Daily Coding Problem #319

Problem

This problem was asked by Airbnb.

An 8-puzzle is a game played on a 3 x 3 board of tiles, with the ninth tile missing. The remaining tiles are labeled 1 through 8 but shuffled randomly. Tiles may slide horizontally or vertically into an empty space, but may not be removed from the board.

Design a class to represent the board, and find a series of steps to bring the board to the state [[1, 2, 3], [4, 5, 6], [7, 8, None]].

Solution

This is a tough problem to implement in an interview setting, but fear not: we will go through it step by step.

Let's get the challenging part out of the way first: the algorithm. We might first think to use a graph algorithm like Dijkstra's or breadth-first search, but there are some challenges. For one, there are tons of cycles. If we consider a state to be any configuration of digits in the puzzle, it is clear that we can go back and forth between any two adjacent states forever.

Even if we prohibit going back to previous states, moving tiles randomly around the board is extremely inefficient. What we need is an algorithm with the following properties:

- For any given position, we should evaluate successor states and select the best one.
- We should track multiple paths at once, and switch paths if there is a path more promising than the current one.

One algorithm with both of these qualities is A*.

The idea is that we will store all paths we are monitoring in a priority queue. With each item we pop from the queue, we:

- Check if the current state matches our goal, in which case we can return the move count and path.
- Find all possible successors to the current state.
- Push each unvisited successor into the heap, along with an updated move count and path.

Crucially, the priority used to push and pop items will be a heuristic which evaluates the closeness of the current state to our goal, plus the number of moves already traveled. This way, shorter paths, as well as those which look more promising, will be selected first.

```
def search(start):
    heap = []
    visited = set()
    heapq.heappush(heap, [start.heuristic, 0, start, ''])

    while heap:
        _, moves, board, path = heapq.heappop(heap)
        if board.tiles == board.goal:
            return moves, path

        visited.add(tuple(board.tiles))
        for successor, direction in board.get_moves():
            if tuple(successor.tiles) not in visited:
                item = [moves + 1 + successor.heuristic, moves + 1, successor, path +
direction]
                heapq.heappush(heap, item)

    return None
```

Note that this algorithm will require our board class to store the tiles and the goal state. We also must implement `get_moves`, which finds all valid successor states from a given position.

We will store the tiles as a simple list.

To find the next moves, we first locate the index of the empty square, represented as zero in our list. Next, we examine each of the four ways of swapping a tile vertically or horizontally into this square. If the movement is valid, we add the new board state into the list of successors to return, as well as the direction the tile was moved.

With this logic settled, we can begin to flesh out our class.

```
class Board:
    def __init__(self, nums, goal='123456780'):
        self.goal = list(map(int, goal))
        self.tiles = nums

        self.empty = self.tiles.index(0)
    ...

    def swap(self, empty, diff):
        tiles = copy(self.tiles)
        tiles[empty], tiles[empty + diff] = tiles[empty + diff], tiles[empty]
        return tiles

    def get_moves(self):
        successors = []
        empty = self.empty

        if empty // 3 > 0:
            successors.append((Board(self.swap(empty, -3)), 'D'))
        if empty // 3 < 2:
            successors.append((Board(self.swap(empty, +3)), 'U'))
        if empty % 3 > 0:
            successors.append((Board(self.swap(empty, -1)), 'R'))
        if empty % 3 < 2:
            successors.append((Board(self.swap(empty, +1)), 'L'))

        return successors
    ...
```

As you may have noticed above, A* depends heavily on the quality of the heuristic used. A bad heuristic is not just inefficient; it may doom the algorithm to failure! For this problem, we would like to estimate how close we are to the goal, which is a board that looks like this:

```
1 2 3
4 5 6
7 8 0
```

One useful metric we can take advantage of is Manhattan distance. To calculate the Manhattan distance of two items in a grid, we count up the number of horizontal or vertical moves it would take to get from one to the other. Our heuristic will then be the sum of each digit's Manhattan distance to the goal state.

```
class Board:
    def __init__(self, nums, goal='123456780'):
        self.goal = list(map(int, goal))
        self.tiles = nums
        self.original = copy(self.tiles)
        self.heuristic = self.heuristic()
    ...

    def manhattan(self, a, b):
        a_row, a_col = a // 3, a % 3
        b_row, b_col = b // 3, b % 3
        return abs(a_row - b_row) + abs(a_col - b_col)

    def heuristic(self):
        total = 0
        for digit in range(1, 9):
            total += self.manhattan(self.tiles.index(digit), self.goal.index(digit))
        return total
```

Putting it all together, given an arbitrary starting list of numbers, we can first initialize a board class with these numbers, and then use our search algorithm to find an efficient solution.

```
import heapq
from copy import copy
```

```
from copy import copy

class Board:
    def __init__(self, nums, goal='123456780'):
        self.goal = list(map(int, goal))
        self.tiles = nums
        self.empty = self.tiles.index(0)
        self.original = copy(self.tiles)
        self.heuristic = self.heuristic()

    def __lt__(self, other):
        return self.heuristic < other.heuristic

    def manhattan(self, a, b):
        a_row, a_col = a // 3, a % 3
        b_row, b_col = b // 3, b % 3
        return abs(a_row - b_row) + abs(a_col - b_col)

    def heuristic(self):
        total = 0
        for digit in range(1, 9):
            total += self.manhattan(self.original.index(digit), self.tiles.index(digit))
            total += self.manhattan(self.tiles.index(digit), self.goal.index(digit))
        return total

    def swap(self, empty, diff):
        tiles = copy(self.tiles)
        tiles[empty], tiles[empty + diff] = tiles[empty + diff], tiles[empty]
        return tiles

    def get_moves(self):
        successors = []
        empty = self.empty

        if empty // 3 > 0:
            successors.append((Board(self.swap(empty, -3)), 'D'))
        if empty // 3 < 2:
            successors.append((Board(self.swap(empty, +3)), 'U'))
        if empty % 3 > 0:
            successors.append((Board(self.swap(empty, -1)), 'R'))
        if empty % 3 < 2:
```

```
            successors.append((Board(self.swap(empty, +1)), 'L'))

        return successors

def search(start):
    heap = []
    closed = set()
    heapq.heappush(heap, [start.heuristic, 0, start, ''])

    while heap:
        _, moves, board, path = heapq.heappop(heap)
        if board.tiles == board.goal:
            return moves, path

        closed.add(tuple(board.tiles))
        for successor, direction in board.get_moves():
            if tuple(successor.tiles) not in closed:

                item = [moves + 1 + successor.heuristic, moves + 1, successor, path +
direction]
                heapq.heappush(heap, item)

    return float('inf'), None

def solve(nums):
    start = Board(nums)
    count, path = search(start)
    return count, path
```

The running time and space of A* is $O(b^d)$ in the worst case, where b is the average number of successors per state, and d is the length of the shortest path solution. Using our heuristic, however, reduces this considerably.

Finally, we should note that up to now we have assumed that a solution always exists, which in fact is not the case. To take an example, we will never be able to permute the following grid to our goal state:

```
1 2 3
4 5 6
8 7 0
```

In this case, we will end up evaluating every possible permutation of the board reachable from the starting state, which will be around `O(N!)`, where `N` is the number of digits.