U23AI021

Lab assignment 06
Artificial Intelligence

Harshil Andhariya
u23ai021@coed.svnit.ac.in

Q1.

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

using namespace std;

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

struct Node {
    vector<vector<int>> state;
    int x, y, cost, depth;
    Node* parent;

    Node(vector<vector<int>> s, int x, int y, int depth,
Node* p) : state(s), x(x), y(y), depth(depth), parent(p) {
        cost = depth + heuristic();
    }

    int heuristic() {
        int h = 0;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (state[i][j] != 0) {
                    int val = state[i][j] - 1;
                    int targetX = val / N;
                    int targetY = val % N;
                    h += abs(i - targetX) + abs(j -
targetY);
                }
            }
        }
        return h;
    }
};

bool isGoal(const vector<vector<int>>& state) {
    return state == goal;
}

vector<Node*> getNeighbors(Node* node) {
    vector<Node*> neighbors;
    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};

    for (int i = 0; i < 4; i++) {
        int nx = node->x + dx[i];
        int ny = node->y + dy[i];
        if (nx >= 0 && nx < N && ny >= 0 && ny < N) {
            vector<vector<int>> newState = node->state;
            swap(newState[node->x][node->y],
newState[nx][ny]);
            neighbors.push_back(new Node(newState, nx, ny,
node->depth + 1, node));
        }
    }
    return neighbors;
}

void printSolution(Node* node) {
    if (node == nullptr) return;
    printSolution(node->parent);
    for (const auto& row : node->state) {
        for (int num : row) {
            cout << num << " ";
        }
        cout << endl;
    }
    cout << "-----" << endl;
}

void bfs(vector<vector<int>> start, int x, int y) {
    queue<Node*> q;
    set<vector<vector<int>>> visited;
    q.push(new Node(start, x, y, 0, nullptr));
    while (!q.empty()) {
        Node* node = q.front(); q.pop();
        if (isGoal(node->state)) {
            printSolution(node);
            return;
        }
        visited.insert(node->state);
        for (Node* neighbor : getNeighbors(node)) {
            if (visited.find(neighbor->state) ==
visited.end()) {
                q.push(neighbor);
            }
        }
    }
}

void dfs(vector<vector<int>> start, int x, int y) {
    stack<Node*> s;
    set<vector<vector<int>>> visited;
    s.push(new Node(start, x, y, 0, nullptr));

    while (!s.empty()) {
        Node* node = s.top(); s.pop();
        if (isGoal(node->state)) {
            printSolution(node);
            return;
        }
        visited.insert(node->state);
        for (Node* neighbor : getNeighbors(node)) {
            if (visited.find(neighbor->state) ==
visited.end()) {
                s.push(neighbor);
            }
        }
    }
}
```

```
void aStar(vector<vector<int>> start, int x, int y) {
    auto cmp = [](Node* a, Node* b) { return a->cost > b-
>cost; };
    priority_queue<Node*, vector<Node*>, decltype(cmp)>
pq(cmp);
    set<vector<vector<int>>> visited;
    pq.push(new Node(start, x, y, 0, nullptr));

    while (!pq.empty()) {
        Node* node = pq.top(); pq.pop();
        if (isGoal(node->state)) {
            printSolution(node);
            return;
        }
        visited.insert(node->state);
        for (Node* neighbor : getNeighbors(node)) {
            if (visited.find(neighbor->state) ==
visited.end()) {
                pq.push(neighbor);
            }
        }
    }
}

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

    cout << "BFS Solution:\n";
    bfs(start, x, y);

    cout << "\nDFS Solution:\n";
    dfs(start, x, y);

    cout << "\nA* Search Solution:\n";
    aStar(start, x, y);
    return 0;
}
```

BFS Solution:

```
1 2 3
4 0 6
7 5 8
-----
1 2 3
4 5 6
7 0 8
-----
1 2 3
4 5 6
7 8 0
-----
```

DFS Solution:

```
1 2 3
4 0 6
7 5 8
-----
1 2 3
4 6 0
7 5 8
-----
1 2 3
4 6 8
7 5 0
-----
1 2 3
4 6 8
7 0 5
-----
1 2 3
4 6 8
```

```
0 7 5
-----
1 2 3
0 6 8
4 7 5
-----
1 2 3
6 0 8
4 7 5
-----
1 2 3
6 8 0
4 7 5
-----
1 2 3
6 8 5
4 7 0
-----
1 2 3
6 8 5
4 0 7
-----
1 2 3
6 8 5
0 4 7
-----
1 2 3
0 8 5
6 4 7
-----
1 2 3
8 0 5
6 4 7
-----
1 2 3
8 5 0
6 4 7
-----
```

```
1 2 3
8 5 7
6 4 0
-----
1 2 3
8 5 7
6 0 4
-----
1 2 3
8 5 7
0 6 4
-----
1 2 3
0 5 7
8 6 4
-----
1 2 3
5 0 7
8 6 4
-----
1 2 3
5 7 0
8 6 4
-----
1 2 3
5 7 4
8 6 0
-----
1 2 3
5 7 4
8 0 6
-----
1 2 3
5 7 4
0 8 6
-----
1 2 3
0 7 4
```

```
5 8 6
-----
1 2 3
7 0 4
5 8 6
-----
1 2 3
7 4 0
5 8 6
-----
1 2 3
7 4 6
5 8 0
-----
1 2 3
7 4 6
5 0 8
-----
1 2 3
7 4 6
0 5 8
-----
1 2 3
0 4 6
7 5 8
-----
0 2 3
1 4 6
7 5 8
-----
2 0 3
1 4 6
7 5 8
-----
2 3 0
1 4 6
7 5 8
-----
```

```
2 3 6
1 4 0
7 5 8
-----
2 3 6
1 0 4
7 5 8
-----
2 3 6
0 1 4
7 5 8
-----
2 3 6
7 1 4
0 5 8
-----
2 3 6
7 1 4
5 0 8
-----
2 3 6
7 1 4
5 8 0
-----
2 3 6
7 1 0
5 8 4
-----
2 3 6
7 0 1
5 8 4
-----
2 3 6
0 7 1
5 8 4
-----
2 3 6
5 7 1
```

0 8 4

-----

2 3 6

5 7 1

8 0 4

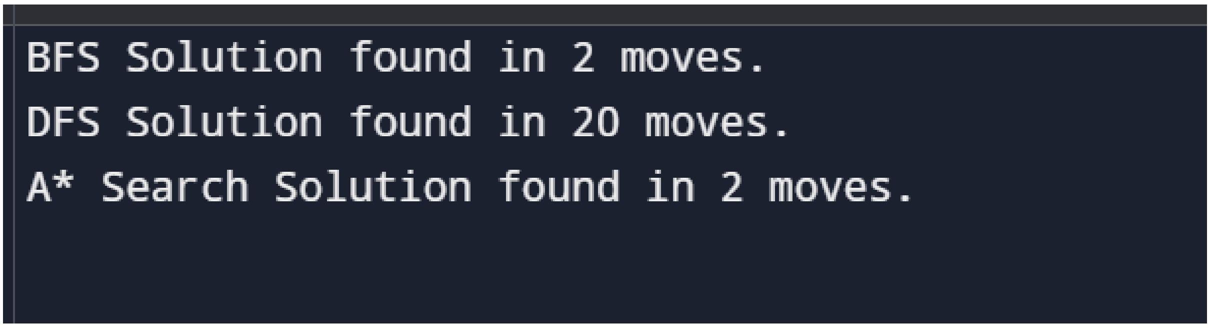
Will go for very large scale

Cause of infinite search space

Q1b)

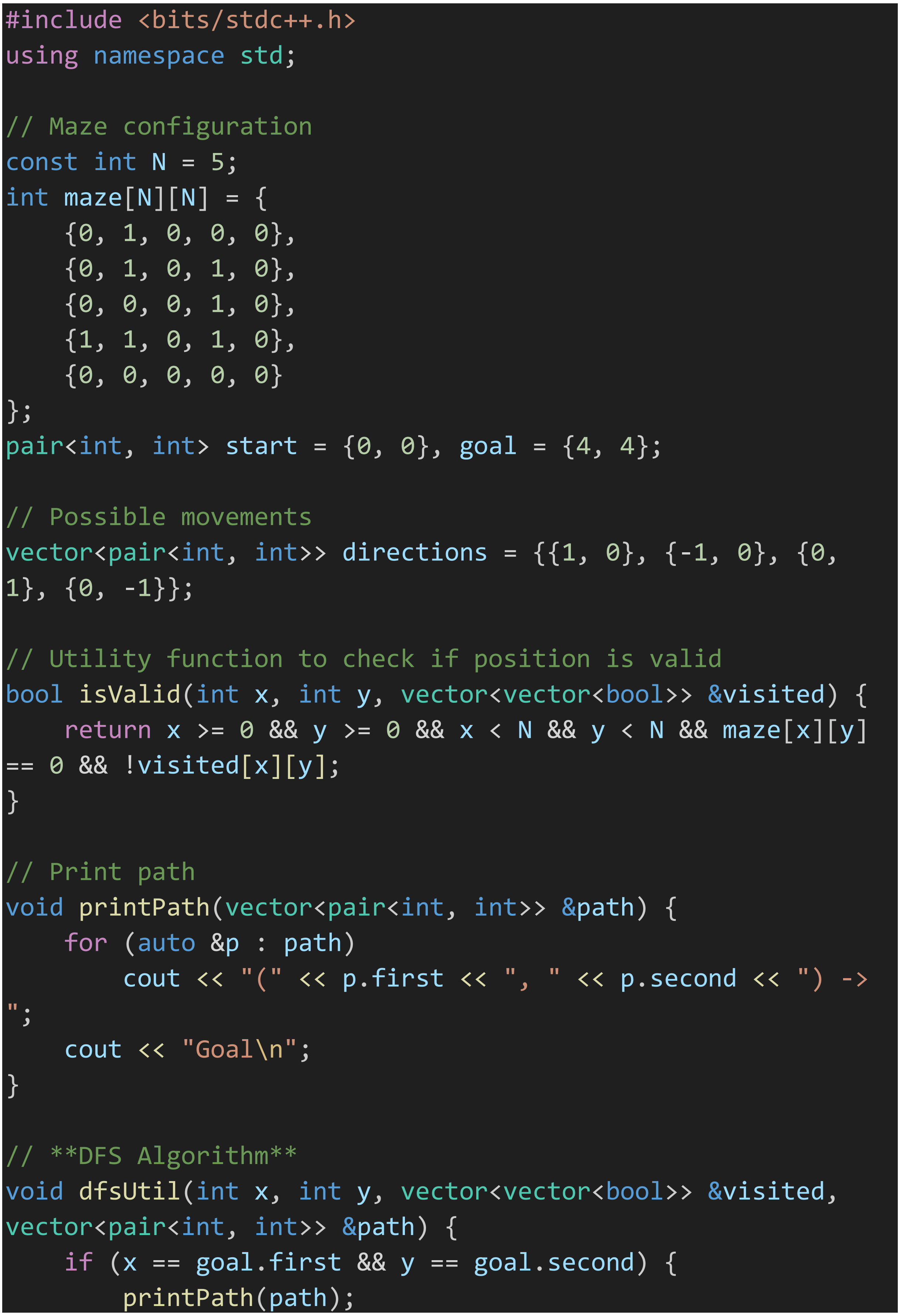
```
#include <iostream>
#include <vector>
#include <queue>
#include <set>
#include <stack>
#include <map>
#include <algorithm>

using namespace std;

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

struct Node {
    vector<vector<int>> state;
    int x, y, cost, depth;
    Node* parent;

    Node(vector<vector<int>> s, int x, int y, int depth, Node* p) : state(s), x(x), y(y), depth(depth), parent(p) {
        cost = depth + heuristic();
    }

    int heuristic() {
        int h = 0;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                if (state[i][j] != 0) {
                    int val = state[i][j] - 1;
                    int targetX = val / N;
                    int targetY = val % N;
                    h += abs(i - targetX) + abs(j - 
targetY);
                }
            }
        }
        return h;
    }
};

bool isGoal(const vector<vector<int>>& state) {
    return state == goal;
}

vector<Node*> getNeighbors(Node* node) {
    vector<Node*> neighbors;
    int dx[] = {-1, 1, 0, 0};
    int dy[] = {0, 0, -1, 1};

    for (int i = 0; i < 4; i++) {
        int nx = node->x + dx[i];
        int ny = node->y + dy[i];
        if (nx >= 0 && nx < N && ny >= 0 && ny < N) {
            vector<vector<int>> newState = node->state;
            swap(newState[node->x][node->y], 
newState[nx][ny]);
            neighbors.push_back(new Node(newState, nx, ny, 
node->depth + 1, node));
        }
    }
    return neighbors;
}
```

```
int getSolutionDepth(Node* node) {
    int moves = 0;
    while (node->parent) {
        moves++;
        node = node->parent;
    }
    return moves;
}

void bfs(vector<vector<int>> start, int x, int y) {
    queue<Node*> q;
    set<vector<vector<int>>> visited;
    q.push(new Node(start, x, y, 0, nullptr));

    while (!q.empty()) {
        Node* node = q.front(); q.pop();
        if (isGoal(node->state)) {
            cout << "BFS Solution found in " <<
getSolutionDepth(node) << " moves." << endl;
            return;
        }
        visited.insert(node->state);
        for (Node* neighbor : getNeighbors(node)) {
            if (visited.find(neighbor->state) ==
visited.end()) {
                q.push(neighbor);
            }
        }
    }
}

void dfs(vector<vector<int>> start, int x, int y, int
maxDepth = 20) {
    stack<Node*> s;
    set<vector<vector<int>>> visited;
    s.push(new Node(start, x, y, 0, nullptr));
    while (!s.empty()) {
        Node* node = s.top(); s.pop();
        if (isGoal(node->state)) {
            cout << "DFS Solution found in " << getSolutionDepth(node) << " moves." << endl;
            return;
        }
        if (node->depth >= maxDepth) continue;
        visited.insert(node->state);
        for (Node* neighbor : getNeighbors(node)) {
            if (visited.find(neighbor->state) == visited.end()) {
                s.push(neighbor);
            }
        }
    }
    cout << "DFS did not find a solution within depth limit." << endl;
}

void aStar(vector<vector<int>> start, int x, int y) {
    auto cmp = [](Node* a, Node* b) { return a->cost > b->cost; };
    priority_queue<Node*, vector<Node*>, decltype(cmp)> pq(cmp);
    set<vector<vector<int>>> visited;
    pq.push(new Node(start, x, y, 0, nullptr));

    while (!pq.empty()) {
        Node* node = pq.top(); pq.pop();
        if (isGoal(node->state)) {
            cout << "A* Search Solution found in " << getSolutionDepth(node) << " moves." << endl;
            return;
        }
        visited.insert(node->state);
        for (Node* neighbor : getNeighbors(node)) {
            if (visited.find(neighbor->state) ==
visited.end()) {
                pq.push(neighbor);
            }
        }
    }
}

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

    bfs(start, x, y);
    dfs(start, x, y);
    aStar(start, x, y);

    return 0;
}
```

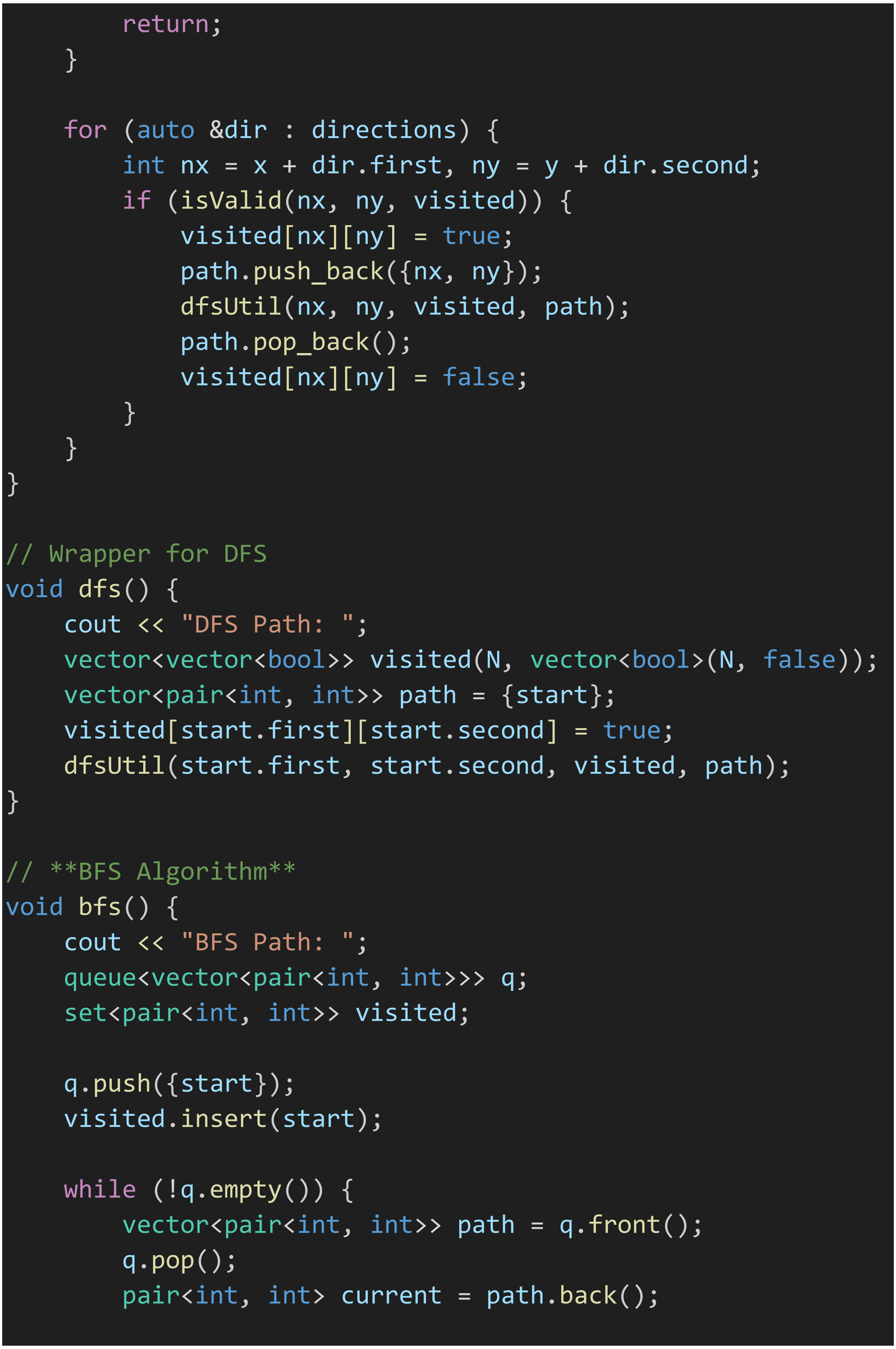


```
BFS Solution found in 2 moves.
DFS Solution found in 20 moves.
A* Search Solution found in 2 moves.
```

Q2)

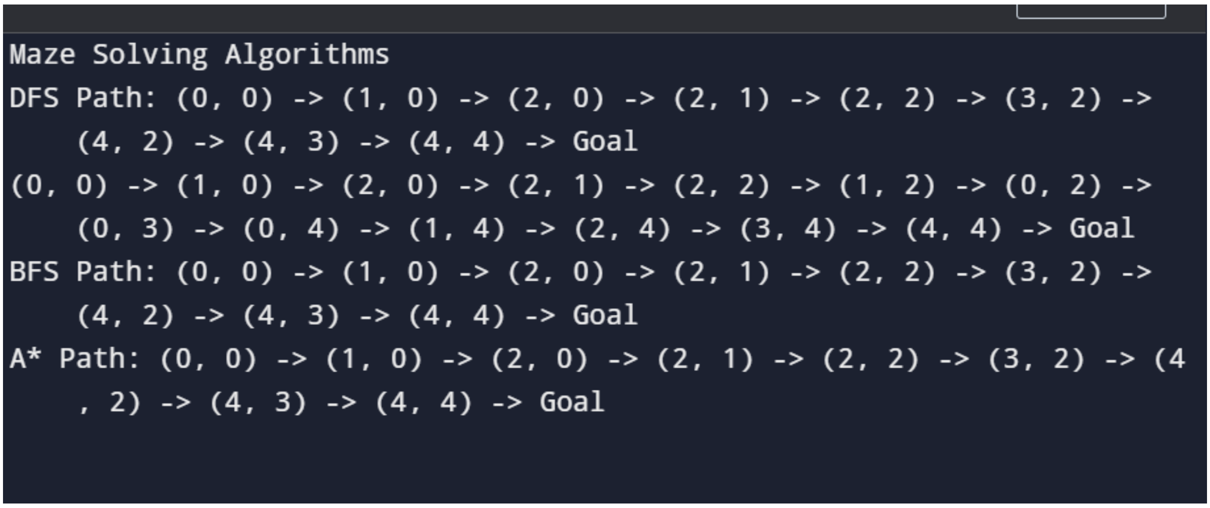


```
#include <bits/stdc++.h>
using namespace std;

// Maze configuration
const int N = 5;
int maze[N][N] = {
    {0, 1, 0, 0, 0},
    {0, 1, 0, 1, 0},
    {0, 0, 0, 1, 0},
    {1, 1, 0, 1, 0},
    {0, 0, 0, 0, 0}
};
pair<int, int> start = {0, 0}, goal = {4, 4};

// Possible movements
vector<pair<int, int>> directions = {{1, 0}, {-1, 0}, {0,
1}, {0, -1}};

// Utility function to check if position is valid
bool isValid(int x, int y, vector<vector<bool>> &visited) {
    return x >= 0 && y >= 0 && x < N && y < N && maze[x][y]
== 0 && !visited[x][y];
}

// Print path
void printPath(vector<pair<int, int>> &path) {
    for (auto &p : path)
        cout << "(" << p.first << ", " << p.second << ") ->
";
    cout << "Goal\n";
}

// **DFS Algorithm**
void dfsUtil(int x, int y, vector<vector<bool>> &visited,
vector<pair<int, int>> &path) {
    if (x == goal.first && y == goal.second) {
        printPath(path);
```



```cpp
        return;
    }

    for (auto &dir : directions) {
        int nx = x + dir.first, ny = y + dir.second;
        if (isValid(nx, ny, visited)) {
            visited[nx][ny] = true;
            path.push_back({nx, ny});
            dfsUtil(nx, ny, visited, path);
            path.pop_back();
            visited[nx][ny] = false;
        }
    }
}

// Wrapper for DFS
void dfs() {
    cout << "DFS Path: ";
    vector<vector<bool>> visited(N, vector<bool>(N, false));
    vector<pair<int, int>> path = {start};
    visited[start.first][start.second] = true;
    dfsUtil(start.first, start.second, visited, path);
}

// **BFS Algorithm**
void bfs() {
    cout << "BFS Path: ";
    queue<vector<pair<int, int>>> q;
    set<pair<int, int>> visited;

    q.push({start});
    visited.insert(start);

    while (!q.empty()) {
        vector<pair<int, int>> path = q.front();
        q.pop();
        pair<int, int> current = path.back();
        if (current == goal) {
            printPath(path);
            return;
        }

        for (auto &dir : directions) {
            int nx = current.first + dir.first, ny =
current.second + dir.second;
            if (isValid(nx, ny, *new vector<vector<bool>>(N,
vector<bool>(N, false))) && visited.find({nx, ny}) ==
visited.end()) {
                vector<pair<int, int>> newPath = path;
                newPath.push_back({nx, ny});
                q.push(newPath);
                visited.insert({nx, ny});
            }
        }
    }
}

// **A* Search Algorithm**
struct Node {
    int x, y, g, h;
    vector<pair<int, int>> path;

    Node(int _x, int _y, int _g, int _h, vector<pair<int,
int>> _path)
        : x(_x), y(_y), g(_g), h(_h), path(_path) {}

    bool operator>(const Node &other) const {
        return (g + h) > (other.g + other.h);
    }
};

// Heuristic function (Manhattan Distance)
int heuristic(int x, int y) {
    return abs(x - goal.first) + abs(y - goal.second);
}
```

```
// A* Algorithm
void aStar() {
    cout << "A* Path: ";
    priority_queue<Node, vector<Node>, greater<Node>> pq;
    set<pair<int, int>> visited;

    pq.push(Node(start.first, start.second, 0,
heuristic(start.first, start.second), {start}));

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

        if (current.x == goal.first && current.y ==
goal.second) {
            printPath(current.path);
            return;
        }

        if (visited.find({current.x, current.y}) !=
visited.end()) continue;
        visited.insert({current.x, current.y});

        for (auto &dir : directions) {
            int nx = current.x + dir.first, ny = current.y +
dir.second;
            if (isValid(nx, ny, *new vector<vector<bool>>(N,
vector<bool>(N, false))) && visited.find({nx, ny}) ==
visited.end()) {
                vector<pair<int, int>> newPath =
current.path;
                newPath.push_back({nx, ny});
                pq.push(Node(nx, ny, current.g + 1,
heuristic(nx, ny), newPath));
            }
        }
    }
}

int main() {
    cout << "Maze Solving Algorithms\n";
    dfs();
    bfs();
    aStar();
    return 0;
}
```



```
Maze Solving Algorithms
DFS Path: (0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (2, 2) -> (3, 2) ->
    (4, 2) -> (4, 3) -> (4, 4) -> Goal
(0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (2, 2) -> (1, 2) -> (0, 2) ->
    (0, 3) -> (0, 4) -> (1, 4) -> (2, 4) -> (3, 4) -> (4, 4) -> Goal
BFS Path: (0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (2, 2) -> (3, 2) ->
    (4, 2) -> (4, 3) -> (4, 4) -> Goal
A* Path: (0, 0) -> (1, 0) -> (2, 0) -> (2, 1) -> (2, 2) -> (3, 2) -> (4
    , 2) -> (4, 3) -> (4, 4) -> Goal
```