864. Shortest Path to Get All Keys II BFS II Bit Manipulation II BFS Visit More than Once

29 June 2023 08:16 AM

G-09 BFS Problem Type-4 II BFS Visit More than Once

[Node can be visited more than Once]

Can visit a cell twice due to RED & BLUE colours.

1129. Shortest Path with Alternating Colors

Medium 3.1K 164

https://leetcode.com/problems/shortest-path-with-alternating-colors/

Can visit a cell K times due to how many obstacles removed/remaining.

1293. Shortest Path in a Grid with Obstacles Elimination

Hard 4.1K 72

https://leetcode.com/problems/shortest-path-in-a-grid-with-obstacles-elimination/

Here also a cell can be visited multiple times due to keys.

864. Shortest Path to Get All Keys

Hard 964 41

https://leetcode.com/problems/shortest-path-to-get-all-keys/

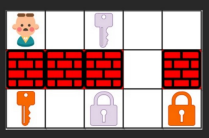
864. Shortest Path to Get All Keys

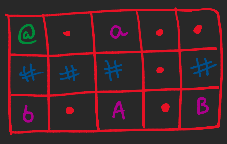
Hard 1.1K 53

Companies

You are given an `m x n` grid `grid` where:

- `'.'` is an empty cell.
- `'#'` is a wall.
- `'@'` is the starting point.
- Lowercase letters represent keys.
- Uppercase letters represent locks.





Moving in a grid

You start at the starting point and one move consists of walking one space in one of the four cardinal directions. You cannot walk outside the grid, or walk into a wall.

If you walk over a key, you can pick it up and you cannot walk over a lock unless you have its corresponding key.

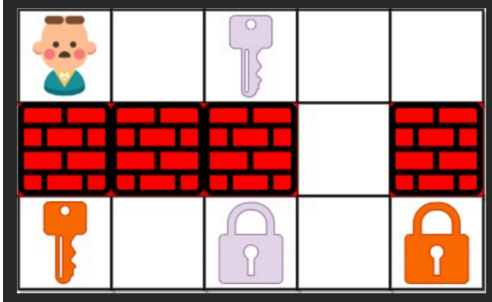
For some `1 <= k <= 6`, there is exactly one lowercase and one uppercase letter of the first `k` letters of the English alphabet in the grid. This means that there is exactly one key for each lock, and one lock for each key; and also that the letters used to represent the keys and locks were chosen in the same order as the English alphabet.

→ At max 6 keys & 6 Locks

Return the lowest number of moves to acquire all keys. If it is impossible, return `-1`.

shortest Path Travelled → In total

Example 1:



→ BFS from source (@)

↓

Task is to collect all the keys

↓

Since we do BFS to all the keys ??

↳ may be we find a lock in between for which Key is Needed !!

Input: grid = ["@.a..","###.#","b.A.B"]
Output: 8
Explanation: Note that the goal is to obtain all the keys not to open all the locks.

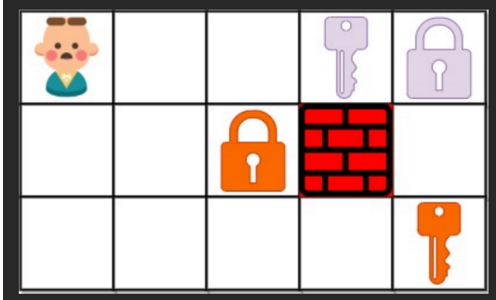
may be we find a lock between for which Key is Needed !!

So, we had to keep track of Keys found so far.

i.e. BFS state {x, y, Keys}

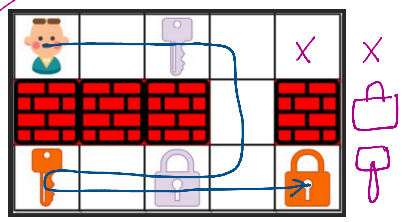
Keys → 6 keys → vector → {a, d}
Bit Mask - {001001} → 9
map → {a}-1 {d}-1

Example 2:

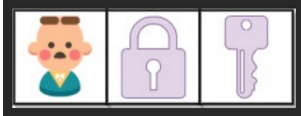


Input: grid = ["@..aA","..B#.","....b"]
Output: 6

Standard - BFS ⇒ Visit every cell exactly once, but here



Example 3:



Input: grid = ["@Aa"]
Output: -1

How to know, when to visit cell again ??

↓

While we get 1 key we could traverse entire Matrix only once for the Lock

↓

As Next Key comes in our traverse ability again renews and now, we can traverse again.

↓

So, earlier visited = {x, y}

Now visited = {keys, x, y} ☺

Constraints:

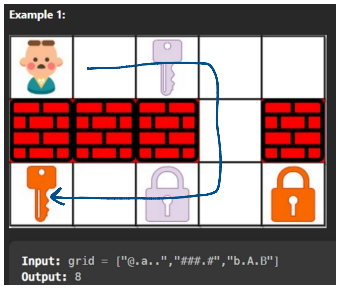
- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 30`
- `grid[i][j]` is either an English letter, `'.'`, `'#'`, or `'@'`.
- The number of keys in the grid is in the range `[1, 6]`.
- Each key in the grid is **unique**.
- Each key in the grid has a matching lock.

Recap :

Starting cell {x, y} → Position with '@'

Max Keys to find = → characters between 'a' to 'f' that are present.

As soon as we find all the keys we stop.




```
int x = -1, y = -1, m = grid.size(), n = grid[0].size(), max_len = -1;
for (int i = 0; i < m; ++i) {
    for (int j = 0; j < n; ++j) {
        char c = grid[i][j];
        if (c == '@') {
            x = i;
            y = j;
        }
        if (c >= 'a' && c <= 'f') {
            max_len = max(c - 'a' + 1, max_len);
        }
    }
}
```

{x,y} starting

⇒ Using BFS → Queue → {x, y} (earlier) → { Keys got so far, x, y }

To traverse the path, keys got so far are required → As they will unlock the Lock :)

Visited → { "keys" + " " + "x" + " " + "y" }

Visited state changed accordingly :)

```
18        vector<int> start = {0, x, y};
19        queue<vector<int>> q;
20        unordered_set<string> visited;
21        visited.insert(to_string(0) + " " + to_string(x) + " " + to_string(y));
22        q.push(start);
```

⇒ Simple BFS → while(!q.empty())

```cpp
while (!q.empty()) {
    int size = q.size();
    for (int k = 0; k < size; ++k) {
        vector<int> curr = q.front();
        q.pop();
        if (curr[0] == (1 << max_len) - 1)
            return step;
        for (auto &d : dirs) {
            int i = curr[1] + d[0];
            int j = curr[2] + d[1];
            int keys = curr[0];
            if (i >= 0 && i < m && j >= 0 && j < n) {
                char c = grid[i][j];
                if (c == '#') continue;
                if (c >= 'a' && c <= 'f')
                    keys |= 1 << (c - 'a');
                if (c >= 'A' && c <= 'F' && ((keys >> (c - 'A')) & 1) == 0) {
                    continue;
                }
                if (!visited.count(to_string(keys) + " " + to_string(i) + " " + to_string(j))) {
                    visited.insert(to_string(keys) + " " + to_string(i) + " " + to_string(j));
                    q.push({keys, i, j});
                }
            }
        }
    }
    ++step;
}
```

curr[0] = keys got so far
curr[1] = current x
curr[2] = current y

keys got so far

```cpp
30                  if (curr[0] == (1 << max_len) - 1)
31                      return step;
```

if we had a, b, c, d

$max\_len = 4 \rightarrow 2^4 = 10000$

$2^4 - 1 = 1111$ (d c b a)

showing, we had

```cpp
for (auto &d : dirs) {
    int i = curr[1] + d[0];
    int j = curr[2] + d[1];
    int keys = curr[0];
    if (i >= 0 && i < m && j >= 0 && j < n) {
        char c = grid[i][j];
        if (c == '#') continue;
        if (c >= 'a' && c <= 'f')
            keys |= 1 << (c - 'a');
```

New i & j

current keys

if valid location

if it's a WALL

if we found a key so add it in our keys

if we had a & c
keys = 0101

if we found d
'd' - 'a' = 3
1 << 3 = 1000

```
        char c = grid[i][j];
        if (c == '#') continue;
        if (c >= 'a' && c <= 'f')
            keys |= 1 << (c - 'a');
        if (c >= 'A' && c <= 'F' && ((keys >> (c - 'A')) & 1) == 0) {
            continue;
        }
        if (!visited.count(to_string(keys) + " " + to_string(i) + " " + to_string(j))) {
            visited.insert(to_string(keys) + " " + to_string(i) + " " + to_string(j));
            q.push({keys, i, j});
        }
    }
}
```

if it a wall

if we find a Key so add it in our Keys

'd' - 'a' = 3

1 << 3 = 1000

Keys | (1<< 3) = 1101

↳ added 'd' as the Key

we have Key for all locks, Now see

As we get a lock ('A' to 'F')

we check if we had a corresponding Key

if already not visited

then visit it

& push in Queue

let's say:

Keys = f e d c b a
1 0 1

lock came as 'C'

'C' - 'A' = 2

1 << 2 = 100

Keys & (1<< 2) = 100

Non. zero

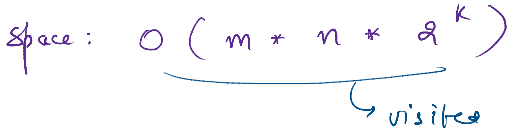
i.e. 'c' as a Key was present

if it would have seen 'zero' then Key not present, so continue & go & find Key first ☺

Time: $O(m * n * 2^K)$

↳ BFS state

Space: $O(m * n * 2^K)$



CODE

```
class Solution {
public:
    int shortestPathAllKeys(vector<string>& grid) {
        int x = -1, y = -1, m = grid.size(), n = grid[0].size(), max_len = -1;
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                char c = grid[i][j];
                if (c == '@') {
                    x = i;
                    y = j;
                }
                if (c >= 'a' && c <= 'f') {
                    max_len = max(c - 'a' + 1, max_len);
                }
            }
        }
        vector<int> start = {0, x, y};
        queue<vector<int>> q;
        unordered_set<string> visited;
        visited.insert(to_string(0) + " " + to_string(x) + " " + to_string(y));
        q.push(start);
        int step = 0;
        vector<vector<int>> dirs {{-1, 0}, {1, 0}, {0, 1}, {0, -1}};
        while (!q.empty()) {
            int size = q.size();
            for (int k = 0; k < size; ++k) {
                vector<int> curr = q.front();
                q.pop();
                if (curr[0] == (1 << max_len) - 1)
                    return step;
                for (auto &d : dirs) {
                    int i = curr[1] + d[0];
                    int j = curr[2] + d[1];
                    int keys = curr[0];
                    if (i >= 0 && i < m && j >= 0 && j < n) {
                        char c = grid[i][j];
                        if (c == '#') continue;
                        if (c >= 'a' && c <= 'f')
                            keys |= 1 << (c - 'a');
                        if (c >= 'A' && c <= 'F' && ((keys >> (c - 'A')) & 1) == 0) {
                            continue;
                        }
                        if (!visited.count(to_string(keys) + " " + to_string(i) + " " + to_string(j))) {
                            visited.insert(to_string(keys) + " " + to_string(i) + " " + to_string(j));
                            q.push({keys, i, j});
                        }
                    }
                }
            }
            ++step;
        }
        return -1;
    }
};
```