1036. Escape a Large Maze **Hash Table Depth-First Search Breadth-First Search** Array Hard **Leetcode Link**

Problem Description

to a target cell. The grid size mentioned is immense (1 million by 1 million), but the actual navigation is obstructed by a number of blocked cells, which cannot be passed through. The objective is to figure out whether it is possible to start from the source cell and reach the target cell by moving one cell at a time in any of the four cardinal directions (north, east, south, or west), without stepping on a blocked cell or going out of the grid's bounds. The specific task at hand is to confirm the reachability of the target cell from the source cell under these conditions. Intuition

The problem presents a situation where we have an infinitely large grid, and we want to determine if we can travel from a source cell

To solve this problem efficiently, considering the vast size of the grid, an exhaustive search approach (like breadth-first search or depth-first search through all possible paths) is not feasible for every possible movement. Instead, the solution uses a smarter

The insight behind the solution is that if there are enough free cells around the source or target then it is highly probable that a path exists. If a blockage exists, it will likely manifest within a relatively small vicinity of the start or end point. Therefore, we don't have to search the entire million by million grid; we only need to explore a reasonably sized area around the source and target.

In the provided solution code, a depth-first search (DFS) algorithm is applied, starting from the source and attempting to reach the target. The algorithm also starts again from the target and attempts to reach the source. This bidirectional approach can quickly determine whether a connection exists between these points because paths found from both sides are highly likely to intersect, given the vast size of the grid.

The DFS function tries to move from the current cell in all four possible directions and checks if any of the new cells either: · Go beyond the grid limits, Are part of the set of blocked cells, or Have been seen already in the current path.

If none of these conditions is met, the DFS continues to the next cell. To prevent the DFS from running indefinitely, a heuristic is used: if we have visited more than a certain number of cells (20,000 in this case), we assume that it is possible to escape, given the

- low probability of still being trapped after traversing through so many cells. This heuristic is based on the theory that if there were an enclosing blockade, it would have been encountered within a much smaller area.

strategy involving a depth-first search (DFS).

Solution Approach

cell is blocked or has already been visited.

while avoiding blocked cells and previously visited cells.

ensuring we do not step outside the 1 million by 1 million grid.

means the target has been reached, and True is returned.

without having to traverse every possible path in the massive grid space.

1. Initialization: The set of blocked cells contains (1,1), (2,1), and (3,1).

grid, and (1,0) is a valid, unblocked, and unseen cell, the DFS continues to (1,0).

The solution approach makes use of a depth-first search (DFS) strategy. DFS is a common algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

• Sets: Both the blocked cells and the cells that have been seen during the DFS traversal are stored in sets. This is because set operations such as checking for membership ((x,y) in blocked) are very efficient, allowing the solution to quickly determine if a

Implementation Details:

Data Structures Used:

Functions: • dfs Function: This function is a recursive function used for DFS traversal. It takes a source (current cell), target cell, and a seen set (cells already visited) as inputs. It tries to reach the target by recursively exploring north, east, south, and west directions,

2. Boundary Conditions: In the recursive dfs function, the first condition checks whether the current cell is within the grid bounds,

3. Blocked and Seen Checks: The function then checks if the current cell is blocked or has already been seen (visited). If either

1. Initialization: The set of blocked cells is constructed at the beginning to have easy, fast access to whether a cell is blocked.

- condition is true, the function returns False, as the path is not valid. 4. Termination Heuristic: The seen set is used to track the number of unique cells visited. If the size of the seen set exceeds
- iterating through the different directions with offsets [[0, -1], [0, 1], [1, 0], [-1, 0]]. It then calls itself recursively (dfs) with each new coordinate as the source.

6. Bidirectional Search: The primary function is Escape Possible calls the DFS function twice, first with source to target, and then

with target to source. Both calls must return True for the overall function to return True.

cells at coordinates (1,1), (2,1), and (3,1). Our aim is to find a path from the source to the target.

5. Recursive Exploration: When the aforementioned conditions fail, the function proceeds to generate the next possible moves by

20,000, the function assumes it is possible to reach the target, as explained in the intuition part. If the current cell is the target, it

areas. The 20,000-step heuristic ensures that even in the absence of a clear path, we avoid lengthy and unnecessary traversals, relying on probability to assert the likely existence of a path in such a vastly sized grid. The combination of these techniques allows the solution to efficiently predict the possibility of reaching the target from the source

By using DFS and bidirectional search, the problem is solved effectively through localized exploration around the source and target

Example Walkthrough Imagine a grid that is a 5×5 section extracted from the infinitely large grid mentioned in the problem description. In this grid, let's

consider the source cell to be at coordinates (0,0) and the target cell to be at coordinates (4,4). Now, let's place some blocked

2. First DFS Call (Source to Target): We call the dfs function with the source at (0,0), the target at (4,4), and an initially empty seen set.

b. The function then tries to explore adjacent cells (0,-1), (0,1), (1,0), and (-1,0). Since (0,-1) and (-1,0) are outside the

a. The function checks that (0,0) is within the grid, not blocked, and not seen before, so it adds (0,0) to the seen set.

d. This process continues following a pattern that avoids blocked and previously seen cells, aiming to find the target.

c. At (1,0), the process repeats, exploring adjacent cells. Since (1,1) is blocked, DFS will try (2,0).

Step-by-Step Breakdown:

f. If the target cell (4,4) is reached, the function returns True. 3. Second DFS Call (Target to Source): Assuming the first DFS call returned True, a second DFS is initiated from target to source,

heuristic and bidirectional search to confirm reachability in such a vast space.

if len(seen) > 20000 or current == target:

if dfs(next_position, target, seen):

for delta_x, delta_y in [[0, -1], [0, 1], [1, 0], [-1, 0]]:

next_position = [x + delta_x, y + delta_y]

// Directions representing the possible moves (up, down, left, right)

// A large number used for hashing 2D coordinates into a single number

private static final int BASE = (int) 1e6;

private Set<Integer> blockedSet;

blockedSet = new HashSet<>();

for (int[] block : blocked) {

// Mark the current cell as visited

// Explore in all four directions

for (int[] direction : DIRECTIONS) {

int newX = x + direction[0];

int newY = y + direction[1];

// Continue the search from the new cell

visited.add(x * BASE + y);

return true;

// Set to store the blocked cells as hashed integers

// Initialize the hash set of blocked coordinates

blockedSet.add(block[0] * BASE + block[1]);

// Helper method to perform DFS and check if a path exists

// Coordinate decompression from the current cell

private static final int[][] DIRECTIONS = $\{\{1, 0\}, \{-1, 0\}, \{0, 1\}, \{0, -1\}\}$;

public boolean isEscapePossible(int[][] blocked, int[] source, int[] target) {

// Check if both source to target and target to source are reachable

if (visited.size() > $20000 \mid \mid (x == target[0] \&\& y == target[1])) {$

if (isReachable(new int[] {newX, newY}, target, visited)) {

private boolean isReachable(int[] current, int[] target, Set<Integer> visited) {

// If we have checked a certain number of steps or reached the target, return true

return isReachable(source, target, new HashSet<>()) && isReachable(target, source, new HashSet<>());

Explore all four neighboring positions

return True

4. Outcome: For our example, the DFS would successfully find paths that bypass the blocked cells and reach the target, for both source to target and target to source, resulting in the overall function returning True.

This example demonstrates how the DFS function operates around obstacles and why we don't need to explore every cell, relying on

class Solution: def isEscapePossible(self, blocked: List[List[int]], source: List[int], target: List[int]) -> bool: # Helper function to perform depth-first search def dfs(current, target, seen): # Unpack the current coordinates

21 return True 22 return False 23 24 # Convert the list of blocked positions into a set for faster lookup

```
e. If at any point the size of the seen set exceeds 20,000 (not possible in this small grid, but applicable in the actual problem),
the function will return True.
```

following similar steps.

Python Solution

12

13

14

15

16

17

18

19

20

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

61

60 };

return true;

return false;

Typescript Solution

// Explore in all 4 directions.

for (auto& dir : directions) {

return true;

if (dfs(next, target, visited)) {

1 type Point = [number, number]; // Represents a point in the grid.

const GRID_SIZE: number = 1e6; // The size of the grid.

vector<int> next = {currentX + dir[0], currentY + dir[1]};

let blockedSet: Set<number> = new Set(); // Set to keep track of blocked cells.

x, y = current# Check if the current position is out of bounds, blocked or already seen if not $(0 \le x \le 10**6)$ and $0 \le y \le 10**6)$ or (x, y) in blocked or (x, y) in seen: 8 return False 9 # Mark the current position as seen 10 seen.add((x, y)) 11

encloses an area of roughly 20000 squares (theoretical upper bound given by the problem constraints).

If the number of seen positions is large enough or the target is reached, return True.

The choice of 20000 is because in the worst case, the blocked area can form a perimeter which

25 blocked = set(map(tuple, blocked)) 26 # Perform dfs from both the source to the target, and the target to the source 27 # to check if both paths are unblocked. 28 return dfs(source, target, set()) and dfs(target, source, set()) 29

26 int x = current[0], y = current[1]; 27 28 // Check if current is out of bounds, is blocked, or has been visited already if $(x < 0 \mid | x >= BASE \mid | y < 0 \mid | y >= BASE \mid | blockedSet.contains(x * BASE + y) | visited.contains(x * BASE + y)) {$ 29 30 return false; 31

Java Solution

1 import java.util.HashSet;

public class Solution {

2 import java.util.Set;

```
return true;
 48
 49
 50
             return false;
 51
 52
 53
 54
C++ Solution
   #include <vector>
  2 #include <unordered set>
    using namespace std;
     typedef unsigned long long ULL;
   class Solution {
    public:
         vector<vector<int>> directions = {\{0, 1\}, \{0, -1\}, \{1, 0\}, \{-1, 0\}\}};
         unordered_set<ULL> blockedSet;
 10
         const int GRID SIZE = 1e6;
 11
 12
 13
         // Checks if it's possible to escape from source to target given a list of blocked cells.
         bool isEscapePossible(vector<vector<int>>& blockedCells, vector<int>& source, vector<int>& target) {
 14
 15
             // Reset the hashed set of blocked cells for a new escape query.
             blockedSet.clear();
 16
 17
             // Hashing each block cell position into a single number and storing it in blockedSet.
             for (auto& cell : blockedCells) {
 18
                 blockedSet.insert(static_cast<ULL>(cell[0]) * GRID_SIZE + cell[1]);
 19
 20
 21
 22
             // Create two sets to keep track of visited cells for both directions.
 23
             unordered_set<ULL> visitedFromSource;
 24
             unordered_set<ULL> visitedFromTarget;
 25
 26
             // Run bidirectional DFS to see if both source and target can reach each other.
             return dfs(source, target, visitedFromSource) && dfs(target, source, visitedFromTarget);
 27
 28
 29
 30
         // Depth First Search to find a path from source to target, avoiding blocked cells.
 31
         bool dfs(vector<int>& source, vector<int>& target, unordered_set<ULL>& visited) {
             int currentX = source[0], currentY = source[1];
 32
 33
             int targetX = target[0], targetY = target[1];
 34
 35
             // Check if out of bounds, at a blocked cell, or the cell has already been visited.
 36
             if (currentX < 0 || currentX >= GRID_SIZE || currentY < 0 || currentY >= GRID_SIZE ||
                 targetX < 0 || targetX >= GRID_SIZE || targetY < 0 || targetY >= GRID_SIZE ||
 37
 38
                 blockedSet.count(static_cast<ULL>(currentX) * GRID_SIZE + currentY) ||
 39
                 visited.count(static_cast<ULL>(currentX) * GRID_SIZE + currentY)) {
 40
                 return false;
 41
 42
 43
             // Mark the current cell as visited.
 44
             visited.insert(static_cast<ULL>(currentX) * GRID_SIZE + currentY);
 45
             // If we have visited enough cells or reached the target, consider it possible to escape.
 46
 47
             if (visited.size() > 20000 || (currentX == targetX && currentY == targetY)) {
```

23 24 25 } 26

```
const directions: Point[] = [[0, 1], [0, -1], [1, 0], [-1, 0]]; // Possible directions of movement.
   // Hash a 2D grid position to a unique number.
   function hashPosition(x: number, y: number): number {
        return x * GRID_SIZE + y;
10 }
11
12 // Checks if it's possible to escape from source to target given a list of blocked cells.
   function isEscapePossible(blockedCells: Point[], source: Point, target: Point): boolean {
        blockedSet.clear(); // Reset the set for new escape query.
14
15
       // Fill up the set with the hashed positions of blocked cells.
16
       blockedCells.forEach(cell => {
17
            blockedSet.add(hashPosition(cell[0], cell[1]));
       });
18
19
        let visitedFromSource: Set<number> = new Set();
20
        let visitedFromTarget: Set<number> = new Set();
21
22
       // Run bidirectional DFS to see if both source and target can reach each other.
        return dfs(source, target, visitedFromSource) && dfs(target, source, visitedFromTarget);
   // Depth First Search to find a path from source to target, avoiding blocked cells.
   function dfs(source: Point, target: Point, visited: Set<number>): boolean {
29
        const [currentX, currentY] = source;
        const [targetX, targetY] = target;
30
31
32
       // Check if current position is out of bounds, blocked, or already visited.
       if (currentX < 0 || currentX >= GRID_SIZE || currentY < 0 || currentY >= GRID_SIZE ||
34
            blockedSet.has(hashPosition(currentX, currentY)) ||
35
            visited.has(hashPosition(currentX, currentY))) {
36
            return false;
37
38
39
       // Mark the current position as visited.
40
        visited.add(hashPosition(currentX, currentY));
41
42
       // If the number of visited cells is over a threshold or we've reached the target, escape is possible.
       if (visited.size > 20000 || (currentX === targetX && currentY === targetY)) {
43
44
            return true;
45
46
47
       // Try to move in each direction to find a path.
48
        for (let dir of directions) {
           const nextX = currentX + dir[0];
49
           const nextY = currentY + dir[1];
50
           if (dfs([nextX, nextY], target, visited)) {
51
52
               return true;
53
54
55
56
        return false;
57 }
58
```

takes O(B) time, where B is the size of the blocked list.

Time and Space Complexity The time complexity of the provided code is O(B + (D^2)), where B is the number of blocked cells and D is the distance that needs to be traversed in the worst case. This is because the dfs function is called twice — once from the source to the target, and once from the target to the source. Each call to dfs can potentially explore up to 20,000 cells due to the check if len(seen) > 20000. This check works as an early termination condition; without this condition, the algorithm would have had a time complexity of O((10^6)^2), which is the total number of cells. However, since blocked is a set, the check (x, y) in blocked can be performed in O(1) time on average, leading to the time complexity of the exploration to be O(D^2). The initial transformation of the blocked list into a set also

D is the maximum depth of the dfs recursion stack. The effective limit of the recursion stack is based on the early termination condition when len(seen) exceeds 20,000. The seen set also contributes to the space complexity, which in the worst case will hold the same number of elements as the depth of dfs. So, the space needed is dependent on the number of blocked cells and the recursive depth that results from the early stopping of the search once 20,000 cells have been seen.

The space complexity of the code is O(B + D), where B is the number of elements in the blocked list that is converted into a set and