optimal solution. It may find a deeper solution before finding a shallower one, depending on the order in which nodes are explored.
3. Time Complexity:
   * In the worst case, the time complexity of DFS is *O*(*bm*), where *b* is the branching factor and *m* is the maximum depth of the search space. DFS can be less efficient than BFS if the solution is deep and the branching factor is high.
4. Space Complexity:
   * The space complexity of DFS is *O*(*bm*), where *b* is the branching factor and *m* is the maximum depth of the search space. Unlike BFS, DFS only needs to store a single path at any point in time, leading to lower memory requirements.

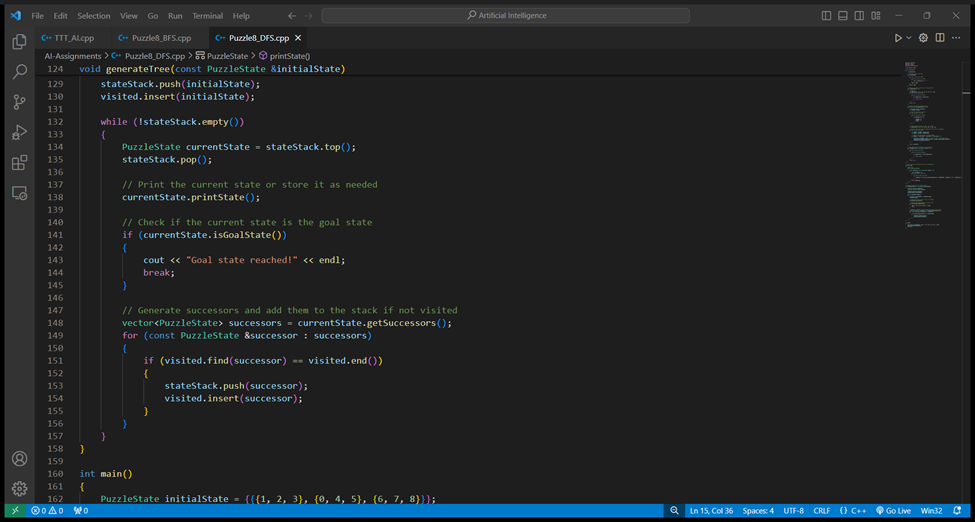
Code:-

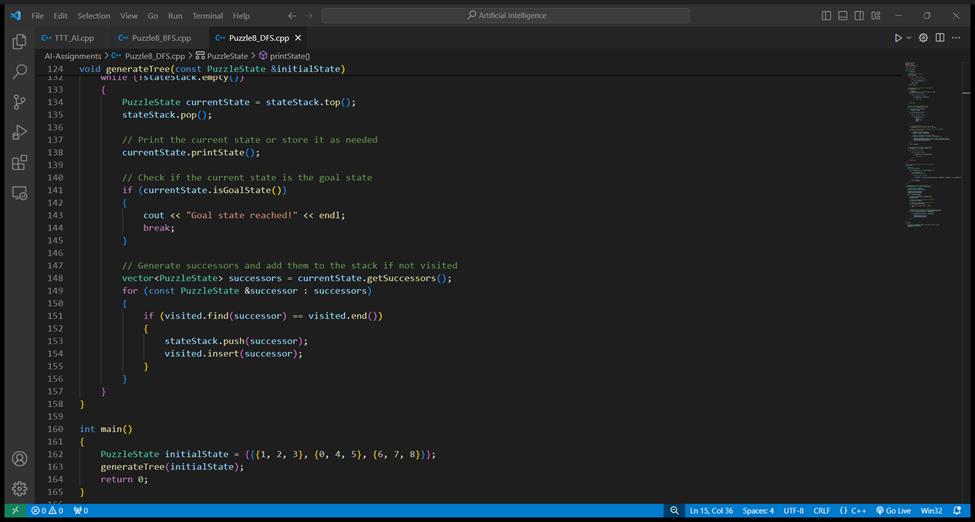


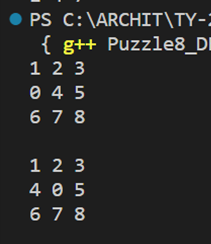












1. **N-Queens Problem:**

**Overview:**

The n-queens is the problem of placing n queens on n × n chessboard such that no two queens can attack each other. Given an integer n, return all distinct solutions to the n -queens puzzle. Each solution contains a distinct board configuration of the queen’s placement, where ‘Q’ and ‘.’ indicates queen and space respectively.

This can be possible by setting the following rules-

1. Each row should have one queen
2. Each column should have one queen
3. No two queens can attack each other

**Approaches to solve this problem:**

1. **DFS**

* DFS strategy to solve NQueen’s problem-
* Start in the leftmost column
* If all queens are placed return true
* Try all rows in the current column. Do the following for every row.
* If the queen can be placed safely in this row
* Then mark this [row, column] as part of the solution and recursively check if placing queen here leads to a solution.
* If placing the queen in [row, column] leads to a solution then return true.
* If placing queen doesn’t lead to a solution then unmark this [row, column] then backtrack and try other rows.
* If all rows have been tried and the valid solution is not found return false to trigger backtracking.

Time complexity  O(N! \* N)

Space complexity O(N^2)

#include <bits/stdc++.h>

using namespace std;

class Solution {

    private:

    bool isSafe(int row, int col, vector<string> board, int n) {

      // store copies of pos(row,col)

      int duprow = row;

      int dupcol = col;

      while (row >= 0 && col >= 0) { //check north west diagonal O(n)

        if (board[row][col] == 'Q')

          return false;

        row--;

        col--;

      }

      col = dupcol;

      row = duprow;

      while (col >= 0) { //check west O(n)

        if (board[row][col] == 'Q')

          return false;

        col--;

      }

      row = duprow;

      col = dupcol;

      while (row < n && col >= 0) { //check south west diagonal O(n)

        if (board[row][col] == 'Q')

          return false;

        row++;

        col--;

      }

      return true;

    }

    void solve(int col, vector < string > & board, vector < vector < string >> & ans, int n) {

      if (col == n) { //base case- last column=n-1

        ans.push\_back(board);

        return;

      }

      for (int row = 0; row < n; row++) { //iterate through every row (fill each row) O(n)

        if (isSafe(row, col, board, n)) { //check if position is safe

          board[row][col] = 'Q';

          solve(col + 1, board, ans, n); //recursively check for each column (fill each column)

          board[row][col] = '.'; //while coming back, remove queen

        }

      }

    }

public:

vector<vector<string>> solveNQueens(int n) {

      vector<vector<string>> ans;

      vector<string> board(n);

      string s(n, '.');

      for (int i = 0; i < n; i++) {

        board[i] = s;

      }

      solve(0, board, ans, n);

      return ans;

    }

};

int main() {

  int n = 4;

  Solution obj;

  vector <vector<string>> ans = obj.solveNQueens(n);

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

    cout << "Arrangement " << i + 1 << "\n";

    for (int j = 0; j < ans[0].size(); j++) {

      cout << ans[i][j];

      cout << endl;

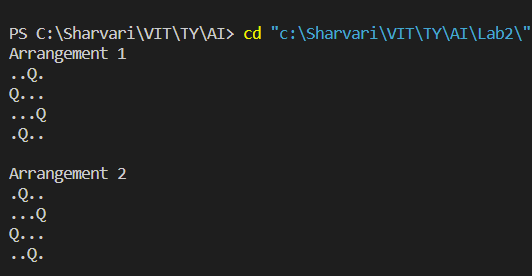
    }

    cout << endl;

  }

  return 0;

}



1. **BFS**

BFS algorithm to solve NQueen’s problem-

* The algorithm enters a loop that continues until the queue is empty.
* In each iteration:
* A state is dequeued from the front of the queue.
* If the dequeued state represents a valid solution (no conflicts between queens), it is added to the solutions vector.
* If the state has conflicts (two queens attacking each other), it is discarded, and the algorithm proceeds to the next iteration of the loop.
* If the state is not a solution, child states are generated by placing a queen in the next row in all possible columns without conflicts.
* Each of these child states is then enqueued into the queue to explore further.

* Time complexity  O(N^N)
* Space complexity O(N^2)

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

class NQueens {

private:

    int size;

public:

    NQueens(int size) : size(size) {}

    bool conflict(const vector<pair<int, int>>& queens) {

        for (int i = 1; i < queens.size(); ++i) {

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

                int a = queens[i].first;

                int b = queens[i].second;

                int c = queens[j].first;

                int d = queens[j].second;

                if (a == c || b == d || abs(a - c) == abs(b - d)) {

                    return true;

                }

            }

        }

        return false;

    }

    vector<vector<pair<int, int>>> solveBFS() {

        vector<vector<pair<int, int>>> solutions;

        if (size < 1) {

            return solutions;

        }

        queue<vector<pair<int, int>>> q;

        q.push({});

        while (!q.empty()) {

            auto solution = q.front();

            q.pop();

            if (conflict(solution)) {

                continue;

            }

            int row = solution.size();

            if (row == size) {

                solutions.push\_back(solution);

                continue;

            }

            for (int col = 0; col < size; ++col) {

                vector<pair<int, int>> queens = solution;

                queens.push\_back({row, col});

                q.push(queens);

            }

        }

        return solutions;

    }

    void print(const vector<pair<int, int>>& queens) {

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

            cout << " ---";

        }

        cout << endl;

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

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

                char p = ' ';

                for (auto& queen : queens) {

                    if (queen.first == i && queen.second == j) {

                        p = 'Q';

                        break;

                    }

                }

                cout << "| " << p << " ";

            }

            cout << "|" << endl;

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

                cout << " ---";

            }

            cout << endl;

        }

    }

};

int main() {

    int n = 4;

    NQueens nQueens(n);

    vector<vector<pair<int, int>>> solutions = nQueens.solveBFS();

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

        cout << "Arrangement " << i + 1 << ":\n";

        nQueens.print(solutions[i]);

        cout << endl;

    }

    return 0;

}

