OPEN ENDED EXPERIMENT

Submitted by:

Names:

Mahinur Rahman Mahin (242014165)

Noorjahan Akter (251014048)

Fatema Akter Koli (251014046)

Ratri Saha (243014126)

Course Title: Data Structure Lab

Course Code: CSE 1302

Semester: Fall 2025

Section: 02

Date: 09.12.2025

Submitted to:

Dewan Ahmed Muhtasim

Department of Computer Science & Engineering

Problem: We consider a 3×3 city block grid (A to I).
Each block is connected through Streets (horizontal edges) and Roads (vertical edges). Movement is allowed only through these named paths. The objective is to design a program in C that:

- ❖ Stores the city's road connections
- ❖ Takes a source block and destination block as input .For example B → G
- ❖ Computes any valid path between them
- ❖ Displays the exact Street/Road names used along the path.
  For example Take Street 1 → Road 2 → Street 3

Design:

- **I.** graph[node] → Stores which blocks are connected and by which road.
- II. queue → Used by BFS to visit blocks level-by-level.
- III. visited[] → Marks blocks already checked to avoid repeats.

Algorithm Explanation:

Breadth-First Search (BFS)

- At first we start from the source block and then we use a queue.
- After that we visit neighbors one by one and record parent and the Street/Road name used.
- We have to stop when destination is reached and backtrack to build the full path.

Pseudocode:

```
IF source == destination
 PRINT "Already at same block"
 EXIT
 BFS:
 Mark source visited
 parent[source] = -1
 Enqueue source
```

```
WHILE queue not empty
current = dequeue
 FOR each neighbor of current
 IF connected AND not visited
visited = true
parent[neighbor] = current
 enqueue neighbor
 IF neighbor == destination
  STOP BFS
  ENDIF
  ENDIF
  END FOR
  END WHILE
  IF destination not reached
  PRINT "No path found"
 EXIT
 Build path using parent array (push into stack)
  PRINT "Start at source"
 WHILE stack not empty
 pop next node
  PRINT street/road name
 END WHILE
 PRINT "Arrive at destination"
  END
```

Time Complexity:

- $O(N^2)$

Implementation:

Code :

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#define N 9

int adj[N][N];

char path_names[N][N][20];

char *Block_Name[] = {"A", "B", "C", "D", "E", "F", "G", "H", "I"};
```

Explanation: adj[N][N] is the adjacency matrix. It stores connections between blocks. A value of 1 means connected, 0 means not connected. Path names[N][N][20] stores the name of the street between blocks. Block Name stores the names of blocks from A to I.

```
struct Queue {

int items[N] , front,rear; };

void initQueue(struct Queue *q) {

q->front = -1, q->rear = -1; }

int isEmpty(struct Queue *q) {

return q->rear == -1;  }

void enqueue(struct Queue *q, int value) {

if (q->rear == N - 1) return;

if (q->front == -1) q->front = 0;

q->rear++;

q->items[q->rear] = value;  }

int dequeue(struct Queue *q) {

if (isEmpty(q)) return -1;

int item = q->items[q->front];

q->front++;

if (q->front > q->rear) {

q->front = q->rear = -1;   }

return item;  }

struct Stack {
```

Explanation: Queue structure has items for storing elements, and front and rear for tracking the ends of the queue. InitQueue initializes the queue. IsEmpty checks whether the queue is empty. Enqueue adds a value to the queue. Dequeue removes a value from the queue.

```
int items[N], top; };
void initStack(struct Stack *s) {
s->top = -1;  }
void push(struct Stack *s, int value) {
if (s->top < N - 1) {
s->top++;
s->items[s->top] = value;   } }
int pop(struct Stack *s) {
if (s->top >= 0) {
return s->items[s->top--]; }
return -1;  }
void initializeGraph() {
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
adj[i][j] = 0;
strcpy(path_names[i][j], "");  } }
adj[0][1] = adj[1][0] = 1;
strcpy(path_names[0][1], "Street 1");  strcpy(path_names[1][0], "Street 1");
adj[1][2] = adj[2][1] = 1;
strcpy(path_names[1][2], "Street 1");  strcpy(path_names[2][1], "Street 1");
adj[3][4] = adj[4][3] = 1;
strcpy(path_names[3][4], "Street 2");  strcpy(path_names[4][3], "Street 2");
adj[4][5] = adj[5][4] = 1;
strcpy(path_names[4][5], "Street 2");  strcpy(path_names[5][4], "Street 2");
adj[6][7] = adj[7][6] = 1;
strcpy(path_names[6][7], "Street 3");  strcpy(path_names[7][6], "Street 3");
adj[7][8] = adj[8][7] = 1;
strcpy(path_names[7][8], "Street 3");  strcpy(path_names[8][7], "Street 3");
adj[0][3] = adj[3][0] = 1;
strcpy(path_names[0][3], "Road 1");  strcpy(path_names[3][0], "Road 1");
adj[3][6] = adj[6][3] = 1;
strcpy(path_names[3][6], "Road 1");  strcpy(path_names[6][3], "Road 1");
```

Explanation: Stack structure has items for storing elements and top for the top index.Push adds a value to the stack.Pop removes and returns the top value from the stack. The stack is used to reverse the path so it can be printed from start to destination.

```
adj[1][4] = adj[4][1] = 1;

strcpy(path_names[1][4], "Road 2");  strcpy(path_names[4][1], "Road 2");

adj[4][7] = adj[7][4] = 1;

strcpy(path_names[4][7], "Road 2");  strcpy(path_names[7][4], "Road 2");

adj[2][5] = adj[5][2] = 1;

strcpy(path_names[2][5], "Road 3");  strcpy(path_names[5][2], "Road 3");

adj[5][8] = adj[8][5] = 1;

strcpy(path_names[5][8], "Road 3");

strcpy(path_names[8][5], "Road 3");   }
```

All adjacency matrix values are set to 0 and all path names are set to empty. Connections between blocks are defined and street names are stored. For example, Block A is connected to Block B through Street 1.This setup creates the network of blocks and streets in the city.

```
void findPath(int src, int dest) {

if (src == dest) {

printf("already Block %s!\n", Block_Name[src]);

return;   }

int visited[N] = {0} , parent[N] , found = 0 ; ;

for (int i = 0; i < N; i++) parent[i] = -1;

struct Queue q ;

initQueue(&q);

visited[src] = 1, enqueue(&q, src);

while (!isEmpty(&q) && !found) {

int j = dequeue(&q);

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

if (adj[j][i] == 1 && !visited[i]) {

visited[i] = 1;

parent[i] = j ;

enqueue(&q, i);

if (i == dest) {

found = 1;

break;   }}} }

if (!found) {

printf("No path %s\n", Block_Name[src], Block_Name[dest]);

return;  }

struct   Stack path;
```

```
initStack(&path);
int node = dest;
while (node != -1) {
push(&path, node);
node = parent[node]; }
printf("Start at Block %s,\n", Block_Name[src]);
int prev = pop(&path);
while (path.top >= 0) {
int current = pop(&path);
printf("Take %s,\n", path_names[prev][current]);
prev = current; }
printf("Arrive at Block %s.\n", Block_Name[dest]);  }
int getBlockIndex(char block) {
if (block >= 'a' && block <= 'z') {
block = block - 32;  }
for (int i = 0; i < N; i++) {
if (block == Block_Name[i][0]) {
return i;  } }
return -1;  }
int main() {
initializeGraph();
char source1, source2;
printf("Enter source block (A-I): ");
scanf(" %c", &source1);
printf("Enter destination block (A-I): ");
Hide quoted text
scanf(" %c", &source2);
int a = getBlockIndex(source1), b = getBlockIndex(source2);
if (a == -1 || b == -1) {
printf("Invalid block name!\n");
return 1;  }
printf("\n");
```

If the source and destination are the same, the program prints that the user is already at the block.Arrays visited and parent track which blocks have been visited and the parent of each block for reconstructing the path.BFS starts from the source block and explores neighbors until the destination is found.When the destination is found, the program stops BFS.

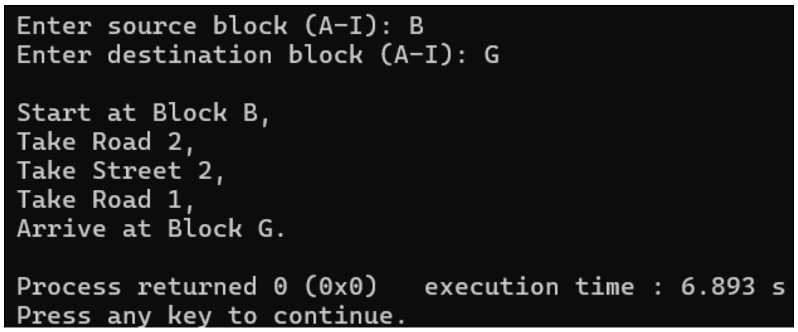
The function converts a block letter (A-I) into an index from 0 to 8.It also converts lowercase letters to uppercase.Returns -1 if the input block is invalid.

Initializes the graph.Takes input from the user for source and destination blocks. Converts the block letters into indices. Checks if the input is valid. Calls the findPath function to find and print the path.

findPath(a, b);

return 0; }

OUTPUT:



```
Enter source block (A-I): B
Enter destination block (A-I): G

Start at Block B,
Take Road 2,
Take Street 2,
Take Road 1,
Arrive at Block G.

Process returned 0 (0x0)   execution time : 6.893 s
Press any key to continue.
```

Pros & Cons:

Advantages :-

- BFS always finds a shortest valid path
- Supports Street/Road name labels
- Simple and efficient structure
- Easy to extend

Disadvantages:-

- Static 3×3 grid
- BFS cannot handle weighted paths
- Removing a connection needs manual array shifting

Alternative solution:

#define MAX 20

char graph[MAX][MAX][2];

int adjCount[MAX];

char blocks[9][2] = {"A","B","C","D","E","F","G","H","I"};

```
int getIndex(char name[]) {
    for (int i = 0; i < 9; i++)
        if (strcmp(blocks[i], name) == 0)
            return i;
    return -1;}
void addEdge(char a[], char b[]) {
    int u = getIndex(a) , v = getIndex(b);
    strcpy(graph[u][adjCount[u]++], b);
    strcpy(graph[v][adjCount[v]++], a); }
void bfs_shortest_path(char start[], char end[]) {
    int startIdx = getIndex(start) , endIdx   = getIndex(end);
    int queue[MAX], front = 0, rear = 0;
    int visited[MAX] = {0} , parent[MAX];
    for (int i = 0; i < MAX; i++) parent[i] = -1;
    queue[rear++] = startIdx;
    visited[startIdx] = 1;
    while (front < rear) {
        int current = queue[front++];
        if (current == endIdx) break;
        for (int i = 0; i < adjCount[current]; i++) {
            int nxt = getIndex(graph[current][i]);
            if (!visited[nxt]) {
                visited[nxt] = 1;
                parent[nxt] = current;
                queue[rear++] = nxt; }}}
    int path[MAX], len = 0,cur = endIdx;
    while (cur != -1) {
        path[len++] = cur;
        cur = parent[cur]; }
    printf("\nShortest Path: ");
    for (int i = len - 1; i >= 0; i--) {
        printf("%s", blocks[path[i]]);
        if (i) printf(" -> "); }
    printf("\n\n"); }
int main() {
    addEdge("A","B"); addEdge("B","C");
    addEdge("D","E"); addEdge("E","F");
    addEdge("G","H"); addEdge("H","I");
    addEdge("A","D"); addEdge("D","G");
    addEdge("B","E"); addEdge("E","H");
    addEdge("C","F"); addEdge("F","I");
    char source[2], dest[2];
    printf("Enter Source Block: ");
    scanf("%s", source);
    printf("Enter Destination Block: ");
    scanf("%s", dest);
    bfs_shortest_path(source, dest);
    return 0; }
```

Conclusion:

In this experiment, we created a small city map using a graph, where each block is a node and each Street/Road is a labeled connection. By using the BFS algorithm, we were able to find a valid and shortest path between any two blocks. The program also shows the exact Street or Road names used in the journey. This project helped us understand how graphs and BFS work in real-life problems, such as city navigation or GPS systems. The solution is simple, clear, and can be easily extended by adding or removing routes. Overall, the experiment successfully shows how data structures can be used to solve practical pathfinding problems.