East West University

Department of CSE

CSE246

Algorithms

Project Name

“Library Book Search System”

Semester & Year

Spring 2024

Name of Student & Id :

Abrar Hossain Zahin

2022-2-60-040

UmmeHaney

2020-2-60-009

SadiaShatabdi

2022-1-60-036

Course Instructor :

Amit Mandal

Lecturer

Department of Computer Science and Engineering, EWU

Date of Report Submitted : 8 June, 2024

1.Problem Statement : Libraries often contain thousands of books spread across numerous shelves, making it challenging for users to locate specific books quickly. Traditional methods of searching for books, such as manually browsing shelves or using simple catalog systems, can be time-consuming and inefficient. The need arises for a system that can not only provide auto-complete suggestions for book titles based on user input but also guide users to the exact location of the book within the library.

2. Objective : The objective of this project is to develop a Library Book Search System that:

- Provides auto-complete suggestions for book titles using binary search.
- Finds the shortest path to a book's location within the library using Breadth-First Search (BFS).

3. Methodology / Flow Chart

Methodology :

1. **Algorithms**:
    - **Binary Search**: Used to search for book titles efficiently. The auto-complete feature suggests book titles based on a given prefix.
    - **Breadth-First Search (BFS)**: Used to find the shortest path from the user's current location to the book's location in the library.
2. **Implementation**:
    - **Book Data Storage**: Store book titles and their locations (row and column) in a file.

- **Binary Search for Title Lookup**: Implement binary search to suggest books based on user-entered prefixes.

- **Shortest Path Calculation**: Implement BFS to guide users from their current location to the book's location.

4.**User Interaction**:

- The system prompts the user to enter a prefix for the book title.

- It provides auto-complete suggestions based on the prefix.

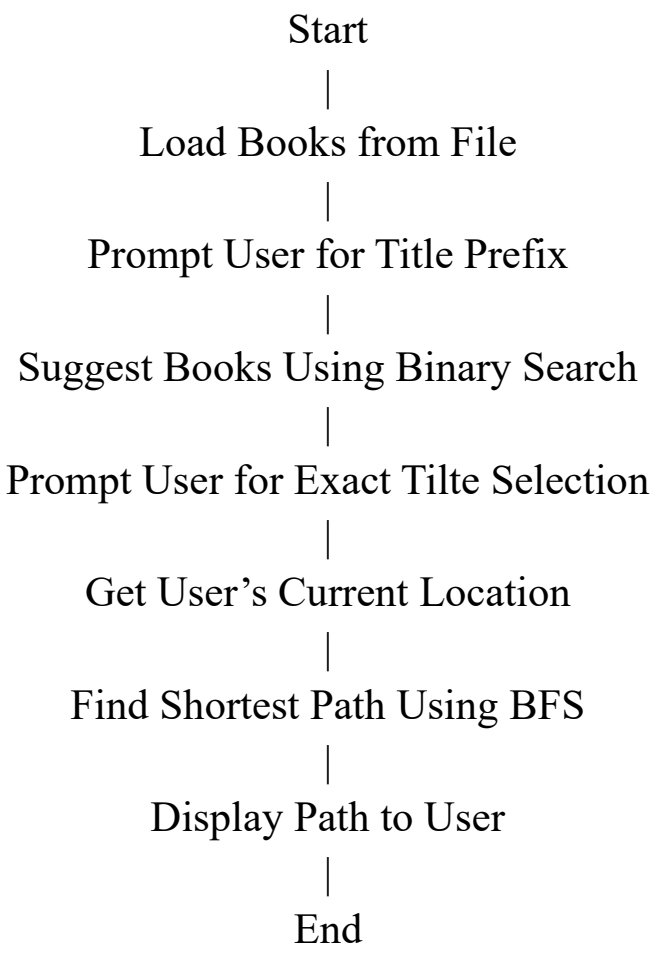
- The user selects the exact book title from the suggestions.

- The system then calculates and displays the shortest path to the book's location.

5.**BFS Implementation**

- **Initialization:** A queue is initialized to store nodes to be explored, and arrays are used to track visited nodes and parent nodes for path reconstruction.
- **Exploration:** The algorithm dequeues the front node, checks if it is the destination, and if not, enqueues its unvisited neighbors.
- **Path Construction:** If the destination is reached, the path is reconstructed using the parent array.

Flow Chart :



4. Limitations

- The system assumes a static library layout. Changes in the layout require updating the library map data.
- The efficiency of BFS may be reduced if the library is very large or has many obstacles.
- The binary search for auto-complete suggestions may not handle large datasets optimally without additional data structures like a trie.

Source Code :

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_ROWS 15
#define MAX_COLS 15
#define MAX_BOOKS 100

// Structure to store book information
typedef struct {
    char title[50];
    int row;
    int col;
} Book;

// Queue implementation for BFS
typedef struct {
    int row;
    int col;
} QueueNode;

typedef struct {
    QueueNode nodes[MAX_ROWS * MAX_COLS];
    int front;
    int rear;
} Queue;

void initQueue(Queue *q) {
    q->front = -1;
    q->rear = -1;
}

int isQueueEmpty(Queue *q) {
    return q->front == -1;
}

void enqueue(Queue *q, int row, int col) {
    if (q->rear == MAX_ROWS * MAX_COLS - 1) {
        printf("Queue overflow\n");
        return; }
    if (isQueueEmpty(q)) {
        q->front = 0;
    }
    q->rear++;
    q->nodes[q->rear].row = row;
    q->nodes[q->rear].col = col;
}
```

```
QueueNode dequeue(Queue *q) {
    if (isQueueEmpty(q)) {
        printf("Queue underflow\n");
        exit(1);
    }
    QueueNode node = q->nodes[q->front];
    if (q->front == q->rear) {
        q->front = q->rear = -1;
    } else {
        q->front++;
    }
    return node;
}

// Function
int binarySearch(Book books[], int n, char *title, int *lowerBound, int *upperBound);

void bfs(int startRow, int startCol, int destRow, int destCol, int rows, int cols, int library[MAX_ROWS][MAX_COLS]);

void printPath(int parent[MAX_ROWS][MAX_COLS][2], int destRow, int destCol);
int loadBooksFromFile(const char *filename, Book books[], int maxBooks);

void suggestBooks(Book books[], int n, char *prefix);

int main() {
    Book books[MAX_BOOKS];
    int numBooks = loadBooksFromFile("books.txt", books, MAX_BOOKS);

    if (numBooks == -1) {
        printf("Error loading books from file.\n");
        return 1;
    }

    // Example library layout (0 = open path, 1 = obstacle)
    int library[MAX_ROWS][MAX_COLS] = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0},
        {0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 1, 0, 0, 0, 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, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
        {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };
```

```
    char prefix[50];
    printf("Enter the book title prefix to search for: ");
    scanf("%s", prefix);

    suggestBooks(books, numBooks, prefix);

    char title[50];
    printf("Enter the exact book title to search for: ");
    scanf(" %[^\n]%*c", title);

    int index = binarySearch(books, numBooks, title, NULL, NULL);
    if (index == -1) {
        printf("Book not found.\n");
        return 0;
    }

    int userRow, userCol;
    printf("Enter your current location (row and column): ");
    scanf("%d %d", &userRow, &userCol);

    printf("Searching for '%s' at (%d, %d)...\n", books[index].title, books[index].row, books[index].col);
    bfs(userRow, userCol, books[index].row, books[index].col, MAX_ROWS, MAX_COLS, library);

    return 0;
}

// Function to load books from a file
int loadBooksFromFile(const char *filename, Book books[], int maxBooks) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("Error opening file.\n");
        return -1;
    }

    int count = 0;
    while (count < maxBooks && fscanf(file, "%[^,],%d,%d\n", books[count].title, &books[count].row,
&books[count].col) == 3) {
        count++;
    }

    fclose(file);
    return count;
}

// Function to perform binary search and find lower and upper bounds
int binarySearch(Book books[], int n, char *title, int *lowerBound, int *upperBound) {
    int left = 0, right = n - 1, mid;
    int cmp;

    // Find the lower bound
```

```
    while (left < right) {
        mid = left + (right - left) / 2;
        cmp = strncmp(books[mid].title, title, strlen(title));
        if (cmp < 0) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    if (strncmp(books[left].title, title, strlen(title)) != 0) {
        return -1;
    }
    if (lowerBound) {
        *lowerBound = left;
    }

    // Find the upper bound
    right = n - 1;
    while (left < right) {
        mid = left + (right - left + 1) / 2;
        cmp = strncmp(books[mid].title, title, strlen(title));
        if (cmp > 0) {
            right = mid - 1;
        } else {
            left = mid;
        }
    }
    if (upperBound) {
        *upperBound = left + 1;
    }

    return left;
}
void suggestBooks(Book books[], int n, char *prefix) {
    int lowerBound, upperBound;
    if (binarySearch(books, n, prefix, &lowerBound, &upperBound) == -1) {
        printf("No suggestions found for the prefix '%s'.\n", prefix);
        return;
    }

    printf("Suggestions for '%s':\n", prefix);
    for (int i = lowerBound; i < upperBound; i++) {
        printf("%s\n", books[i].title);
    }
}
void bfs(int startRow, int startCol, int destRow, int destCol, int rows, int cols, int library[MAX_ROWS][MAX_COLS]) {
    int visited[MAX_ROWS][MAX_COLS] = {0};
    int parent[MAX_ROWS][MAX_COLS][2];
    Queue queue;
    initQueue(&queue);
```

```
    int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

    enqueue(&queue, startRow, startCol);
    visited[startRow][startCol] = 1;
    parent[startRow][startCol][0] = -1;
    parent[startRow][startCol][1] = -1;

    while (!isQueueEmpty(&queue)) {
        QueueNode current = dequeue(&queue);

        if (current.row == destRow && current.col == destCol) {
            printf("Path found:\n");
            printPath(parent, destRow, destCol);
            return;
        }

        for (int i = 0; i < 4; i++) {
            int newRow = current.row + directions[i][0];
            int newCol = current.col + directions[i][1];

            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols && !visited[newRow][newCol]
&& library[newRow][newCol] == 0) {
                enqueue(&queue, newRow, newCol);
                visited[newRow][newCol] = 1;
                parent[newRow][newCol][0] = current.row;
                parent[newRow][newCol][1] = current.col;
            }
        }
    }

    printf("No path found to the destination.\n");
}
void printPath(int parent[MAX_ROWS][MAX_COLS][2], int destRow, int destCol) {
    if (parent[destRow][destCol][0] == -1 && parent[destRow][destCol][1] == -1) {
        printf("(%d, %d)\n", destRow, destCol);
        return;
    }

    printPath(parent, parent[destRow][destCol][0], parent[destRow][destCol][1]);
    printf(" -> (%d, %d)\n", destRow, destCol);
}
```

Time Complexity :

1. Time Complexity: $O(n)$, where nn is the number of books in the file.
2. The binarySearch function is called and has a time complexity of $O(\log n)$. The range of books that match the prefix is then printed, which is a linear operation $O(m)$, where mm is the number of matching books. Overall, the complexity is $O(\log n+m)$.
3. $O(\log n)$, where n is the number of books.
4. $O(V+E)$, where V is the number of vertices (cells in the grid) and E is the number of edges (possible movements). In the worst case, $V=\text{rows}\times\text{cols}V$ and E can be up to $4V$.

The overall complexity depends on the specific operation being performed. For auto-complete suggestions, the most time-consuming part is the binary search and printing the results: $O(\log n+m)$. For pathfinding, it's $O(V+E)$.

Simulation :

Function BFS

void bfs(int startRow, int startCol, int destRow, int destCol, int rows, int cols, int library[MAX_ROWS][MAX_COLS]) { int visited[MAX_ROWS][MAX_COLS] = {0}; int parent[MAX_ROWS][MAX_COLS][2]; Queue queue; initQueue(&queue);

- **Purpose**: This function performs a Breadth-First Search (BFS) to find the shortest path from a start position to a destination position in a grid-based library.
- **Details**:
  - **int visited[MAX_ROWS][MAX_COLS] = {0};**: Initializes a 2D array to keep track of visited nodes, initially all set to 0 (unvisited).
  - **int parent[MAX_ROWS][MAX_COLS][2];**: Initializes a 3D array to store the parent of each node, which will be used to reconstruct the path.
  - **Queue queue;**: Declares a queue to be used for BFS.

- **initQueue(&queue);**: Initializes the queue.

  Direction Array

  int directions[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
  • **Purpose**: Defines the possible moves (up, down, left, right) in the grid. **Enqueue Start Position**
  • **Purpose**: Enqueue the starting position and mark it as visited.

- **enqueue(&queue, startRow, startCol);**: Adds the starting position to the queue.
- **visited[startRow][startCol] = 1;**: Marks the starting position as visited.
- **parent[startRow][startCol][0] = -1;**: Sets the parent of the starting position to -1 (indicating no parent).
- **parent[startRow][startCol][1] = -1;**: Same as above, for column.

  BFS Loop
  • **Purpose**: Continue processing nodes until the queue is empty.

• **while (!isQueueEmpty(&queue))**: Loop until the queue is empty.

• **QueueNode current = dequeue(&queue);**: Dequeues the front node from the queue and assigns it to **current**. **Check Destination**

• **Purpose**: Check if the current node is the destination.
• **if (current.row == destRow && current.col == destCol)**: Checks if the

current node is the destination.

Explore Neighbors
• **Purpose**: Explore all neighboring nodes. • **Details**:

- **for (int i = 0; i < 4; i++)**: Loop through all possible moves.
- **int newRow = current.row + directions[i][0];**: Calculate the new row

  index.

- **int newCol = current.col + directions[i][1];**: Calculate the new column

  index.

  Validate Neighbor

- **Purpose**: Check if the neighbor is within bounds, not visited, and not an obstacle. If valid, enqueue it.
- **Details**:
    - **if (newRow >= 0 && newRow < rows && newCol >= 0 &&**

      newCol < cols): Check if the neighbor is within the grid bounds.

    - **&& !visited[newRow][newCol]**: Check if the neighbor has not been visited.
    - **&& library[newRow][newCol] == 0)**: Check if the neighbor is not an obstacle.

Function Binary Search:

int binarySearch(Book books[], int n, char *title) {

- **int binarySearch**: This declares that the function **binarySearch** returns an integer.
- **Book books[]**: This parameter is an array of **Book** structures. Each **Book** contains a title and its location in the library grid.

Binary Search Loop

while (left <= right) {
• This **while** loop continues as long as the **left** index is less than or equal to the

right index, meaning
there are still elements to consider. **Calculating the Middle Index**

int mid = left + (right - left) / 2;
• **mid** : The **mid** index is calculated by taking the average of **left** and **right**. This is done to prevent potential overflow issues that could occur with **(left + right) / 2**. **Comparing the Mid Element with the Target**
int cmp = **strcmp**(books[mid].title, title);

• **int cmp**: **strcmp** is used to compare the book title at the **mid** index with the target **title**.

Checking the Comparison Result

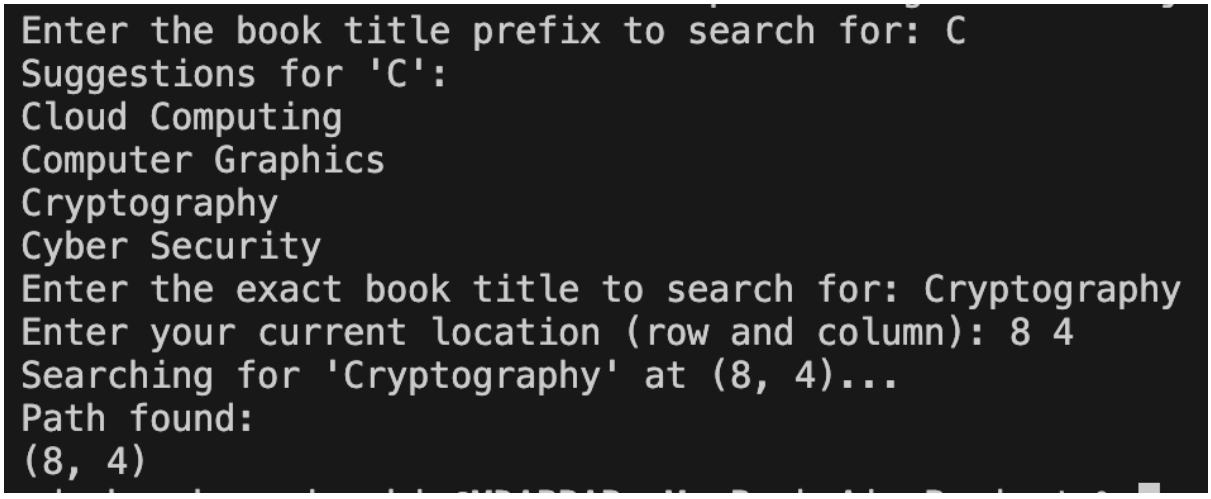
if (cmp == 0) return mid;

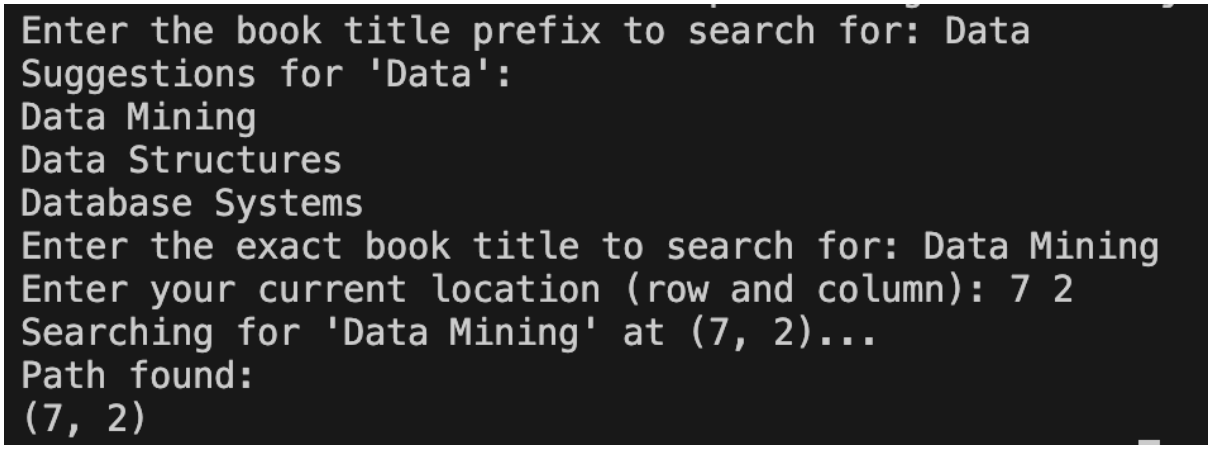
• If **cmp** is **0**, it means the book title at the **mid** index matches the target **title**. The function returns **mid**, the index of the found book. if(cmp<0)left=mid+1;

5. Results The Library Book Search System effectively:

- Suggests book titles based on user input using binary search.

- Finds the shortest path to a book's location using BFS. Users can quickly find books and navigate the library efficiently, reducing the time spent searching for books.





5. **Conclusions** The Library Book Search System combines binary search and BFS to provide a comprehensive solution for locating books in a library. The system enhances user experience by providing auto-complete suggestions and guiding users to the exact location of books, making library navigation more efficient and user-friendly.

Book.txt :

Algorithms,2,3
Artificial Intelligence,5,6
Automata Theory,3,7
Big Data,6,1
Blockchain,4,8
Cloud Computing,5,3
Computer Graphics,3,10
Cryptography,8,4
Cyber Security,2,8
Data Mining,7,2
Data Structures,1,1
Database Systems,6,11
Deep Learning,9,5
Design Patterns,4,12
Distributed Systems,7,8
Ethical Hacking,5,10
Game Development,6,14
Human-Computer Interaction,8,1
Information Retrieval,9,9
Internet of Things,7,12
Machine Learning,5,14
Mobile Computing,10,3
Natural Language Processing,8,8
Neural Networks,6,5
Operating Systems,4,2
Parallel Computing,10,10
Programming Languages,0,0
Quantum Computing,7,14
Robotics,8,11
Software Engineering,3,4
Software Testing,9,2
System Design,10,8
Theoretical Computer Science,11,5
Virtual Reality,12,1
Web Development,11,9
3D Printing,13,4
Agile Methodologies,12,7
Algorithms in C,10,12
Bioinformatics,11,11
Computer Vision,12,13
Cyber-Physical Systems,14,2
Edge Computing,13,10
Embedded Systems,14,5
Smart Cities,12,11