1036. Escape a Large Maze

Hard | Depth-First Search | Breadth-First Search | Array | Hash Table | Leetcode Link

Problem Description

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 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

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 strategy involving a depth-first search (DFS).

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.

Solution Approach

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.

Data Structures Used:

- **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 cell is blocked or has already been visited.

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, while avoiding blocked cells and previously visited cells.

Implementation Details:

1. **Initialization**: The set of blocked cells is constructed at the beginning to have easy, fast access to whether a cell is blocked.
2. **Boundary Conditions**: In the recursive `dfs` function, the first condition checks whether the current cell is within the grid bounds, ensuring we do not step outside the 1 million by 1 million grid.
3. **Blocked and Seen Checks**: The function then checks if the current cell is blocked or has already been seen (visited). If either 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 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 means the target has been reached, and `True` is returned.
5. **Recursive Exploration**: When the aforementioned conditions fail, the function proceeds to generate the next possible moves by 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 `isEscapePossible` 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`.

By using DFS and bidirectional search, the problem is solved effectively through localized exploration around the source and target 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 without having to traverse every possible path in the massive grid space.

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

Step-by-Step Breakdown:

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

   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.

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

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

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

   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`.

   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`, following similar steps.
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 heuristic and bidirectional search to confirm reachability in such a vast space.

Python Solution

```python
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
            x, y = current
            # Check if the current position is out of bounds, blocked or already seen
            if not (0 <= x < 10**6 and 0 <= y < 10**6) or (x, y) in blocked or (x, y) in seen:
                return False
            # Mark the current position as seen
            seen.add((x, y))
            # 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
            # encloses an area of roughly 20000 squares (theoretical upper bound given by the problem constraints).
            if len(seen) > 20000 or current == target:
                return True
            # Explore all four neighboring positions
            for delta_x, delta_y in [[0, -1], [0, 1], [1, 0], [-1, 0]]:
                next_position = [x + delta_x, y + delta_y]
                if dfs(next_position, target, seen):
                    return True
            return False

        # Convert the list of blocked positions into a set for faster lookup
        blocked = set(map(tuple, blocked))
        # Perform dfs from both the source to the target, and the target to the source
        # to check if both paths are unblocked.
        return dfs(source, target, set()) and dfs(target, source, set())

```

Java Solution

```java
import java.util.HashSet;
import java.util.Set;

public class Solution {
    // Directions representing the possible moves (up, down, left, right)
    private static final int[][] DIRECTIONS = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    // A large number used for hashing 2D coordinates into a single number
    private static final int BASE = (int) 1e6;
    // Set to store the blocked cells as hashed integers
    private Set<Integer> blockedSet;

    public boolean isEscapePossible(int[][] blocked, int[] source, int[] target) {
        // Initialize the hash set of blocked coordinates
        blockedSet = new HashSet<>();
        for (int[] block : blocked) {
            blockedSet.add(block[0] * BASE + block[1]);
        }

        // Check if both source to target and target to source are reachable
        return isReachable(source, target, new HashSet<>()) && isReachable(target, source, new HashSet<>());
    }

    // Helper method to perform DFS and check if a path exists
    private boolean isReachable(int[] current, int[] target, Set<Integer> visited) {
        // Coordinate decompression from the current cell
        int x = current[0], y = current[1];

        // Check if current is out of bounds, is blocked, or has been visited already
        if (x < 0 || x >= BASE || y < 0 || y >= BASE || blockedSet.contains(x * BASE + y) || visited.contains(x * BASE + y)) {
            return false;
        }

        // Mark the current cell as visited
        visited.add(x * BASE + y);

        // If we have checked a certain number of steps or reached the target, return true
        if (visited.size() > 20000 || (x == target[0] && y == target[1])) {
            return true;
        }

        // 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
            if (isReachable(new int[] {newX, newY}, target, visited)) {
                return true;
            }
        }

        return false;
    }
}

```

C++ Solution

```cpp
#include <vector>
#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;
    const int GRID_SIZE = 1e6;

    // 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) {
        // Reset the hashed set of blocked cells for a new escape query.
        blockedSet.clear();
        // Hashing each block cell position into a single number and storing it in blockedSet.
        for (auto& cell : blockedCells) {
            blockedSet.insert(static_cast<ULL>(cell[0]) * GRID_SIZE + cell[1]);
        }

        // Create two sets to keep track of visited cells for both directions.
        unordered_set<ULL> visitedFromSource;
        unordered_set<ULL> visitedFromTarget;

        // 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.
    bool dfs(vector<int>& source, vector<int>& target, unordered_set<ULL>& visited) {
        int currentX = source[0], currentY = source[1];
        int targetX = target[0], targetY = target[1];

        // Check if out of bounds, at a blocked cell, or the cell has already been visited.
        if (currentX < 0 || currentX >= GRID_SIZE || currentY < 0 || currentY >= GRID_SIZE ||
            targetX < 0 || targetX >= GRID_SIZE || targetY < 0 || targetY >= GRID_SIZE ||
            blockedSet.count(static_cast<ULL>(currentX) * GRID_SIZE + currentY) ||
            visited.count(static_cast<ULL>(currentX) * GRID_SIZE + currentY)) {
            return false;
        }

        // Mark the current cell as visited.
        visited.insert(static_cast<ULL>(currentX) * GRID_SIZE + currentY);

        // If we have visited enough cells or reached the target, consider it possible to escape.
        if (visited.size() > 20000 || (currentX == targetX && currentY == targetY)) {
            return true;
        }

        // Explore in all 4 directions.
        for (auto& dir : directions) {
            vector<int> next = {currentX + dir[0], currentY + dir[1]};
            if (dfs(next, target, visited)) {
                return true;
            }
        }
        return false;
    }
};

```

Typescript Solution

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

const GRID_SIZE: number = 1e6; // The size of the grid.
let blockedSet: Set<number> = new Set(); // Set to keep track of blocked cells.
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;
}

// 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.
    // Fill up the set with the hashed positions of blocked cells.
    blockedCells.forEach(cell => {
        blockedSet.add(hashPosition(cell[0], cell[1]));
    });

    let visitedFromSource: Set<number> = new Set();
    let visitedFromTarget: Set<number> = new Set();

    // 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 {
    const [currentX, currentY] = source;
    const [targetX, targetY] = target;

    // Check if current position is out of bounds, blocked, or already visited.
    if (currentX < 0 || currentX >= GRID_SIZE || currentY < 0 || currentY >= GRID_SIZE ||
         blockedSet.has(hashPosition(currentX, currentY)) ||
         visited.has(hashPosition(currentX, currentY))) {
        return false;
    }

    // Mark the current position as visited.
    visited.add(hashPosition(currentX, currentY));

    // 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)) {
        return true;
    }

    // Try to move in each direction to find a path.
    for (let dir of directions) {
        const nextX = currentX + dir[0];
        const nextY = currentY + dir[1];
        if (dfs([nextX, nextY], target, visited)) {
            return true;
        }
    }

    return false;
}

```

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

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 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.