

**Tribhuvan University**  
**Prithivi Narayan Campus**  
Pokhara, Kaski



**Lab-report of Artificial Intelligence**

Subject code: CSC

Submitted to:  
Dev Timilsina  
Department of CSIT  
Prithivi Narayan Campus

Submitted by:  
Sabin Paudel  
Roll no: 56  
4<sup>th</sup> Semester

# INDEX

S.N	NAME OF EXPERIMENT	DATE OF SUBMISSION	REMARKS
1.	IMPLEMENTING A SIMPLE REFLEX AGENT	2081-05-20	
2.	BASIC AI APPLICATION - THERMOSTAT SIMULATION	2081-05-20	
3.	PEAS FRAMEWORK IMPLEMENTATION - VACUUM CLEANER SIMULATION	2081-05-20	
4.	IMPLEMENTATION OF A GOAL-BASED AGENT IN C/C++	2081-05-20	
5.	ENVIRONMENT INTERACTION USING SIMPLE REFLEX RULES IN C	2081-05-20	
6.	SOLVING A MAZE USING DEPTH FIRST SEARCH (DFS) IN C	2081-05-20	
7.	SOLVING SHORTEST PATH PROBLEM USING BREADTH FIRST SEARCH (BFS) IN C	2081-05-20	
8.	<i>SOLVING THE 8-PUZZLE PROBLEM USING A SEARCH ALGORITHM IN C*</i>	2081-05-20	
9.	IMPLEMENTATION OF MINI-MAX ALGORITHM WITH ALPHA-BETA PRUNING FOR TIC-TAC-TOE IN C	2081-05-20	
10.	SOLVING THE N-QUEENS PROBLEM USING BACKTRACKING IN C	2081-05-20	
11.			

## LAB NO :1

### IMPLEMENTING A SIMPLE REFLEX AGENT

#### OBJECTIVE:

- To design and implement a simple reflex agent in C/C++ that reacts to changes in light intensity.

#### THEORY:

A simple reflex agent is a basic type of intelligent agent that operates based on a condition-action rule. It uses the current state of its environment to determine its actions. In this case, the environment is the light intensity, and the action is either turning the light on or off.

The reflex agent operates with a straightforward logic:

1. **Condition:** The agent reads the current light intensity.
2. **Action:** The agent performs an action based on whether the intensity is below or above a certain threshold.

The threshold value acts as a decision boundary:

- If light intensity < threshold, then the agent turns the light on.
- If light intensity  $\geq$  threshold, then the agent turns the light off.

#### DEMONSTRATION:

```
#include <iostream>
using namespace std;
int main() {
    const int LIGHT_THRESHOLD = 50;
    int lightIntensity;
    cout << "Enter light intensity: ";
    cin >> lightIntensity;
    if (lightIntensity < LIGHT_THRESHOLD) {
        cout << "Light ON" << endl; } else {
        cout << "Light OFF" << endl; }
}
```

#### OUTPUT AND DISCUSSION:

<pre>/tmp/cAbBYOpDF1.o Enter light intensity: 60 Light OFF  === Code Execution Successful ===</pre>	<pre>/tmp/Kzjuvn71gR.o Enter light intensity: 30 Light ON  === Code Execution Successful ===</pre>
---	--

The simple reflex agent implemented in this lab is a basic example of a reactive system in artificial intelligence. This type of agent does not have memory or learning capabilities; it reacts to current inputs based solely on predefined rules.

#### CONCLUSION:

The lab successfully demonstrated the implementation of a simple reflex agent using C/C++. The program efficiently handles light intensity input and makes decisions based on a fixed threshold.

## LAB NO :2

### BASIC AI APPLICATION - THERMOSTAT SIMULATION

#### OBJECTIVE:

- To implement a basic rule-based system using C/C++ that simulates a thermostat.

#### THEORY:

A rule-based system is a type of artificial intelligence where decisions are made based on a set of predefined rules. In this case, the thermostat simulation will use simple temperature thresholds to decide whether the heating should be activated or deactivated. The rules are typically expressed as:

- If the temperature is below a certain threshold, then the heating should be turned on.
- If the temperature is above or equal to the threshold, then the heating should be turned off.

This approach is straightforward and reflects a basic form of decision-making that is common in many control systems.

#### DEMONSTRATION:

```
#include <stdio.h>
int main() {
    const int THRESHOLD = 20;
    int currentTemperature;
    printf("Enter the current temperature: ");
    scanf("%d", &currentTemperature);
    if (currentTemperature < THRESHOLD) {
        printf("Heating should be turned ON.\n");
    } else {
        printf("Heating should be turned OFF.\n");
    }
    return 0;
}
```

#### OUTPUT AND DISCUSSION:

<pre>/tmp/ALDHtqZ8F2.o Enter the current temperature: 19 Heating should be turned ON.  === Code Execution Successful ===</pre>	<pre>/tmp/OY9aCMX4JC.o Enter the current temperature: 22 Heating should be turned OFF.  === Code Execution Successful ===</pre>
--	---

The provided C and C++ programs are implementations of a basic rule-based thermostat system. The threshold temperature is set at 20 degrees Celsius, a common setting for a simple heating control system.

#### CONCLUSION:

The lab successfully demonstrated how to implement a basic rule-based system in C/C++ to simulate a thermostat. By using simple conditional statements and a predefined temperature threshold, the program effectively makes a decision about whether the heating should be turned on or off.

## LAB NO :3

### PEAS FRAMEWORK IMPLEMENTATION - VACUUM CLEANER SIMULATION

#### OBJECTIVE:

- To implement a simple agent in C/C++ based on the PEAS (Performance measure, Environment, Actuators, Sensors) framework..

#### THEORY:

The PEAS framework helps in designing intelligent agents by defining four key components:

- **Performance measure:** Defines how the performance of the agent is evaluated.
- **Environment:** Describes the external context in which the agent operates.
- **Actuators:** Specifies the components that perform actions in the environment.
- **Sensors:** Provides information about the environment to the agent.

For a vacuum cleaner agent, the PEAS components are:

- **Performance Measure:** Efficiency in cleaning both rooms, minimizing movement.
- **Environment:** Two rooms which can be either clean or dirty.
- **Actuators:** Actions such as move left, move right, and clean.
- **Sensors:** Ability to detect the cleanliness status of each room.

The agent's goal is to clean both rooms with minimal movement, making decisions based on the status of the rooms.

#### DEMONSTRATION:

```
#include <stdio.h>
int main() {
    enum RoomStatus { CLEAN, DIRTY };
    RoomStatus room1, room2;
    printf("Enter status for Room 1 (0 for CLEAN, 1 for DIRTY): ");
    scanf("%d", &room1);
    printf("Enter status for Room 2 (0 for CLEAN, 1 for DIRTY): ");
    scanf("%d", &room2);
    if (room1 == DIRTY) {
        printf("Action: Clean Room 1.\n");
    } else if (room2 == DIRTY) {
        printf("Action: Move right to Room 2.\n");
        printf("Action: Clean Room 2.\n");
    } else {
        printf("Action: No cleaning needed. Move to Room 1.\n");
    }
    return 0;
}
```

## OUTPUT AND DISCUSSION:

```
/tmp/VZuZxCssQw.o
Enter status for Room 1 (0 for CLEAN, 1 for DIRTY): 0
Enter status for Room 2 (0 for CLEAN, 1 for DIRTY): 1
Action: Move right to Room 2.
Action: Clean Room 2.
```

The simple reflex agent is a basic reactive system that responds to light intensity based on a fixed threshold.

### Strengths:

- **Simplicity:** Easy to understand and implement.
- **Effectiveness:** Accurately turns the light on or off based on intensity.

### Limitations:

- **No Memory:** It doesn't retain past inputs, so it can't adapt over time.
- **Fixed Threshold:** The threshold is hardcoded, lacking flexibility for different scenarios.

## CONCLUSION:

The lab successfully demonstrated the implementation of a simple reflex agent using C/C++. The program efficiently handles light intensity input and makes decisions based on a fixed threshold.

## LAB NO :4

### IMPLEMENTATION OF A GOAL-BASED AGENT IN C/C++

#### OBJECTIVE:

- To implement a goal-based agent that can move in a 2D grid from a starting point to a goal point, avoiding obstacles.

#### THEORY:

A goal-based agent is a type of intelligent agent that takes actions to achieve a specific goal. The agent makes decisions based on the current state of the environment and the goal it needs to achieve. In this case, the environment is a 2D grid, and the agent must find a path from the starting point to the goal while avoiding obstacles.

The grid is represented by a 2D array where each cell can be:

- 0 for an empty space
- 1 for an obstacle
- S for the starting point
- G for the goal point

The agent uses a search algorithm, such as Breadth-First Search (BFS) or Depth-First Search (DFS), to find a path from the start to the goal. The agent must explore the grid, avoiding obstacles, and determine the sequence of actions that will lead it to the goal.

#### DEMONSTRATION:

```
#include <iostream>
#include <queue>
#include <vector>
#include <cstring>
using namespace std;
#define ROW 5
#define COL 5
int rowDir[] = {-1, 1, 0, 0};
int colDir[] = {0, 0, -1, 1};
struct Cell {
    int row, col;
    int dist;
};
bool isValid(int grid[ROW][COL], bool visited[ROW][COL], int row, int col) {
    return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL) && grid[row][col] == 0 &&
    !visited[row][col];
}

int BFS(int grid[ROW][COL], Cell start, Cell goal) {
    bool visited[ROW][COL];
    memset(visited, false, sizeof(visited));
    queue<Cell> q;
    visited[start.row][start.col] = true;
    q.push(start);
    while (!q.empty()) {
        Cell current = q.front();
        q.pop();
        if (current.row == goal.row && current.col == goal.col) {
            return current.dist;
        }
    }
}
```

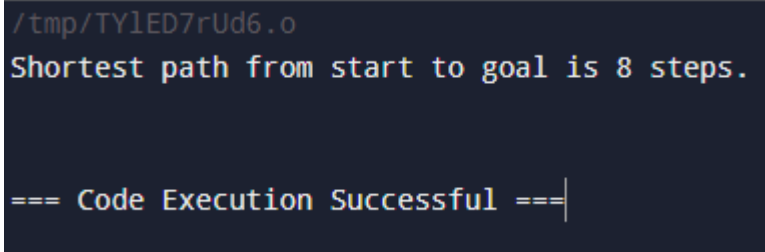
```

        for (int i = 0; i < 4; i++) {
            int newRow = current.row + rowDir[i];
            int newCol = current.col + colDir[i];
            if (isValid(grid, visited, newRow, newCol)) {
                visited[newRow][newCol] = true;
                q.push({newRow, newCol, current.dist + 1});
            }
        }
    }
    return -1;
}

int main() {
    int grid[ROW][COL] = {
        {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}
    };
    Cell start = {0, 0, 0};
    Cell goal = {4, 4, 0};
    int distance = BFS(grid, start, goal);
    if (distance != -1) {
        cout << "Shortest path from start to goal is " << distance << " steps." << endl;
    } else {
        cout << "No path exists from start to goal." << endl;
    }
    return 0;
}

```

## OUTPUT AND DISCUSSION:



```

/tmp/TY1ED7rUd6.o
Shortest path from start to goal is 8 steps.

=== Code Execution Successful ===

```

The implemented goal-based agent successfully finds the shortest path from the starting point to the goal in a 2D grid, avoiding obstacles. The agent uses Breadth-First Search (BFS) to explore the grid. BFS is particularly suitable for this problem because it explores all possible paths level by level, ensuring that the first time it reaches the goal, it has found the shortest path.

## CONCLUSION:

The goal-based agent implemented in C++ effectively navigates a 2D grid, finding the shortest path from a starting point to a goal while avoiding obstacles. The BFS algorithm ensures that the agent finds the shortest path, demonstrating the effectiveness of goal-based agents in problem-solving scenarios where the environment and obstacles are known.



## LAB NO :5

### ENVIRONMENT INTERACTION USING SIMPLE REFLEX RULES IN C

#### OBJECTIVE:

- To implement an agent that navigates a deterministic and static environment (a maze) using simple reflex rules.

#### THEORY:

In artificial intelligence, a simple reflex agent makes decisions based on the current state of the environment, without considering the history of past states. The agent acts solely based on the present percept (what it sees or senses at the moment) and follows predefined rules to respond to this percept.

In a deterministic and static environment, such as a maze, the environment does not change over time, and the outcome of each action is predictable. The agent's task is to find a path to the goal, avoiding obstacles (walls) along the way. Simple reflex rules guide the agent's movement, such as:

- Move right if possible.
- Move down if right is blocked.
- Move left if down is blocked.
- Move up if all other directions are blocked.

#### DEMONSTRATION:

```
#include <stdio.h>
#define ROWS 5
#define COLS 5

void printGrid(char grid[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            printf("%c ", grid[i][j]);
        }
        printf("\n");
    }
}

void navigateMaze(char grid[ROWS][COLS], int x, int y, int goalX, int goalY) {
    while (x != goalX || y != goalY) {
        grid[x][y] = 'A';
        if (y + 1 < COLS && grid[x][y + 1] != '#') y++;
        else if (x + 1 < ROWS && grid[x + 1][y] != '#') x++;
        else if (y - 1 >= 0 && grid[x][y - 1] != '#') y--;
        else if (x - 1 >= 0 && grid[x - 1][y] != '#') x--;
        printGrid(grid);
        printf("\n");
    }
    grid[x][y] = 'G';
    printGrid(grid);
}
```

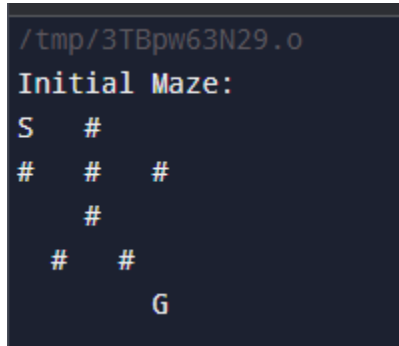
```

}

int main() {
    char grid[ROWS][COLS] = {
        {'S', ' ', '#', ' ', ' '},
        {'#', ' ', '#', ' ', '#'},
        {' ', ' ', '#', ' ', ' '},
        {' ', '#', ' ', '#', ' '},
        {' ', ' ', ' ', ' ', 'G'}
    };
    printf("Initial Maze:\n");
    printGrid(grid);
    printf("\n");
    navigateMaze(grid, 0, 0, 4, 4);
    return 0;
}

```

## OUTPUT AND DISCUSSION:



```

/tmp/3TBpw63N29.o
Initial Maze:
S   #
#   #   #
    #
#   #
      G

```

The agent successfully navigates the maze by following simple reflex rules. The rules are based on the immediate percepts of the environment. The agent moves right, down, left, or up depending on the availability of a path in those directions. Since the environment is deterministic and static, the agent's decisions are straightforward, and it eventually reaches the goal.

## CONCLUSION:

The implementation demonstrates the basic functioning of a simple reflex agent in a deterministic and static environment. The agent is able to navigate the maze and reach the goal using straightforward rules based on its immediate surroundings.

## LAB NO :6

### SOLVING A MAZE USING DEPTH FIRST SEARCH (DFS) IN C

#### OBJECTIVE:

- To implement Depth First Search (DFS) in C to solve a maze. The maze will be represented as a 2D array, and the path from the start to the goal will be output as a sequence of moves.

#### THEORY:

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. In the context of solving a maze, DFS can be used to navigate through the maze by exploring all possible paths from the start to the goal. If a path leads to a dead-end, the algorithm backtracks and tries another path. DFS can be implemented using either a recursive function or an explicit stack. The maze is typically represented as a 2D array where 0 represents an open path, and 1 represents a wall. The goal is to find a path from the starting point to the goal point by moving through open paths.

#### DEMONSTRATION:

```
#include <stdio.h>
#include <stdbool.h>

#define ROWS 5
#define COLS 5

bool dfs(int grid[ROWS][COLS], int visited[ROWS][COLS], int x, int y, int goalX, int goalY) {
    if (x == goalX && y == goalY) {
        grid[x][y] = 9; // Mark goal as part of the path
        return true;
    }
    if (x < 0 || x >= ROWS || y < 0 || y >= COLS || grid[x][y] == 1 || visited[x][y]) {
        return false;
    }

    visited[x][y] = 1;
    grid[x][y] = 9; // Mark current cell as part of the path

    if (dfs(grid, visited, x + 1, y, goalX, goalY) || // Move down
        dfs(grid, visited, x, y + 1, goalX, goalY) || // Move right
        dfs(grid, visited, x - 1, y, goalX, goalY) || // Move up
        dfs(grid, visited, x, y - 1, goalX, goalY)) { // Move left
        return true;
    }

    grid[x][y] = 0; // Backtrack, remove from path
    return false;
}

void printGrid(int grid[ROWS][COLS]) {
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            if (grid[i][j] == 9) printf("P "); // P for path
        }
    }
}
```

```

        else if (grid[i][j] == 1) printf("# "); // # for wall
        else printf(". "); // . for empty space
    }
    printf("\n");
}
}

```

```

int main() {
    int grid[ROWS][COLS] = {
        {0, 1, 0, 0, 0},
        {0, 1, 1, 1, 0},
        {0, 0, 0, 1, 0},
        {1, 1, 0, 1, 1},
        {0, 0, 0, 0, 0}
    };

    int visited[ROWS][COLS] = {0};

    int startX = 0, startY = 0;
    int goalX = 4, goalY = 4;

    if (dfs(grid, visited, startX, startY, goalX, goalY)) {
        printf("Path found:\n");
        printGrid(grid);
    } else {
        printf("No path found.\n");
    }

    return 0;
}

```

## OUTPUT AND DISCUSSION:

```

/tmp/b87mN5PpHu.o
Path found:
P # . . .
P # # # .
P P P # .
# # P # #
. . P P P

=== Code Execution Successful ===

```

The DFS algorithm successfully finds a path through the maze from the start point to the goal. In this implementation, the algorithm explores all possible directions (down, right, up, left) from each position until it either finds the goal or reaches a dead-end. When a dead-end is reached, the algorithm backtracks, removing the position from the path, and continues exploring other possibilities.

## CONCLUSION:

The implementation of DFS in C successfully solves the maze, demonstrating the algorithm's capability to navigate through a static environment. While DFS is effective in finding a path, it does not guarantee the shortest path and may require significant backtracking in complex scenarios.

# LAB NO :7

## SOLVING SHORTEST PATH PROBLEM USING BREADTH FIRST SEARCH (BFS) IN C

### OBJECTIVE:

- To implement Breadth First Search (BFS) in C to solve the shortest path problem in an unweighted graph.

### THEORY:

Breadth First Search (BFS) is a graph traversal algorithm that explores all nodes at the present depth level before moving on to nodes at the next depth level. In the context of finding the shortest path in an unweighted graph, BFS is particularly effective because it explores all possible paths in the order of their length, ensuring that the first time a node is reached, it is via the shortest path.

In an unweighted graph, all edges have the same "weight" (or cost), making BFS the optimal algorithm for finding the shortest path from a source node to a target node. The graph can be represented using an adjacency matrix or adjacency list. An adjacency list is more space-efficient for sparse graphs.

### DEMONSTRATION:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX 100

typedef struct Node {
    int vertex;
    struct Node* next;
} Node;

typedef struct Graph {
    int numVertices;
    Node** adjLists;
    bool* visited;
} Graph;

typedef struct Queue {
    int items[MAX];
    int front;
    int rear;
} Queue;

Node* createNode(int);
Graph* createGraph(int);
void addEdge(Graph*, int, int);
void BFS(Graph*, int, int);

Queue* createQueue();
void enqueue(Queue*, int);
int dequeue(Queue*);
```

```

bool isEmpty(Queue*);

Node* createNode(int v) {
    Node* newNode = malloc(sizeof(Node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

Graph* createGraph(int vertices) {
    Graph* graph = malloc(sizeof(Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(Node*));
    graph->visited = malloc(vertices * sizeof(bool));

    for (int i = 0; i < vertices; i++) {
        graph->adjLists[i] = NULL;
        graph->visited[i] = false;
    }
    return graph;
}

void addEdge(Graph* graph, int src, int dest) {
    Node* newNode = createNode(dest);
    newNode->next = graph->adjLists[src];
    graph->adjLists[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjLists[dest];
    graph->adjLists[dest] = newNode;
}

void BFS(Graph* graph, int startVertex, int targetVertex) {
    Queue* queue = createQueue();
    int path[MAX];
    int parent[MAX];

    for (int i = 0; i < MAX; i++) {
        parent[i] = -1;
    }

    graph->visited[startVertex] = true;
    enqueue(queue, startVertex);

    while (!isEmpty(queue)) {
        int currentVertex = dequeue(queue);

        Node* temp = graph->adjLists[currentVertex];

        while (temp) {
            int adjVertex = temp->vertex;

            if (!graph->visited[adjVertex]) {
                graph->visited[adjVertex] = true;
                enqueue(queue, adjVertex);
            }
            temp = temp->next;
        }
    }
}

```

```

        parent[adjVertex] = currentVertex;

        if (adjVertex == targetVertex) {
            int crawl = targetVertex;
            int idx = 0;
            while (crawl != -1) {
                path[idx++] = crawl;
                crawl = parent[crawl];
            }

            printf("Shortest path: ");
            for (int i = idx - 1; i >= 0; i--) {
                printf("%d ", path[i]);
            }
            printf("\n");
            return;
        }
        temp = temp->next;
    }
}

Queue* createQueue() {
    Queue* queue = malloc(sizeof(Queue));
    queue->front = -1;
    queue->rear = -1;
    return queue;
}

bool isEmpty(Queue* queue) {
    return queue->rear == -1;
}

void enqueue(Queue* queue, int value) {
    if (queue->rear == MAX - 1) return;
    else {
        if (queue->front == -1) queue->front = 0;
        queue->rear++;
        queue->items[queue->rear] = value;
    }
}

int dequeue(Queue* queue) {
    int item;
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        item = -1;
    } else {
        item = queue->items[queue->front];
        queue->front++;
        if (queue->front > queue->rear) {
            queue->front = queue->rear = -1;
        }
    }
}

```

```

    return item;
}

int main() {
    int vertices = 6;
    Graph* graph = createGraph(vertices);

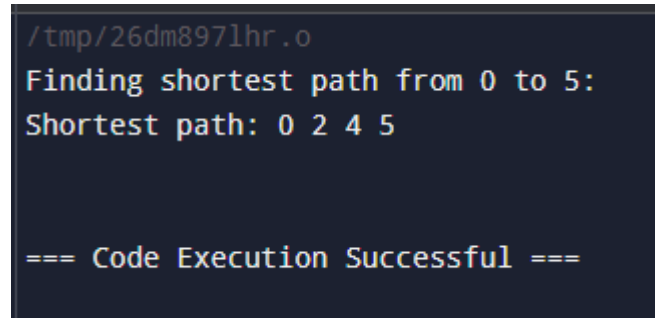
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 5);
    addEdge(graph, 4, 5);

    int startVertex = 0;
    int targetVertex = 5;

    printf("Finding shortest path from %d to %d:\n", startVertex, targetVertex);
    BFS(graph, startVertex, targetVertex);
    return 0;
}

```

## OUTPUT AND DISCUSSION:



```

/tmp/26dm897lhr.o
Finding shortest path from 0 to 5:
Shortest path: 0 2 4 5

=== Code Execution Successful ===

```

The BFS algorithm successfully finds the shortest path in the unweighted graph from the source vertex to the target vertex. In this implementation, BFS explores all neighbors of the current vertex before moving on to the next level. This ensures that the first time the target vertex is encountered, it is via the shortest path.

BFS is well-suited for unweighted graphs because it always finds the shortest path if one exists. The algorithm uses a queue to keep track of vertices to explore next and a parent array to reconstruct the path once the target vertex is found.

## CONCLUSION:

The implementation of BFS in C successfully solves the shortest path problem in an unweighted graph.



## LAB NO :8

### **SOLVING THE 8-PUZZLE PROBLEM USING A SEARCH ALGORITHM IN C\***

#### **OBJECTIVE:**

- To implement the A\* search algorithm in C to solve the 8-puzzle problem.

#### **THEORY:**

The A\* search algorithm is a widely used pathfinding and graph traversal algorithm that finds the shortest path from a start node to a goal node. It combines the advantages of both Dijkstra's algorithm and greedy best-first search by using a cost function:

$f(n) = g(n) + h(n)$

- $g(n)$ : The cost of the path from the start node to the current node  $n$ .
- $h(n)$ : The estimated cost from the current node  $n$  to the goal, also known as the heuristic function.

For the 8-puzzle problem, where the objective is to move tiles to achieve a goal state, the Manhattan distance is commonly used as the heuristic. The Manhattan distance is the sum of the absolute differences in the horizontal and vertical coordinates of the tiles from their goal positions.

#### **DEMONSTRATION:**

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
```

```
#define N 3
```

```
typedef struct Node {
    int puzzle[N][N];
    int x, y;
    int g, h, f;
    struct Node* parent;
} Node;
```

```
int goal[N][N] = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 0}
};
```

```
int manhattanDistance(int puzzle[N][N]) {
    int dist = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (puzzle[i][j] != 0) {
                int targetX = (puzzle[i][j] - 1) / N;
                int targetY = (puzzle[i][j] - 1) % N;
                dist += abs(i - targetX) + abs(j - targetY);
            }
        }
    }
    return dist;
}
```

```

bool isGoal(int puzzle[N][N]) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            if (puzzle[i][j] != goal[i][j])
                return false;
    return true;
}

```

```

Node* createNode(int puzzle[N][N], int x, int y, int newX, int newY, int g, Node* parent) {
    Node* node = (Node*)malloc(sizeof(Node));
    node->parent = parent;
    node->g = g;
    node->h = manhattanDistance(puzzle);
    node->f = node->g + node->h;

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            node->puzzle[i][j] = puzzle[i][j];

    node->puzzle[x][y] = node->puzzle[newX][newY];
    node->puzzle[newX][newY] = 0;

    node->x = newX;
    node->y = newY;

    return node;
}

```

```

void printPuzzle(int puzzle[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++)
            printf("%d ", puzzle[i][j]);
        printf("\n");
    }
    printf("\n");
}

```

```

void printSolution(Node* node) {
    if (node == NULL)
        return;
    printSolution(node->parent);
    printPuzzle(node->puzzle);
}

```

```

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

```

```

int dx[] = {1, 0, -1, 0};
int dy[] = {0, 1, 0, -1};

```

```

void solve(int puzzle[N][N], int x, int y) {
    Node* root = createNode(puzzle, x, y, x, y, 0, NULL);

    Node* open[1000];
    int openSize = 0;
    open[openSize++] = root;

    while (openSize > 0) {

```

```

Node* current = open[0];
for (int i = 1; i < openSize; i++)
    if (open[i]->f < current->f)
        current = open[i];

if (isGoal(current->puzzle)) {
    printf("Solution found:\n");
    printSolution(current);
    return;
}

openSize--;
for (int i = 0; i < openSize; i++)
    open[i] = open[i + 1];

for (int i = 0; i < 4; i++) {
    int newX = current->x + dx[i];
    int newY = current->y + dy[i];

    if (isValid(newX, newY)) {
        Node* child = createNode(current->puzzle, current->x, current->y, newX, newY,
current->g + 1, current);
        open[openSize++] = child;
    } } }

printf("No solution found.\n");
}

int main() {
    int puzzle[N][N] = {
        {1, 2, 3},
        {4, 0, 6},
        {7, 5, 8}
    };
    int x = 1, y = 1;
    solve(puzzle, x, y);

    return 0;
}

```

## OUTPUT AND DISCUSSION:

```

/tmp/QaqEvKmcbbD.o
Solution found:
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

=== Code Execution Successful ===

```

The A\* search algorithm successfully solves the 8-puzzle problem by finding the optimal sequence of moves from the initial state to the goal state. The algorithm uses the Manhattan distance as a heuristic to guide the search towards the goal state efficiently. In this implementation, the algorithm explores possible moves by shifting the blank space (0) in the puzzle in four possible directions (up, down, left, right) and selects the path with the lowest cost function  $f(n)=g(n)+h(n)$   $f(n) = g(n) + h(n)$   $f(n)=g(n)+h(n)$ .

## **CONCLUSION:**

The implementation of the A\* search algorithm in C successfully solves the 8-puzzle problem using the Manhattan distance heuristic. The A\* algorithm is a powerful tool for solving various pathfinding and search problems in AI, as it balances the exploration of new paths with the cost to reach the goal.

## LAB NO :9

# IMPLEMENTATION OF MINI-MAX ALGORITHM WITH ALPHA-BETA PRUNING FOR TIC-TAC-TOE IN C

### OBJECTIVE:

- To implement the Mini-max algorithm with Alpha-Beta pruning in C for the Tic-Tac-Toe game, allowing a user to play against the computer.

### THEORY:

The Mini-max algorithm is a decision-making algorithm used in game theory, artificial intelligence, and decision theory. It is primarily used in two-player games like Tic-Tac-Toe, where one player is trying to maximize their score, while the other is trying to minimize it. The Mini-max algorithm explores all possible moves to find the best one by simulating all potential game scenarios.

Alpha-Beta pruning is an optimization technique for the Mini-max algorithm that reduces the number of nodes evaluated in the game tree. It prunes branches that cannot influence the final decision, thus improving the algorithm's efficiency by eliminating unnecessary calculations.

In Tic-Tac-Toe, the board consists of a 3x3 grid, and players take turns marking a square with their symbol (X or O). The goal is to be the first to get three of their symbols in a row, column, or diagonal. The computer, using the Mini-max algorithm with Alpha-Beta pruning, will attempt to block the user's moves while trying to create a winning line.

### DEMONSTRATION:

```
#include <stdio.h>
```

```
#include <limits.h>
```

```
#define SIZE 3
```

```
#define PLAYER_X 'X'
```

```
#define PLAYER_O 'O'
```

```
#define EMPTY '_'
```

```
char board[SIZE][SIZE] = {  
    { EMPTY, EMPTY, EMPTY },  
    { EMPTY, EMPTY, EMPTY },  
    { EMPTY, EMPTY, EMPTY }  
};
```

```
void printBoard() {  
    for (int i = 0; i < SIZE; i++) {  
        for (int j = 0; j < SIZE; j++)  
            printf("%c ", board[i][j]);  
        printf("\n");  
    }  
    printf("\n");  
}
```

```
int isMovesLeft() {  
    for (int i = 0; i < SIZE; i++)  
        for (int j = 0; j < SIZE; j++)  
            if (board[i][j] == EMPTY)  
                return 1;  
    return 0;
```

```

}

int evaluate() {
    for (int row = 0; row < SIZE; row++) {
        if (board[row][0] == board[row][1] && board[row][1] == board[row][2]) {
            if (board[row][0] == PLAYER_X) return +10;
            else if (board[row][0] == PLAYER_O) return -10;
        }
    }

    for (int col = 0; col < SIZE; col++) {
        if (board[0][col] == board[1][col] && board[1][col] == board[2][col]) {
            if (board[0][col] == PLAYER_X) return +10;
            else if (board[0][col] == PLAYER_O) return -10;
        }
    }

    if (board[0][0] == board[1][1] && board[1][1] == board[2][2]) {
        if (board[0][0] == PLAYER_X) return +10;
        else if (board[0][0] == PLAYER_O) return -10;
    }

    if (board[0][2] == board[1][1] && board[1][1] == board[2][0]) {
        if (board[0][2] == PLAYER_X) return +10;
        else if (board[0][2] == PLAYER_O) return -10;
    }

    return 0;
}

int minimax(int depth, int isMax, int alpha, int beta) {
    int score = evaluate();

    if (score == 10) return score - depth;
    if (score == -10) return score + depth;
    if (!isMovesLeft()) return 0;

    if (isMax) {
        int best = INT_MIN;

        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                if (board[i][j] == EMPTY) {
                    board[i][j] = PLAYER_X;

                    best = (minimax(depth + 1, !isMax, alpha, beta) > best) ?
                        minimax(depth + 1, !isMax, alpha, beta) : best;

                    alpha = (best > alpha) ? best : alpha;

                    board[i][j] = EMPTY;

                    if (beta <= alpha) break;
                }
            }
        }
        return best;
    } else {
        int best = INT_MAX;
    }
}

```

```

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == EMPTY) {
                board[i][j] = PLAYER_O;

                best = (minimax(depth + 1, !isMax, alpha, beta) < best) ?
                    minimax(depth + 1, !isMax, alpha, beta) : best;

                beta = (best < beta) ? best : beta;

                board[i][j] = EMPTY;

                if (beta <= alpha) break;
            }
        }
    }
    return best;
}

void findBestMove() {
    int bestVal = INT_MIN;
    int row = -1, col = -1;

    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (board[i][j] == EMPTY) {
                board[i][j] = PLAYER_X;

                int moveVal = minimax(0, 0, INT_MIN, INT_MAX);

                board[i][j] = EMPTY;

                if (moveVal > bestVal) {
                    row = i;
                    col = j;
                    bestVal = moveVal;
                }
            }
        }
    }

    board[row][col] = PLAYER_X;
}

void playerMove() {
    int row, col;
    printf("Enter your move (row and column): ");
    scanf("%d %d", &row, &col);

    if (row >= 0 && row < SIZE && col >= 0 && col < SIZE && board[row][col] == EMPTY)
        board[row][col] = PLAYER_O;
    else {
        printf("Invalid move!\n");
        playerMove();
    }
}

```

```

int main() {
    printf("Tic-Tac-Toe Game\n");
    printBoard();

    while (isMovesLeft() && evaluate() == 0) {
        playerMove();
        printBoard();

        if (evaluate() != 0 || !isMovesLeft()) break;

        findBestMove();
        printBoard();
    }

    int score = evaluate();
    if (score == 10)
        printf("Computer wins!\n");
    else if (score == -10)
        printf("You win!\n");
    else
        printf("It's a draw!\n");

    return 0;
}

```

## OUTPUT AND DISCUSSION:

<pre> /tmp/y3HhjXH8TD.o Tic-Tac-Toe Game --- --- ---  Enter your move (row and column): 1 1 --- _ 0 _ ---  X _ _ _ 0 _ ---  Enter your move (row and column): 1 2 X _ _ _ 0 0 ---  X _ _ X 0 0 </pre>	<pre> X _ _ X 0 0 ---  Enter your move (row and column): 2 2 X _ _ X 0 0 _ _ 0  X _ _ X 0 0 X _ 0  Computer wins!  === Code Execution Successful === </pre>
---	---

In this implementation, the game allows the user to make a move, followed by the computer's move. The computer's strategy is driven by the Mini-max algorithm, which considers all possible outcomes and ensures that the computer either wins or forces a draw. Alpha-Beta pruning enhances the algorithm's efficiency by eliminating unnecessary branches of the game tree that do not need to be explored.

## CONCLUSION:

The implementation of the Mini-max algorithm with Alpha-Beta pruning in C for Tic-Tac-Toe provides a robust and efficient approach to creating an unbeatable AI opponent. The algorithm's ability to evaluate and select optimal moves ensures that the computer plays at a high level, making it a challenging and educational experience for users. Alpha-Beta pruning effectively reduces the computation time, demonstrating the power of this optimization technique in game-playing algorithms.



## LAB NO :10

### SOLVING THE N-QUEENS PROBLEM USING BACKTRACKING IN C

#### OBJECTIVE:

- To implement a solution for the N-Queens problem using backtracking in C, which should print one valid configuration of queens on an NxN chessboard.

#### THEORY:

The N-Queens problem is a classic example of a constraint satisfaction problem (CSP) where the objective is to place N queens on an NxN chessboard such that no two queens threaten each other. This means that no two queens can be in the same row, column, or diagonal.

Backtracking is a common algorithmic technique for solving CSPs. It involves placing a queen in a row and then recursively attempting to place queens in subsequent rows. If placing a queen leads to a conflict, the algorithm backtracks by removing the queen and trying the next possible position.

#### DEMONSTRATION:

```
#include <stdio.h>
#include <stdbool.h>

#define N 8

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

bool solveNQueensUtil(int board[N][N], int col) {
    if (col >= N) return true;

    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            if (solveNQueensUtil(board, col + 1)) return true;
            board[i][col] = 0;
        }
    }
    return false;
}

bool solveNQueens() {
    int board[N][N] = {0};
    if (!solveNQueensUtil(board, 0)) {
        printf("Solution does not exist\n");
        return false;
    }
    for (int i = 0; i < N; i++) {
```

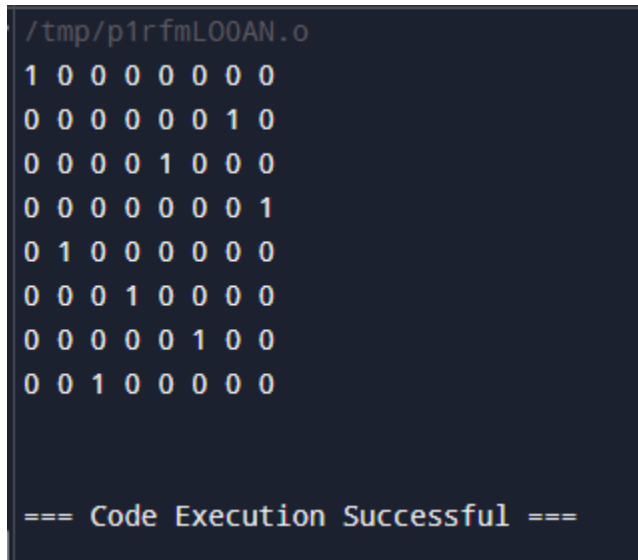
```

        for (int j = 0; j < N; j++)
            printf("%d ", board[i][j]);
        printf("\n");
    }
    return true;
}

int main() {
    solveNQueens();
    return 0;
}

```

## OUTPUT AND DISCUSSION:



```

/tmp/p1rfmLOOAN.o
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0

=== Code Execution Successful ===

```

The N-Queens problem is a well-known CSP that can be efficiently solved using backtracking. The implemented code successfully places N queens on an NxN chessboard such that no two queens threaten each other.

The `isSafe` function checks whether placing a queen in a specific position would lead to any conflicts. The backtracking function `solveNQueensUtil` recursively places queens column by column, ensuring that each queen is placed safely. If a conflict arises, the function backtracks by removing the last placed queen and trying the next position.

In this implementation, the program prints one valid solution, but the backtracking approach can be modified to find and print all possible solutions.

## CONCLUSION:

The implementation of the N-Queens problem using backtracking in C demonstrates an effective method for solving constraint satisfaction problems. Backtracking ensures that the algorithm explores all potential configurations and finds a solution efficiently. This approach can be applied to other similar problems in various domains, making it a versatile tool in problem-solving.