Daily Coding Problem #313

Problem

This problem was asked by Citrix.

You are given a circular lock with three wheels, each of which display the numbers `0` through `9` in order. Each of these wheels rotate clockwise and counterclockwise.

In addition, the lock has a certain number of "dead ends", meaning that if you turn the wheels to one of these combinations, the lock becomes stuck in that state and cannot be opened.

Let us consider a "move" to be a rotation of a single wheel by one digit, in either direction. Given a lock initially set to `000`, a target combination, and a list of dead ends, write a function that returns the minimum number of moves required to reach the target state, or `None` if this is impossible.

Solution

First note that at the start, we can turn any of our three wheels one digit up or down. So we can consider the combinations `001`, `009`, `010`, `090`, `100`, and `900` to be one unit away from the beginning position. Of course, if any of these combinations were in our set of deadends, they would not be valid moves. Given these new positions, we can continue to find all positions with distance two, such as `002` and `080`.

If we continue this process, making sure not to visit any combinations we have used in the past, we will eventually get to our target value, or run out of moves to make. From an algorithms perspective, we can implement this using a breadth-first search. At a given level `k` we pop from a queue all the valid positions, and use those to find the valid positions at level `k + 1`. If none of the current combinations work, we increment the level and try our new queue values.

```
from collections import deque

def unlock(start, target, deadends):
    queue = deque([start])
    visited = set(deadends)
    level = 0

    while queue:
        for _ in range(len(queue)):
            curr = queue.popleft()
            if curr == target:
                return level

            else:
                for i, wheel in enumerate(curr):
                    lower = curr[:i] + str((int(wheel) - 1) % 10) + curr[i + 1:]
                    higher = curr[:i] + str((int(wheel) + 1) % 10) + curr[i + 1:]

                    if lower not in visited:
                        visited.add(lower)
                        queue.append(lower)
                    if higher not in visited:
                        visited.add(higher)
                        queue.append(higher)

        level += 1

    return None
```

The space complexity of this algorithm can be measured by how many items must be stored in our queue at once. In the worst case, the target will be far enough from the start that we will end up going through almost all combinations. The memory required will therefore be bounded by $O(d^w)$, where `d` is the number of digits per wheel, and `w` is the number of wheels.

Since for each item we are slicing at various places to create the next positions, and a slice may be `O(w)`, the time complexity will be $O(w * d^w)$.