1298. Maximum Candies You Can Get from Boxes

Hard   Breadth-First Search   Graph   Array   Leetcode Link

Problem Description

You start with an array of boxes, where each box is either open or closed, contains a certain number of candies, may contain keys to other boxes, and might have other boxes inside it. You initially have a set of boxes; these are the ones you can attempt to open if they are not locked. The goal is to collect as many candies as possible by opening boxes (if you can), collecting candies, and using the keys found inside to open new boxes, which in turn can also contain more candies, boxes, and keys.

Specifically:

- Each box has a status indicating whether it's open (`1`) or closed (`0`).
- Each box contains a certain number of candies.
- Each box may contain keys that allow you to open other boxes.
- Each box may contain other boxes.

You can only collect candies from boxes that are open. If you find a new box within a box, you can attempt to open it either with a key or if it's already open.

The task is to determine the maximum number of candies you can collect following these rules.

Intuition

To solve this problem, we can use a strategy similar to Breadth-First Search (BFS). BFS is a common approach in graph traversal algorithms which can also be applied to problems like this where there are layers of items to explore (in this case, boxes within boxes).

The main idea behind BFS is to systematically explore the data structures, visiting all neighbors before moving to the next level in the search space. In this problem, this translates to:

1. Taking all initially available, open boxes.
2. Collecting all candies from these boxes.
3. Using all available keys to open any boxes we have.
4. Exploring all boxes found within these boxes.

To ensure an efficient process, we maintain:

- A queue to keep track of all boxes we can open and need to process.
- A hash set (`has`) to keep track of all boxes we have at our disposal, regardless of whether they are open or closed.
- A set (`took`) to keep track of all boxes from which we have already taken candies.

A box is enqueued only if it's open and it has not been processed before, ensuring we don't recount candies or reprocess boxes. We continue this process until there are no more boxes left to open.

By iterating through the boxes in this manner, we guarantee that we've collected all possible candies we can get given the initial conditions.

Solution Approach

The solution is implemented using a Breadth-First Search (BFS) strategy. Here's a step-by-step breakdown of the approach:

1. Start by creating a `deque` (a double-ended queue) called `q` to hold the indexes of all open boxes we initially have (`status[i] == 1`) from `initialBoxes`.

2. Calculate the starting number of candies by summing up the candies in all initially open boxes. This is stored in an `ans` variable.

3. Create a set called `has` that keeps track of all the boxes we currently have (both open and closed), and another set called `took` which keeps track of all the boxes from which we've already taken candies.

4. Start the BFS loop by continuously dequeuing a box index from `q` as long as `q` is not empty. For each box index `i`:

   a. Iterate over the list of keys in `keys[i]`, and for each key `k`: - Set the status of the box with the label `k` to open (`status[k] = 1`). - If we have this box (`k in has`) and we haven't taken candies from it (`k not in took`), add the number of candies from that box to `ans`, mark it as `took`, and append the box index to queue `q` to process its contents.

   b. Iterate over the list of boxes in `containedBoxes[i]`, and for each contained box `j`: - Add box `j` to the set `has` (now we have this box). - If the box `j` is open (`status[j]`) and we haven't taken candies from it (`j not in took`), add the number of candies from that box to `ans`, mark it as `took`, and append the box index to queue `q` to process its contents.

5. Continue processing until `q` runs out of boxes. At the end of the BFS loop, `ans` holds the maximum number of candies that can be collected.

This approach ensures we are efficiently visiting each openable box exactly once and collecting all possible candies by keeping track of the boxes we have and those from which we've already collected candies.

Example Walkthrough

Let's illustrate the solution approach with a small example:

Suppose we have the following initial setup:

- `status = [1, 0, 1, 0]` means boxes at indices 0 and 2 are initially open.
- `candies = [7, 5, 3, 4]` means the boxes contain 7, 5, 3, and 4 candies, respectively.
- `keys = [[], [1], [3], []]` means box at index 1 has a key to box 1, and box at index 2 has a key to box 3.
- `containedBoxes = [[2], [3], [], []]` means box at index 0 has box 2 inside, and box at index 1 has box 3 inside.
- `initialBoxes = [0, 1]` means we initially have box 0 and 1 in our possession.

Using the solution approach, we proceed with the BFS strategy:

1. We start by creating a `deque` `q` with the open boxes from `initialBoxes`, which are box 0 and box 1. But since box 1 is closed, we only add box 0 to `q`: `q = deque([0])`.

2. We sum up the candies from initially open boxes, so `ans = candies[0] = 7`.

3. We create the sets `has = {0, 1}` since we have boxes 0 and 1 and `took = {0}` because we've taken candies from box 0.

4. We begin the BFS loop:

   - Dequeuing box 0 from `q`, no keys in `keys[0]` but we have a contained box `containedBoxes[0] = [2]`. We add box 2 to `has`, since it's open, add its candies to `ans` (total now 10), and mark it as `took`. Now, `q = deque([2])`.

   - Dequeuing box 2 from `q`, we find a key in `keys[2] = [3]` that opens box 3. We add candies from box 3 (since it's now open and we have it in `has`), increasing `ans` to 14, and add box 3 to `took`. No contained boxes in box 2, so we continue.

5. The `q` is now empty, and the BFS loop ends. `ans` is 14, which is the maximum number of candies we can collect following the rules.

By keeping track of which boxes we have and from which we've taken candies, along with opening new boxes using the keys we find, we've managed to collect all the candies we possibly could.

Python Solution

```python
from collections import deque
from typing import List

class Solution:
    def maxCandies(
        self,
        statuses: List[int],  # statuses of boxes (0 for locked, 1 for open)
        candies: List[int],  # number of candies in each box
        keys: List[List[int]],  # list of keys for each box
        containedBoxes: List[List[int]],  # list of boxes contained within each box
        initialBoxes: List[int],  # list of boxes you start with
    ) -> int:
        # Queue with all the open boxes we can go through
        queue = deque([box for box in initialBoxes if statuses[box] == 1])
        # Calculate initial number of candies from open boxes
        total_candies = sum(candies[box] for box in initialBoxes if statuses[box] == 1)
        # Keep track of all boxes in hand whether they are locked or not
        boxes_in_hand = set(initialBoxes)
        # Keep a set of boxes from which candies have been taken
        boxes_accessed = {box for box in initialBoxes if statuses[box] == 1}

        # Keep iterating while there are boxes in the queue
        while queue:
            current_box = queue.popleft()  # Take the first box from the queue
            # Iterate over the keys contained in the current box
            for key in keys[current_box]:
                statuses[key] = 1  # Use the key to open the box with the matching number
                # Check if the now-opened box is in hand and hasn't been accessed yet
                if key in boxes_in_hand and key not in boxes_accessed:
                    total_candies += candies[key]  # Add the candies from the box
                    boxes_accessed.add(key)  # Mark the box as accessed
                    queue.append(key)  # Add the box to the queue to process further

            # Iterate over the boxes contained within the current box
            for box_id in containedBoxes[current_box]:
                boxes_in_hand.add(box_id)  # Add the contained box to boxes in hand
                # Check if the box is open and hasn't been accessed yet
                if statuses[box_id] and box_id not in boxes_accessed:
                    total_candies += candies[box_id]  # Add candies from the contained box
                    boxes_accessed.add(box_id)  # Mark the contained box as accessed
                    queue.append(box_id)  # Add the contained box to the queue for further processing

        # Return the total number of candies collected
        return total_candies

```

Java Solution

```java
class Solution {

    public int maxCandies(int[] status, int[] candies, int[][] keys, int[][] containedBoxes, int[] initialBoxes) {
        int totalCandies = 0; // To keep track of the total number of candies collected
        int boxCount = status.length; // Number of boxes
        boolean[] boxHas = new boolean[boxCount]; // Keeps track of whether we have access to a box
        boolean[] boxOpened = new boolean[boxCount]; // Keeps track of whether we've opened a box
        Deque<Integer> queue = new ArrayDeque<>(); // Queue to process the boxes to be opened

        // Initialize by going through the initial boxes
        for (int boxIndex : initialBoxes) {
            boxHas[boxIndex] = true; // Mark that we have this box
            // If we can open it, add its candies and enqueue it for further processing
            if (status[boxIndex] == 1) {
                totalCandies += candies[boxIndex];
                boxOpened[boxIndex] = true;
                queue.offer(boxIndex);
            }
        }

        // Process the queue while there are boxes to open
        while (!queue.isEmpty()) {
            int currentBoxIndex = queue.poll(); // Take the next box from the queue

            // Process all keys in the current box
            for (int keyIndex : keys[currentBoxIndex]) {
                status[keyIndex] = 1; // Change the status to open for the boxes for which we now have keys
                // If we have not opened the box and now found the key, add candies and enqueue it
                if (boxHas[keyIndex] && !boxOpened[keyIndex]) {
                    totalCandies += candies[keyIndex];
                    boxOpened[keyIndex] = true;
                    queue.offer(keyIndex);
                }
            }

            // Process all boxes contained inside the current box
            for (int containedBoxIndex : containedBoxes[currentBoxIndex]) {
                boxHas[containedBoxIndex] = true; // Mark that we now have this box
                // If we can open it and haven't before, add candies and enqueue
                if (status[containedBoxIndex] == 1 && !boxOpened[containedBoxIndex]) {
                    totalCandies += candies[containedBoxIndex];
                    boxOpened[containedBoxIndex] = true;
                    queue.offer(containedBoxIndex);
                }
            }
        }

        // Return the total candies collected from all boxes we could open
        return totalCandies;
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    int maxCandies(vector<int>& statuses, vector<int>& candies, vector<vector<int>>& keys, vector<vector<int>>& containedBoxes, vec
        int totalCandies = 0; // This will hold the running total of candies collected.
        int numBoxes = statuses.size(); // Get the number of boxes.
        vector<bool> hasBox(numBoxes, false); // Tracks if we have a box.
        vector<bool> openedBox(numBoxes, false); // Tracks if we've opened a box.
        queue<int> openableBoxes; // Queue to hold boxes that can be opened.

        // Loop over all initially available boxes and try to open them.
        for (int boxId : initialBoxes) {
            hasBox[boxId] = true; // We have this box.
            // If the box is not locked, collect its candies and consider its contents.
            if (statuses[boxId]) {
                totalCandies += candies[boxId]; // Add candies from the current box.
                openedBox[boxId] = true; // Mark the box as opened.
                openableBoxes.push(boxId); // Add it to the queue of boxes to process.
            }
        }

        // Process boxes while there are openable boxes available.
        while (!openableBoxes.empty()) {
            int currentBoxId = openableBoxes.front();
            openableBoxes.pop();

            // Go through the keys obtained from the current box and try to open corresponding boxes.
            for (int key : keys[currentBoxId]) {
                statuses[key] = 1; // The box that corresponds to the key can now be opened.
                // If we have the box and have not yet opened it, collect candies and open it.
                if (hasBox[key] && !openedBox[key]) {
                    totalCandies += candies[key]; // Add candies from the box we can now open.
                    openedBox[key] = true; // Mark the box as opened.
                    openableBoxes.push(key); // Add it to the queue of boxes to process.
                }
            }

            // Go through all boxes contained within the current box.
            for (int containedBox : containedBoxes[currentBoxId]) {
                hasBox[containedBox] = true; // We have this box.
                // If the box is not locked and we haven't opened it yet, collect candies.
                if (statuses[containedBox] && !openedBox[containedBox]) {
                    totalCandies += candies[containedBox]; // Add candies from the contained box.
                    openedBox[containedBox] = true; // Mark the box as opened.
                    openableBoxes.push(containedBox); // Add it to the queue of boxes to process.
                }
            }
        }

        return totalCandies;
    }
};

```

Typescript Solution

```typescript
// Define the functions and variables in the global scope

// This function calculates the maximum number of candies one can collect
// from initially available boxes and any boxes we can subsequently open.
// statuses: An array indicating if a box is locked (0) or unlocked (1)
// candies: An array representing the number of candies in each box
// keys: An array of arrays where each subarray contains keys found in a box
// containedBoxes: An array of arrays where each subarray contains boxes found in a box
// initialBoxes: An array of the boxes we start with
function maxCandies(
  statuses: number[],
  candies: number[],
  keys: number[][],
  containedBoxes: number[][],
  initialBoxes: number[]
): number {
  let totalCandies = 0; // Running total of collected candies
  const numBoxes = statuses.length; // Total number of boxes
  const hasBox = new Array<boolean>(numBoxes).fill(false); // Tracks possession of boxes
  const openedBox = new Array<boolean>(numBoxes).fill(false); // Tracks whether a box is opened
  const openableBoxes: number[] = []; // Queue of boxes that can be opened

  // Initialize the process with the initially available boxes
  for (const boxId of initialBoxes) {
    hasBox[boxId] = true; // We have this box
    if (statuses[boxId] === 1) { // If the box is unlocked
      totalCandies += candies[boxId]; // Add its candies to the total
      openedBox[boxId] = true; // Mark the box as opened
      openableBoxes.push(boxId); // Add to the queue for processing
    }
  }

  // While there are boxes that can be opened, continue processing
  while (openableBoxes.length > 0) {
    const currentBoxId = <number>openableBoxes.shift(); // Get the next box to process

    // Process keys from the current box
    for (const key of keys[currentBoxId]) {
      statuses[key] = 1; // The corresponding box can now be opened (unlocked)
      if (hasBox[key] && !openedBox[key]) { // If we have and haven't opened this box
        totalCandies += candies[key]; // Collect its candies
        openedBox[key] = true; // Mark it as opened
        openableBoxes.push(key); // Add to queue for processing
      }
    }

    // Process all boxes contained within the current box
    for (const containedBox of containedBoxes[currentBoxId]) {
      hasBox[containedBox] = true; // We now have this box
      if (statuses[containedBox] === 1 && !openedBox[containedBox]) { // If it's unlocked and not opened
        totalCandies += candies[containedBox]; // Collect its candies
        openedBox[containedBox] = true; // Mark it as opened
        openableBoxes.push(containedBox); // Add to queue for processing
      }
    }
  }

  return totalCandies; // Return the total candies collected
}

// Example usage of the function
// const maxCollectedCandies = maxCandies([status array], [candies array], [keys array], [containedBoxes array], [initialBoxes arra

```

Time and Space Complexity

The time complexity of the given code is `O(N + K + B)` where `N` is the number of boxes, `K` is the total number of keys, and `B` is the total number of boxes inside the other boxes. This complexity arises because each box, key, and inner box is processed at most once. You process a box only when you have the key and it's available to you, which happens once for each box that meets these conditions. Similarly, you access each key and inner box only once.

The space complexity of the code is `O(N)`. This complexity comes from storing the states (`status`), the box information in `has` and `took`, and the maximum size of the queue (`q`) which can contain all the boxes at worst if all get unlocked concurrently. The `q` could be less than `O(N)` if the boxes are opened one by one, but in the worst case, if all the boxes are obtained and can be opened, they all could be in the queue. Hence, the `has` and `took` sets, along with the queue `q`, contribute to the space complexity, leading to `O(N)` in the worst-case scenario.