789. Escape The Ghosts

Medium Array Math

Problem Description

In this game, you're playing a version of PAC-MAN where you're trying to reach a specific target point on an infinite 2-D grid, while avoiding being caught by ghosts. You start at `[0, 0]` and your target location is given by `[x_target, y_target]`. Ghosts are placed on the grid, and their starting positions are given in a 2D array `ghosts`, where each entry `ghosts[i]` represents the starting position `[xi, yi]` of the `i-th` ghost.

On every turn, both you and the ghosts can choose one of the following actions: move one unit north, south, east, or west, or stay still. The key is that all moves happen at the same time. You can only safely reach your target if you can get there before any ghost can reach you. If you reach the target or any square at the same time as a ghost, you do not escape and you lose the game.

Your task is to determine if it's possible for you to reach the target before any of the ghosts can get to you, under any circumstances. If you can always reach the target no matter how the ghosts move, you should return `true`, otherwise, you should return `false`.

Intuition

The intuition behind the solution is based on comparing distances. Since the grid is infinite and you and the ghosts move simultaneously, the only way a ghost can catch you is if it is closer to the target than you are, or if it gets to your path before you reach your target. If you are closer to the target than any of the ghosts, then you can always reach the target first by moving directly towards it.

The solution leverages the Manhattan distance, which in a grid layout is the sum of the horizontal and vertical distances. Our PAC-MAN character can reach the target if the Manhattan distance from `[0, 0]` to the `target` is less than the Manhattan distance from any ghost's starting position to the `target`. This is because if our character is closer, no matter how the ghosts move, they cannot reach the target before our character does. The code therefore calculates the Manhattan distance from the start to the target, which is `abs(tx) + abs(ty)`, and compares it with the Manhattan distance from each ghost to the target `abs(tx - x) + abs(ty - y)`.

If for every ghost, the ghost's distance to the target is greater than your distance to the target, then you can escape, and the function returns `true`; otherwise, at least one ghost can reach the target quicker or at the same time as you, and you cannot escape, meaning the function should return `false`.

The implementation of the solution uses a simple loop to iterate through each ghost's position in the `ghosts` array. It then applies the Manhattan distance formula to compare the distance of each ghost from the target with your distance from the target.

Algorithms & Formulas

No advanced algorithms are used in this solution. The code hinges on the understanding and application of the Manhattan distance in a grid. The Manhattan distance D between two points `(x1, y1)` and `(x2, y2)` is calculated as:

```
D = |x2 - x1| + |y2 - y1|
```

where $|.|$ denotes the absolute value. This distance calculation assumes that you can only move along grid lines (no diagonal movement).

Data Structures

The provided solution does not utilize any complex data structures; it simply iterates over the list of ghosts' locations.

Patterns

The key pattern here is a direct comparison within a loop that terminates early if any ghost is too close to the target. The Python built-in function `all()` is used to compact the pattern into a single expression.

Code Walkthrough

- `tx, ty = target` extracts the target coordinates.
- The expression `abs(tx - x) + abs(ty - y)` computes the Manhattan distance from each ghost's starting position to the target.
- `abs(tx) + abs(ty)` gives your Manhattan distance to the target from the starting point `[0, 0]`.
- The `all()` function is used to check if, for all ghosts, the ghost's Manhattan distance to the target is greater than your Manhattan distance to the target.
- If the condition is true for all ghosts, the result is `true`, indicating escape is possible; otherwise, it's `false`.

In summary, the solution approach is to use a simple distance comparison strategy, verifying that you are closer to the target than any of the ghosts, allowing for a possible escape.

Let's say we have a target location at `[4, 3]` and there are two ghosts located at `[3, 0]` and at `[2, 5]`. To illustrate the solution approach:

1. Calculate our Manhattan distance to the target. Since we start at `[0, 0]`, our distance to the target at `[4, 3]` is `|4 - 0| + |3 - 0| = 4 + 3 = 7`.
2. Now, we compute the Manhattan distances for each ghost to the target:
   - For the first ghost at `[3, 0]`, the distance to the target `[4, 3]` is `|4 - 3| + |3 - 0| = 1 + 3 = 4`.
   - For the second ghost at `[2, 5]`, the distance to the target `[4, 3]` is `|4 - 2| + |3 - 5| = 2 + 2 = 4`.
3. Compare our distance to each ghost's distance to the target:
   - Our distance to the target is 7, which is greater than both ghost 1 and ghost 2's distance to the target (4 and 4 respectively). In this case, at least one ghost (in this example actually both) could reach the target faster than we can or at the same time if moving optimally.
4. Following the solution approach, since at least one ghost's Manhattan distance is less than or equal to ours, we cannot guarantee reaching the target first. Therefore, the function would return `false` indicating that escape is not possible under these conditions.

Solution Implementation

```python


from typing import List

class Solution:
    def escapeGhosts(self, ghosts: List[List[int]], target: List[int]) -> bool:
        # Unpacking the target's x and y coordinates
        target_x, target_y = target

        # Check if distance from each ghost to the target is greater than
        # the distance from player's starting point (0, 0) to the target.
        # If true for all ghosts, then escape is possible.
        return all(
            abs(target_x - ghost_x) + abs(target_y - ghost_y) > abs(target_x) + abs(target_y)
            for ghost_x, ghost_y in ghosts
        )
        # This comparison uses the Manhattan distance, which is the sum of the
        # absolute differences along each dimension (x and y in this case).

# Here, abs function is used to calculate the absolute value which is required
# to measure the Manhattan distance.
# The all function is used to ensure the condition must be true for all ghosts in the list.
```

Java

```java

class Solution {
    public boolean escapeGhosts(int[][] ghosts, int[] target) {
        // Store the target position coordinates for easier access
        int targetX = target[0];
        int targetY = target[1];

        // Go through all the ghost positions to determine if you can escape.
        for (int[] ghost : ghosts) {
            // Store ghost position coordinates for easier access
            int ghostX = ghost[0];
            int ghostY = ghost[1];

            // Calculate the Manhattan distance from the ghost to the target
            int ghostToTargetDistance = Math.abs(targetX - ghostX) + Math.abs(targetY - ghostY);
            // Calculate the Manhattan distance from the origin to the target
            int originToTargetDistance = Math.abs(targetX) + Math.abs(targetY);

            // If any ghost can reach the target quicker or at the same time as you can from the origin,
            // you cannot escape; return false.
            if (ghostToTargetDistance <= originToTargetDistance) {
                return false;
            }
        }

        // If none of the ghosts can reach the target quicker than you can from the origin,
        // you can escape; return true.
        return true;
    }
}
```

C++

```cpp

class Solution {
public:
    // Function to determine if it's possible to escape ghosts and reach the target
    bool escapeGhosts(vector<vector<int>>& ghosts, vector<int>& target) {
        // Store the target coordinates for easier access
        int targetX = target[0];
        int targetY = target[1];

        // Iterate through the list of ghosts
        for (auto& ghost : ghosts) {
            // Store the ghost's coordinates for easier access
            int ghostX = ghost[0];
            int ghostY = ghost[1];

            // Calculate the Manhattan distance from the ghost to the target
            int ghostToTargetDistance = abs(targetX - ghostX) + abs(targetY - ghostY);

            // Calculate the Manhattan distance from the player's start position (0,0) to the target
            int playerToTargetDistance = abs(targetX) + abs(targetY);

            // If any ghost can reach the target sooner or at the same time as the player,
            // it's not possible to escape, return false
            if (ghostToTargetDistance <= playerToTargetDistance) {
                return false;
            }
        }

        // If no ghost can reach the target sooner than the player, it's possible to escape, return true
        return true;
    }
};
```

TypeScript

```typescript

/**
 * Determines if the player can escape without any ghost reaching the target first.
 * @param ghosts - Array of positions of the ghosts, where each position is represented as an [x, y] tuple.
 * @param target - The target destination as an [x, y] tuple.
 * @returns boolean - true if the player can escape, otherwise false.
 */
function escapeGhosts(ghosts: number[][], target: number[]): boolean {
    // Destructure target coordinates into targetX and targetY for clarity.
    const [targetX, targetY] = target;

    // Loop over each ghost's position.
    for (const [ghostX, ghostY] of ghosts) {
        // Calculate the Manhattan distance from the ghost to the target.
        const ghostDistance = Math.abs(targetX - ghostX) + Math.abs(targetY - ghostY);
        // Calculate the Manhattan distance from the player's starting point (0, 0) to the target.
        const playerDistance = Math.abs(targetX) + Math.abs(targetY);

        // If a ghost can reach the target in a distance that's less than or
        // equal to the player's distance to the target, the player cannot escape.
        if (ghostDistance <= playerDistance) {
            return false;
        }
    }

    // If no ghost can reach the target before the player, the player can escape.
    return true;
}


from typing import List

class Solution:
    def escapeGhosts(self, ghosts: List[List[int]], target: List[int]) -> bool:
        # Unpacking the target's x and y coordinates
        target_x, target_y = target

        # Check if distance from each ghost to the target is greater than
        # the distance from player's starting point (0, 0) to the target.
        # If true for all ghosts, then escape is possible.
        return all(
            abs(target_x - ghost_x) + abs(target_y - ghost_y) > abs(target_x) + abs(target_y)
            for ghost_x, ghost_y in ghosts
        )
        # This comparison uses the Manhattan distance, which is the sum of the
        # absolute differences along each dimension (x and y in this case).

# Here, abs function is used to calculate the absolute value which is required
# to measure the Manhattan distance.
# The all function is used to ensure the condition must be true for all ghosts in the list.
```

Time and Space Complexity

The provided Python code calculates whether a player can escape from all the ghosts to reach a target position on a grid.

Time Complexity:

The time complexity of this code is $O(n)$, where n is the number of ghosts. This is because the code iterates through the list of ghosts once, computing the Manhattan distance between each ghost and both the target and origin point (0,0). The `all` function continues until it finds a ghost who can reach the target before the player or until it has checked all ghosts.

Space Complexity:

The space complexity of the code is $O(1)$. The space used does not grow with the input size (number of ghosts) since no additional data structures are utilized that depend on the number of ghosts. The variables `tx` and `ty` are used to store the coordinates of the target, but these use a constant amount of space. The space for the input list of ghosts and the target point is not counted, as these are inputs and not extra space used for processing.