In-Depth Explanation of C Solution for Robot Artifact Collection Problem

Task 5

May 16, 2025

Contents

1 Data Structures

1.1 The `State` Structure

`State` is a structure representing a single robot position, the keys collected, and the number of steps so far.

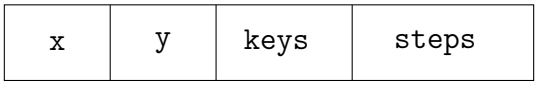
Listing 1: Definition of State

```
typedef struct {
    int x, y, keys, steps;
} State;
```

Field Descriptions:

- `x, y` – Robot's current grid coordinates.
- `keys` – A bitmask representing collected artifacts.
- `steps` – Number of moves taken from the starting position.

Visualization:



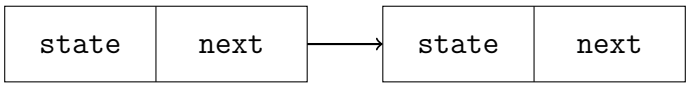
1.2 The `Node` Structure (Queue Element)

Each node in our queue contains a `State` and a pointer to the next node, creating a singly linked list.

Listing 2: Definition of Node

```
typedef struct Node {
    State state;
    struct Node *next;
} Node;
```

This is foundational for implementing a flexible queue to use in Breadth-First Search (BFS). Each node stores all information needed to describe a unique exploration state in the BFS process.



2 Queue Operations: Enqueue and Dequeue

A core part of BFS is maintaining a queue of states to explore. Let's break down the two fundamental queue operations.

2.1 Enqueue Function

The `enqueue` function inserts a new node at the end of the queue.

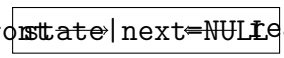
Listing 3: Enqueue Function

```
void enqueue(Node **front, Node **rear, State s) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->state = s;
    node->next = NULL;
    if (*rear) (*rear)->next = node;
    else *front = node;
    *rear = node;
}
```

Explanation:

- `Node **front, Node **rear` are pointers to the queue's front and rear pointers. Using double pointers allows the function to modify the actual queue pointers.
- A new node is allocated and initialized.
- If the queue is empty (`*rear` is NULL), both `front` and `rear` point to the new node.
- Otherwise, the old rear's `next` points to the new node, and `rear` is updated.

Dry Run: Suppose the queue is initially empty (`front = NULL, rear = NULL`). After the first `enqueue`, both point to the new node.



If the queue has nodes, a new node is linked at the end and `rear` is updated.

2.2 Dequeue Function

The `dequeue` function removes a node from the front of the queue and returns its state.

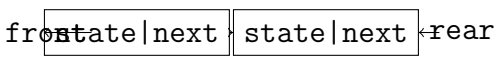
Listing 4: Dequeue Function

```
bool dequeue(Node **front, State *s) {
    if (!*front) return false;
    Node *tmp = *front;
    *s = tmp->state;
    *front = tmp->next;
    free(tmp);
    return true;
}
```

Explanation:

- Returns `false` if the queue is empty.
- Copies the state from the front node.
- Moves the front pointer to the next node.
- Frees the memory of the old front node.

Visualization:



After `dequeue`, front moves to the second node.

3 The `minMoves` Function: BFS with State Tracking

This function implements a breadth-first search (BFS) over the grid to find the minimum moves required to collect all artifacts.

Listing 5: minMoves Function (abridged for space)

```
int minMoves(char **grid, int rows, int cols) {
    // Step 1: Find start and key positions
    int total_keys = 0, start_x = -1, start_y = -1;
    int key_idx[256] = {0};
    int idx = 0;
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++) {
            char c = grid[i][j];
            if (c == '@') { start_x = i; start_y = j; }
            else if (c >= 'a' && c <= 'z') {
                if (!key_idx[(int)c]) key_idx[(int)c] = ++idx;
            }
        }
    total_keys = idx;
    if (total_keys == 0) return 0;
```

Step-by-step Explanation:

1. **Find Start and Artifacts:** The nested loops scan the grid, finding the starting position and assigning each artifact (key) a unique bit index for tracking.

2. **Setup for BFS:**

```
bool visited[MAXN][MAXN][1<<MAX_KEYS] = {0};
Node *front = NULL, *rear = NULL;
State start = {start_x, start_y, 0, 0};
enqueue(&front, &rear, start);
visited[start_x][start_y][0] = 1;
```

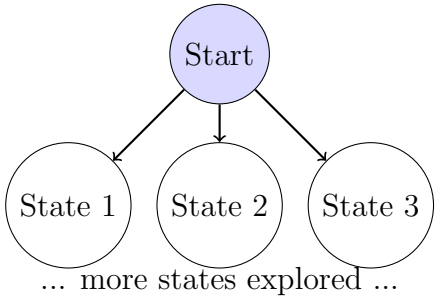
- Initializes a 3D visited array to avoid reprocessing the same state. - Prepares the BFS queue and adds the starting state.

BFS Loop:

```
    while (dequeue(&front, &start)) {
        for (int d = 0; d < 4; d++) {
            int nx = start.x + dx[d], ny = start.y + dy[d];
            int keys = start.keys;
            if (nx < 0 || nx >= rows || ny < 0 || ny >= cols) continue;
            char cell = grid[nx][ny];
            if (cell == '#') continue;
            if (cell >= 'A' && cell <= 'Z') {
                int key_bit = key_idx[(int)(cell-'A'+'a')] - 1;
                if (key_bit < 0 || !(keys & (1<<key_bit))) continue;
            }
            if (cell >= 'a' && cell <= 'z') {
                int key_bit = key_idx[(int)cell] - 1;
                keys |= (1<<key_bit);
                if (keys == (1<<total_keys)-1) return start.steps+1;
            }
            if (!visited[nx][ny][keys]) {
                visited[nx][ny][keys] = 1;
                State next = {nx, ny, keys, start.steps+1};
                enqueue(&front, &rear, next);
            }
        }
    }
    return -1;
}
```

Detailed Explanations:

- Each BFS step explores all four directions from the current cell.
- Checks for walls, boundaries, doors, and artifacts.
- Uses bitmask operations to efficiently track collected artifacts.
- Adds each new reachable, unique state to the queue.
- Returns the number of steps if all artifacts are collected.
- Returns `-1` if not all artifacts can be collected.



BFS State Space Visualization

4 Example Run and Dry-Run Table

Consider the following map:

```
["@..a.",
 "###.#",
 "b.A.B"]
```

- Start at (0,0).
- First collect 'a' and 'b', then use them to pass doors 'A' and 'B'.
- The BFS ensures the minimum number of moves are taken.

Step	Position	Collected Keys	Steps So Far	Notes
0	(0,0)	None	0	Start
1	(0,1)	None	1	Move Right
2	(0,2)	None	2	Move Right
3	(0,3)	'a'	3	Collect 'a'
4	...	...	...	Continue BFS

Table 1: Example BFS dry-run for sample map