Sliding Puzzle

On a 2x3 `board`, there are 5 tiles represented by the integers 1 through 5, and an empty square represented by 0.

A move consists of choosing `0` and a 4-directionally adjacent number and swapping it.

The state of the board is *solved* if and only if the `board` is `[[1,2,3],[4,5,0]].`

Given a puzzle board, return the least number of moves required so that the state of the board is solved. If it is impossible for the state of the board to be solved, return -1.

Examples:

```
Input: board = [[1,2,3],[4,0,5]]
Output: 1
Explanation: Swap the 0 and the 5 in one move.
```

```
Input: board = [[1,2,3],[5,4,0]]
Output: -1
Explanation: No number of moves will make the board solved.
```

```
Input: board = [[4,1,2],[5,0,3]]
Output: 5
Explanation: 5 is the smallest number of moves that solves the board.
An example path:
After move 0: [[4,1,2],[5,0,3]]
After move 1: [[4,1,2],[0,5,3]]
After move 2: [[0,1,2],[4,5,3]]
After move 3: [[1,0,2],[4,5,3]]
After move 4: [[1,2,0],[4,5,3]]
After move 5: [[1,2,3],[4,5,0]]
```

```
Input: board = [[3,2,4],[1,5,0]]
Output: 14
```

Note:

- `board` will be a 2 x 3 array as described above.
- `board[i][j]` will be a permutation of `[0, 1, 2, 3, 4, 5]`.

Solution 1

Update: **distance** in the follows changed to **displacement** to avoid confusion:

Convert array to string, e.g., [[1,2,3],[4,0,5]] -> "123405", hence the corresponding potential swap displacements are: -1, 1, -3, 3. Also note, charAt(2) and charAt(3) are not adjacent in original 2 dimensional int array and therefore are not valid swaps.

```
    public int slidingPuzzle(int[][] board) {
        Set<String> seen = new HashSet<>(); // used to avoid duplicates
        String target = "123450";
        // convert board to string - initial state.
        String s = Arrays.deepToString(board).replaceAll("\\[|\\]|,|\\s", "");
        Queue<String> q = new LinkedList<>(Arrays.asList(s));
        seen.add(s); // add initial state to set.
        int ans = 0; // record the # of rounds of Breadth Search
        while (!q.isEmpty()) { // Not traverse all states yet?
            // loop used to control search breadth.
            for (int sz = q.size(); sz > 0; --sz) {
                String str = q.poll();
                if (str.equals(target)) { return ans; } // found target.
                int i = str.indexOf('0'); // locate '0'
                int[] d = { 1, -1, 3, -3 }; // potential swap displacements.
                for (int k = 0; k < 4; ++k) { // traverse all options.
                    int j = i + d[k]; // potential swap index.
                    // conditional used to avoid invalid swaps.
                    if (j < 0 || j > 5 || i == 2 && j == 3 || i == 3 && j == 2) { c
ontinue; }
                    char[] ch = str.toCharArray();
                    // swap ch[i] and ch[j].
                    char tmp = ch[i];
                    ch[i] = ch[j];
                    ch[j] = tmp;
                    s = String.valueOf(ch); // a new candidate state.
                    if (seen.add(s)) { q.offer(s); } //Avoid duplicate.
                }
            }
            ++ans; // finished a round of Breadth Search, plus 1.
        }
        return -1;
    }
```

written by rock original link here

Solution 2

It is easier for me to think about the board as a string (e.g. "123450" is the solution). I have a map that tells me positions I can move zero from the current one. I also keep track of the zero position, so I do not have to search for it every time.

DFS

I am running DFS with memo to store the string and the smallest number of moves to achieve this permutation. Also, I am pruning all searches that go deeper that the best solution found so far.

```
unordered_map<int, vector<int>> moves{{0,{1,3}},{1,{0,2,4}},{2,{1,5}},{3,{0,4}},{4,
{3,5,1}},{5,{4,2}}};
void dfs(string s, unordered_map<string, int>& m, int cur_zero, int swap_zero, int c
ur_move, int& min_moves) {
  swap(s[cur_zero], s[swap_zero]);
  if (s == "123450") min_moves = min(min_moves, cur_move);
  if (cur_move < min_moves && (m[s] == 0 || m[s] > cur_move)) {
    m[s] = cur_move;
    for (auto new_zero : moves[swap_zero]) dfs(s, m, swap_zero, new_zero, cur_move +
1, min_moves);
  }
}
int slidingPuzzle(vector<vector<int>>& b, int min_moves = INT_MAX) {
  string s = to_string(b[0][0]) + to_string(b[0][1]) + to_string(b[0][2])
    + to_string(b[1][0]) + to_string(b[1][1]) + to_string(b[1][2]);

  dfs(s, unordered_map<string, int>() = {}, s.find('0'), s.find('0'), 0, min_moves)
;
  return min_moves == INT_MAX ? -1 : min_moves;
}
```

BFS

Even with the pruning, DFS solution is kind of slow for this problem. BFS is much faster, and below is the code using same ideas. Thanks @beetlecamera for good suggestions.

```
unordered_map<int, vector<int>> moves{{0,{1,3}},{1,{0,2,4}},{2,{1,5}},{3,{0,4}},{4,
{3,5,1}},{5,{4,2}}};
int slidingPuzzle(vector<vector<int>>& b) {
  string s = to_string(b[0][0]) + to_string(b[0][1]) + to_string(b[0][2])
    + to_string(b[1][0]) + to_string(b[1][1]) + to_string(b[1][2]);

  unordered_map<string, int> m({{s, 0}});
  queue<pair<string, int>> q({{s, s.find('0')}});

  for (;!q.empty() && q.front().first != "123450";q.pop()) {
    for (auto new_zero : moves[q.front().second]) {
      auto str = q.front().first;
      swap(str[q.front().second], str[new_zero]);
      if (m.insert({str, m[q.front().first] + 1}).second) q.push({ str, new_zero });
    }
  }
  return q.empty() ? -1 : m[q.front().first];
}
```

Follow-up tasks:

1. C++ does not have it, but in Java, we could use LinkedHashSet as both the hash and the queue (and get rid of stacks).
2. We could improve the performance by using integer instead of string (e.g. 123,450). Hash operations should be faster, but we will need to write our own swap function.
3. Use priority_queue to see if it boost the performance.**Updated**: tried this one. Looks like a hybrid between DFS and BFS. The performance is worse than DFS.

written by votrubac original link here

Solution 3

Its standard to use A* search to solve n-puzzle problem, and using Manhattan distance will make the heuristic consistent, in which case, we don't have to expand nodes that are already expanded and moved to the closed state.

```
    def slidingPuzzle(self, board):
        self.goal = [[1,2,3], [4,5,0]]
        self.score = [0] * 6

        self.score[0] = [[3, 2, 1], [2, 1, 0]]
        self.score[1] = [[0, 1, 2], [1, 2, 3]]
        self.score[2] = [[1, 0, 1], [2, 1, 2]]
        self.score[3] = [[2, 1, 0], [3, 2, 1]]
        self.score[4] = [[1, 2, 3], [0, 1, 2]]
        self.score[5] = [[2, 1, 2], [1, 0, 1]]

        heap = [(0, 0, board)]
        closed = []

        while len(heap) > 0:
            node = heapq.heappop(heap)
            if node[2] == self.goal:
                return node[1]
            elif node[2] in closed:
                continue
            else:
                for next in self.get_neighbors(node[2]):
                    if next in closed: continue
                    heapq.heappush(heap, (node[1] + 1 + self.get_score(next), node[
1] + 1, next))
            closed.append(node[2])
        return -1

    def get_neighbors(self, board):
        res = []
        if 0 in board[0]:
            r, c = 0, board[0].index(0)
        else:
            r, c = 1, board[1].index(0)

        for offr, offc in [[0, 1], [0, -1], [1, 0], [-1, 0]]:
            if 0 <= r + offr < 2 and 0 <= c + offc < 3:
                board1 = copy.deepcopy(board)
                board1[r][c], board1[r+offr][c+offc] = board1[r+offr][c+offc], boar
d1[r][c]
                res.append(board1)
        return res


    def get_score(self, board):
        score = 0
        for i in range(2):
            for j in range(3):
                score += self.score[board[i][j]][i][j]
        return score
```

Found some time to shorten some of the methods after the contest, here is the revised version:

```
def slidingPuzzle(self, board):
    self.scores = [0] * 6
    goal_pos = {1:(0, 0), 2:(0, 1), 3: (0, 2), 4: (1,0), 5:(1,1), 0:(1, 2)}

    for num in range(6): # Pre calculate manhattan distances
        self.scores[num] = [[abs(goal_pos[num][0] - i) + abs(goal_pos[num][1] - j)
for j in range(3)] for i in range(2)]

    Node = namedtuple('Node', ['heuristic_score', 'distance', 'board'])
    heap = [Node(0, 0, board)]
    closed = []

    while len(heap) > 0:
        node = heapq.heappop(heap)
        if self.get_score(node.board) == 0:
            return node.distance
        elif node.board in closed:
            continue
        else:
            for neighbor in self.get_neighbors(node.board):
                if neighbor in closed: continue
                heapq.heappush(heap, Node(node.distance + 1 + self.get_score(neighb
or), node.distance + 1, neighbor))
        closed.append(node.board)
    return -1

def get_neighbors(self, board):
    r, c = (0, board[0].index(0)) if 0 in board[0] else (1, board[1].index(0))
    res = []

    for offr, offc in [[0, 1], [0, -1], [1, 0], [-1, 0]]:
        if 0 <= r + offr < 2 and 0 <= c + offc < 3:
            board1 = copy.deepcopy(board)
            board1[r][c], board1[r+offr][c+offc] = board1[r+offr][c+offc], board1[r]
[c]
            res.append(board1)
    return res

def get_score(self, board):
    return sum([self.scores[board[i][j]][i][j] for i in range(2) for j in range(3)])
```

written by johnyrufus16 original link here