2849. Determine if a Cell Is Reachable at a Given Time

Medium Math

Leetcode Link

Problem Description

In this problem, you are placed on an infinite 2D grid at coordinates `(sx, sy)`. Your task is to determine whether you can reach a specific cell `(fx, fy)` in exactly `t` seconds. The movement is restricted to each second and you must move to any of the 8 cells adjacent to your current cell. Those cells can be directly to the north, northeast, east, southeast, south, southwest, west, or northwest of the cell you're currently in. You can revisit cells multiple times.

Intuition

The intuitive approach to this problem is to consider the minimum number of moves required to go from `(sx, sy)` to `(fx, fy)`. The number of moves is simply the maximum of the horizontal and vertical distances between the start and end points (`dx` and `dy`). Since we can move to any of the 8 adjacent cells each second, we can increment either or both the x and y positions by 1 each second, heading directly toward the destination in a diagonal line until we line up horizontally or vertically, then moving straight towards the destination.

Now, if these minimum required moves exceed `t`, it's impossible to reach the destination in time. However, if `t` is exactly enough for these moves or more, it becomes possible. There's a catch, though: we cannot reach the exact cell in one move since our first move will always place us in one of the 8 adjacent cells around `(sx, sy)` and never exactly at `(sx, sy)` again, so we must check if `t` is not equal to 1 when the start and end points are the same. Taking all this into account, the solution checks if the maximum of `dx` and `dy` is less than or equal to `t` and handles the special case where the start and end points are the same.

Solution Approach

The implementation of the provided solution approach is straightforward and does not involve complex algorithms or data structures. It relies on absolute differences and comparison operations. Here's a breakdown of the steps followed:

1. The code begins by checking if the starting cell `(sx, sy)` is the same as the final cell `(fx, fy)`. If they are the same and `t` is not 1, it returns `True` as you can simply stay in place for `t` seconds. If `t` is 1, you cannot stay in the same cell since you must move every second, hence it returns `False`.

2. If the starting cell and the final cell are different, the Manhattan distance is not directly used here. Instead, the code calculates the horizontal distance `dx` by finding the absolute difference between `sx` and `fx` and the vertical distance `dy` by finding the absolute difference between `sy` and `fy`.

3. Then, since you could move in a diagonal direction (in which both the `x` and `y` coordinates change), the code finds the maximum of `dx` and `dy`. This is because moving diagonally will reduce both `dx` and `dy` until you align horizontally or vertically with the target cell, at which point you will only need to move in one direction to reach the cell.

4. To reach the destination in exactly `t` seconds, the maximum required steps (either `dx` or `dy`) must be less than or equal to `t`. This ensures that there is the possibility of exactly matching the distance in the given time or having extra time to move around but still end up on the target cell by the `t`-th second.

The provided code snippet puts this approach into practice with an `if` condition and a couple of absolute value calculations. No additional patterns, algorithms, or data structures are needed since the problem is concerned only with calculating distances and not with the specific path taken.

Example Walkthrough

Let's walk through an example to illustrate the solution approach described.

Suppose we start at `sx = 2, sy = 3` and want to reach `fx = 5, fy = 7` in exactly `t = 5` seconds.

1. We check if the starting cell `(sx, sy)` is the same as the final cell `(fx, fy)`. Here, they are not the same, so we will need to move.

2. Next, we calculate the absolute differences for both the horizontal and vertical distances. The horizontal distance `dx` is `|5 - 2| = 3`, and the vertical distance `dy` is `|7 - 3| = 4`.

3. The maximum of `dx` and `dy` is used to find the minimum number of seconds required to reach the target cell. Between `dx=3` and `dy=4`, the maximum is `4`. This is the minimum number of seconds we need to reach the target cell, moving diagonally whenever possible.

4. We check if we can reach the destination in exactly `t` seconds. Since `t = 5`, which is greater than the maximum of `dx` and `dy`, it is possible to reach `fx, fy` in exactly `t` seconds. We have one extra second that we can use to move to an adjacent cell and come back, or simply move in a non-optimal path that ensures we end up on `(fx, fy)` after 5 seconds.

Based on these steps, the solution would return `True`, indicating that yes, we can indeed reach cell `(5, 7)` from `(2, 3)` in exactly `5` seconds.

Python Solution

```python
class Solution:
    def isReachableAtTime(self, start_x: int, start_y: int, finish_x: int, finish_y: int, time: int) -> bool:
        # If the start and finish coordinates are the same
        if start_x == finish_x and start_y == finish_y:
            # If the time is not equal to 1 then it is reachable at the given time
            return time != 1

        # Calculate the absolute differences in x and y coordinates
        delta_x = abs(start_x - finish_x)
        delta_y = abs(start_y - finish_y)

        # Check whether the larger of the x and y differences is within the available time
        # This determines if the target can be reached at the given time
        return max(delta_x, delta_y) <= time


```

Java Solution

```java
class Solution {

    /**
     * Determines if it's possible to reach from the starting point to the final
     * point in a given time t.
     *
     * @param startX The starting x-coordinate.
     * @param startY The starting y-coordinate.
     * @param finalX The final x-coordinate.
     * @param finalY The final y-coordinate.
     * @param time The time constraint within which the point must be reached.
     * @return A boolean indicating whether it is possible to reach the final point in time or not.
     */
    public boolean isReachableAtTime(int startX, int startY, int finalX, int finalY, int time) {
        // Check if the starting and final points are the same and ensure time is not 1
        if (startX == finalX && startY == finalY) {
            return time != 1;
        }

        // Calculate the distance from the starting point to the final point for both x and y coordinates
        int distanceX = Math.abs(startX - finalX);
        int distanceY = Math.abs(startY - finalY);

        // Return true if the maximum distance on either the x-axis or y-axis
        // is less than or equal to allowed time
        return Math.max(distanceX, distanceY) <= time;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Check if it's possible to reach from (startX, startY) to (finishX, finishY) in 't' time steps.
    bool isReachableAtTime(int startX, int startY, int finishX, int finishY, int time) {
        // If starting and finishing positions are the same, we can't reach in exactly one step.
        if (startX == finishX && startY == finishY) {
            return time != 1;
        }

        // Calculate the distance needed to travel in x and y direction.
        int deltaX = abs(finishX - startX);
        int deltaY = abs(finishY - startY);

        // Check if the maximum number of steps required in either x or y direction is less than or equal to the available time steps
        return max(deltaX, deltaY) <= time;
    }
};

```

Typescript Solution

```typescript
// Function to determine if it is possible to reach the final destination (fx, fy)
// from the starting position (sx, sy) within the time 't'.
function isReachableAtTime(startX: number, startY: number, finalX: number, finalY: number, time: number): boolean {
    // Check if the start and final positions are the same
    if (startX === finalX && startY === finalY) {
        // If positions are the same and time is not 1, we can reach the destination
        return time !== 1;
    }

    // Calculate the absolute differences in x and y coordinates to get the distance
    const distanceX = Math.abs(startX - finalX);
    const distanceY = Math.abs(startY - finalY);

    // Return true if the maximum of the x or y distance is less than or equal to the available time
    return Math.max(distanceX, distanceY) <= time;
}

```

Time and Space Complexity

The time complexity of the code is `O(1)`, as all operations performed in the `isReachableAtTime` function are constant time operations. There are no loops or recursive calls that depend on the size of the input. The function performs arithmetic operations and comparisons, which take constant time regardless of the values of `sx`, `sy`, `fx`, `fy`, and `t`.

The space complexity of the code is also `O(1)` since the amount of memory used does not scale with the input size. The function only uses a fixed amount of additional memory for variables `dx` and `dy`, which does not change with the size of the input values.