Bansilal Ramnath Agarwal Charitable Trust's
VISHWAKARMA INSTITUTE OF TECHNOLOGY – PUNE

ARTIFICIAL INTELLIGENCE

Division	A
Batch	2
GR-no	12311493
Roll no	54
Name	Atharva Kangralkar

Assignment 3:

1. Write a program to implement Best first Search algoritham.
2. Write a program to solve 8 puzzel problem using A* algoritham.

1. **Write a program to implement Best first Search algoritham.**

Description:-

Best First Search is a greedy search algorithm that selects the next node to explore based on the
lowest heuristic value. It aims to reach the goal node as quickly as possible by expanding the
most promising node first.

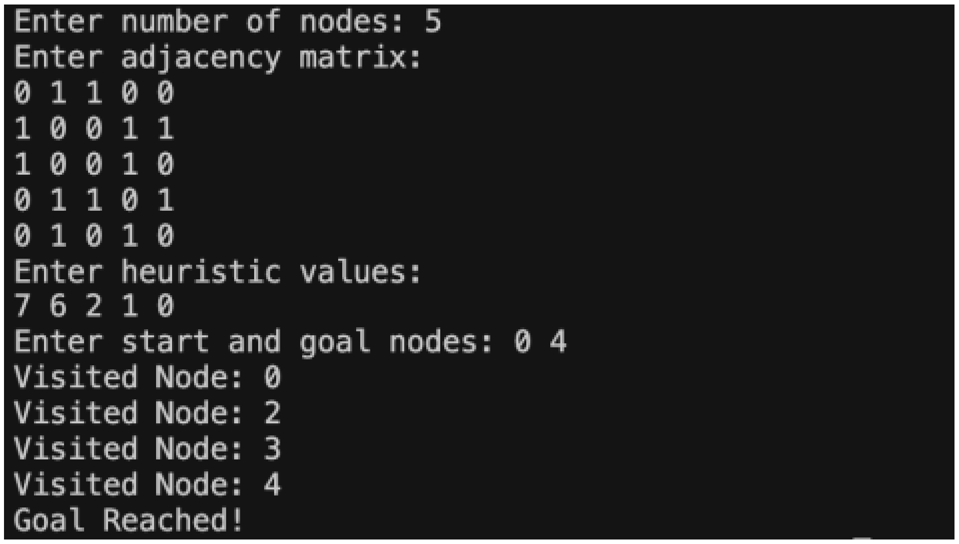
Working

1. Start from the initial node, add it to the open list.
2. Select the node with the lowest heuristic value.
3. If it is the goal node, terminate.
4. Otherwise, move it to the closed list and expand its neighbors.
5. Repeat until the goal is reached or no nodes remain.

- **Code:**

```c
# include <stdio.h>
#include <stdlib.h>
#define MAX 20
int graph[MAX][MAX], visited[MAX], n;
int find_min(int heuristic[], int open[], int size) {
        int min = 9999, index = -1;
        for (int i = 0; i < size; i++) {
                if (open[i] != -1 && heuristic[open[i]] < min) {
                        min = heuristic[open[i]];
                        index = open[i];
                }
        }
        return index;
}
void best_first_search(int start, int goal, int heuristic[]) {
        int open[MAX], closed[MAX], open_size = 0, closed_size = 0;

        open[open_size++] = start;

        while (open_size > 0) {
                int current = find_min(heuristic, open, open_size);
                printf("Visited Node: %d\n", current);
                if (current == goal) {
                        printf("Goal Reached!\n");
                        return;
                }
                for (int i = 0; i < open_size; i++) {
                        if (open[i] == current) {
                                open[i] = -1;
                                break;
                        }
                }
                closed[closed_size++] = current;
                for (int i = 0; i < n; i++) {
                        if (graph[current][i] == 1 && visited[i] == 0) {
                                open[open_size++] = i;
                                visited[i] = 1;
                        }
                }
        }
}
int main() {
        int start, goal, heuristic[MAX];
        printf("Enter number of nodes: ");
        scanf("%d", &n);
        printf("Enter adjacency matrix:\n");
        for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                        scanf("%d", &graph[i][j]);
        printf("Enter heuristic values:\n");
        for (int i = 0; i < n; i++)
                scanf("%d", &heuristic[i]);
        printf("Enter start and goal nodes: ");
        scanf("%d %d", &start, &goal);
        best_first_search(start, goal, heuristic);

        return 0;
}
```

Screenshots/Output:



```
Enter number of nodes: 5
Enter adjacency matrix:
0 1 1 0 0
1 0 0 1 1
1 0 0 1 0
0 1 1 0 1
0 1 0 1 0
Enter heuristic values:
7 6 2 1 0
Enter start and goal nodes: 0 4
Visited Node: 0
Visited Node: 2
Visited Node: 3
Visited Node: 4
Goal Reached!
```

2. Write a program to solve 8 puzzel problem using A* algoritham.

Description:-
A* is a pathfinding algorithm that combines the actual cost (g) to reach a node and a heuristic estimate (h) of the cost to reach the goal, using the formula $f = g + h$. It expands the node with the lowest f value first, balancing between the cheapest path so far and the estimated cost to finish. This strategy efficiently narrows the search space and can guarantee an optimal solution if the heuristic is admissible. A* is widely used in puzzles and graph-based problems due to its efficiency and accuracy. It leverages both uniform-cost search and greedy best-first search principles.

The code maintains nodes with g, h, and f values and uses a priority queue to

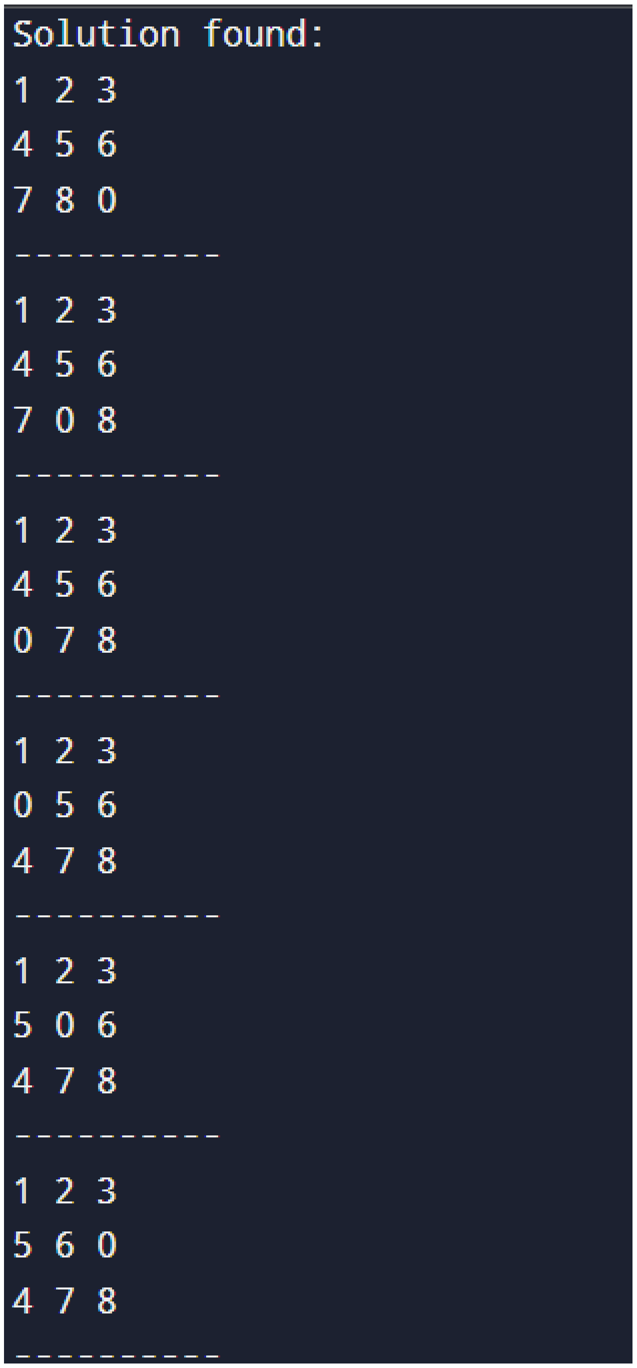
expand the most promising node based on its f value. It tracks visited states and iteratively expands neighbors until the goal state is reached.3. Expand possible moves (up, down, left, right) if valid.
4. Select the move with the lowest f(n).
5. Repeat until the goal state is reached or no solution exists.

- **Code:**

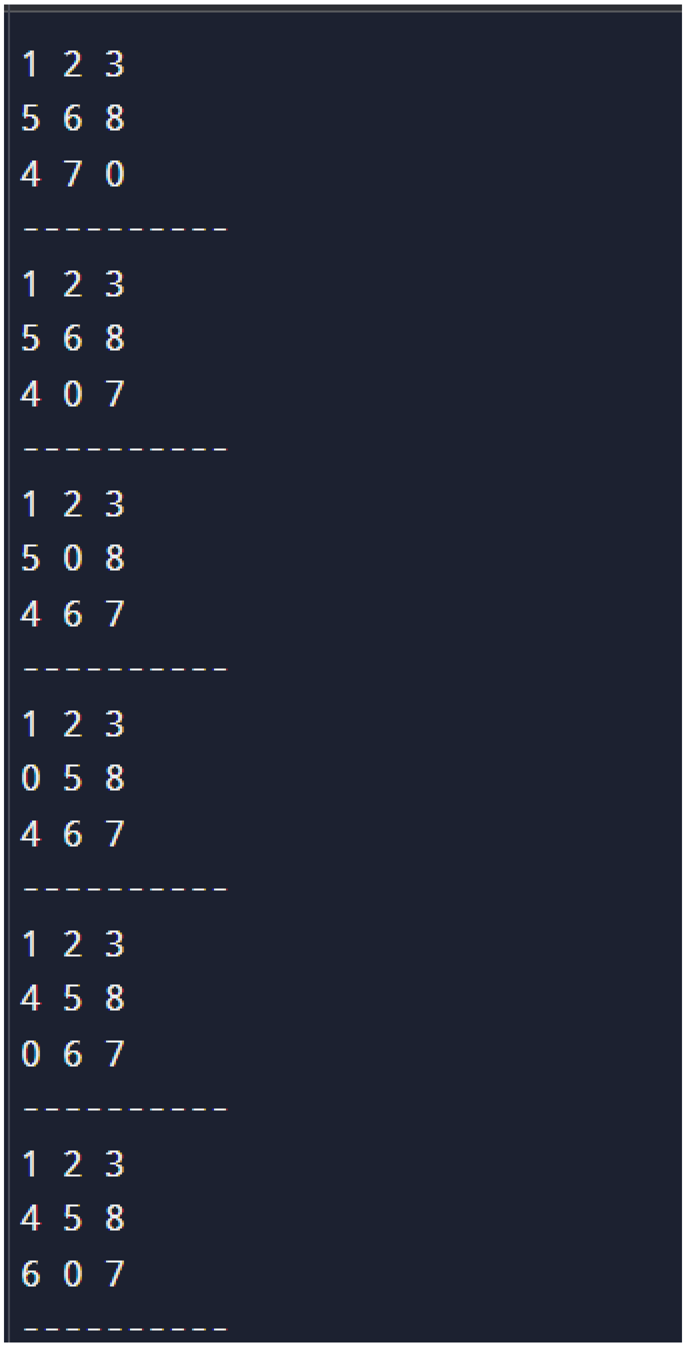
```
#include <iostream>
#include <queue>
#include <set>
#include <cstring>
#include <string>
#include <vector>

using namespace std;

struct Node {
    int state[3][3];
    int g, h, f;
    Node* parent;
};

int goal[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};

int heuristic(int state[3][3]) {
    int h = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (state[i][j] != 0 && state[i][j] != goal[i][j])
                h++;
        }
    }
    return h;
}
```

```
bool isValid(int x, int y) {
    return x >= 0 && x < 3 && y >= 0 && y < 3;
}

void printState(int state[3][3]) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++)
            cout << state[i][j] << " ";
        cout << endl;
    }
    cout << "----------" << endl;
}

string getStateString(int state[3][3]) {
    string str;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            str += to_string(state[i][j]) + ",";
        }
    }
    return str;
}

vector<Node*> getNeighbors(Node* node) {
    vector<Node*> neighbors;
    int x, y;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (node->state[i][j] == 0) {
                x = i, y = j;
            }
        }
    }
    for (int i = 0; i < 4; i++) {
        int newX = x + dx[i], newY = y + dy[i];
        if (isValid(newX, newY)) {
            Node* neighbor = new Node;
            memcpy(neighbor->state, node->state, sizeof(node->state));
            swap(neighbor->state[x][y], neighbor->state[newX][newY]);
            neighbor->g = node->g + 1;
            neighbor->h = heuristic(neighbor->state);
            neighbor->f = neighbor->g + neighbor->h;
            neighbor->parent = node;
            neighbors.push_back(neighbor);
        }
    }
    return neighbors;
}

void solve(int start[3][3]) {
    auto cmp = [](Node* a, Node* b) { return a->f > b->f; };
    priority_queue<Node*, vector<Node*>, decltype(cmp)> pq(cmp);
    set<string> visited;
    Node* root = new Node;
    memcpy(root->state, start, sizeof(root->state));
    root->g = 0;
    root->h = heuristic(root->state);
    root->f = root->g + root->h;
    root->parent = nullptr;
    pq.push(root);

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

        if (memcmp(current->state, goal, sizeof(goal)) == 0) {
            cout << "Solution found:\n";
            while (current) {
                printState(current->state);
                current = current->parent;
            }
            return;
        }

        string stateStr = getStateString(current->state);
        visited.insert(stateStr);
        for (Node* neighbor : getNeighbors(current)) {
            string neighborStr = getStateString(neighbor->state);
            if (visited.find(neighborStr) == visited.end()) {
                pq.push(neighbor);
            }
        }
    }
    cout << "No solution found." << endl;
}

int main() {
    int start[3][3] = {{1, 2, 3}, {4, 0, 5}, {6, 7, 8}};
    solve(start);
    return 0;
}
```

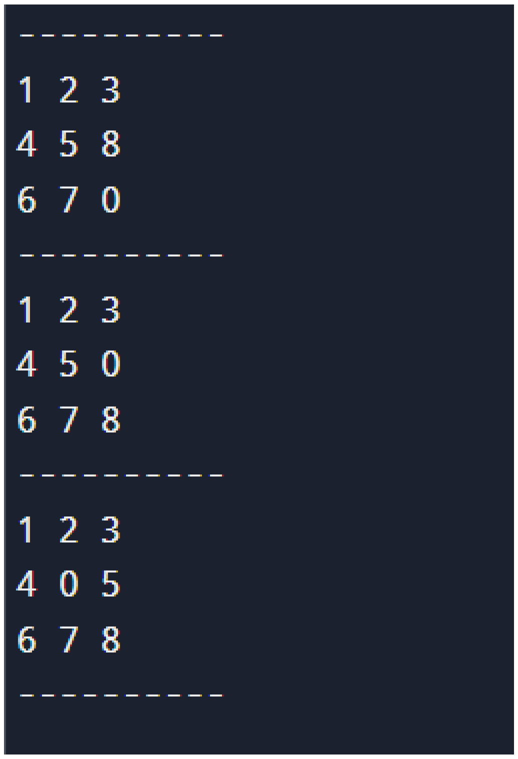
Screenshots/Output:



```
Solution found:
1 2 3
4 5 6
7 8 0
----------
1 2 3
4 5 6
7 0 8
----------
1 2 3
4 5 6
0 7 8
----------
1 2 3
0 5 6
4 7 8
----------
1 2 3
5 0 6
4 7 8
----------
1 2 3
5 6 0
4 7 8
----------
```



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



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