2056. Number of Valid Move Combinations On Chessboard

Leetcode Link

Explanation

In this problem, we are given a chessboard and some chess pieces. The board initially contains only one piece of each type, and their positions are given as input. We are tasked to find the number of unique configurations we can get by moving these pieces on the board according to their valid move rules.

For example, suppose we have the following input:

```
1
2
3 pieces = ["rook", "bishop"]
4 positions = [[1, 1], [2, 1]]
```

The initial configuration of the board will look like this:

```
1
2
3 R B . . . . . .
4 . . . . . . . .
5 . . . . . . . .
6 . . . . . . . .
7 . . . . . . . .
8 . . . . . . . .
9 . . . . . . . .
10 . . . . . . . .
```

where 'R' represents the rook at position (1, 1) and 'B' represents the bishop at position (2, 1).

We need to generate all possible moves for the given pieces and make the moves on the board. Then, we will calculate the unique configurations we can achieve from these moves.

Approach

The solution uses a Depth-First Search (DFS) approach to build all possible move combinations for the given pieces. After calculating these combinations, we can move the pieces on the board according to the moves and calculate the total unique configurations.

The DFS algorithm is implemented in the `getCombMoves` function. The function generates move combinations for each piece by adding all the valid moves of that piece in the current combination. It iterates through all the pieces and generates a move combination for each piece at depth 'i'.

After generating all move combinations, the solution uses the `dfs` function to explore these move combinations and calculate the total unique configurations. For each move combination, the solution checks whether the new board configuration is valid or not. A configuration is valid if all the pieces are within the board boundary (i.e., 1 <= x <= 8 and 1 <= y <= 8) and no two pieces are in the same position. If the configuration is valid, we continue exploring by moving the pieces in a depth-first manner and add the configuration in the answer set.

The `dfs` function uses a bitmask to represent the active pieces. The pieces that are masked as active will make their moves in this turn. In each step, it moves the active pieces according to the given move combination and checks if the new configuration is valid. If the configuration is valid, it stores the configuration in an unordered set and explores the possible next move combination.

The solution uses a hashing function (`hash`) to encode the board configuration into a unique integer value. This allows us to store the board configurations in an unordered set and efficiently calculate the number of unique configurations.

The time complexity of the solution is O(29^4 * 6 * 7).

Example

Now, let's walk through an example to see how the approach works:

Suppose we have this input:

```
1
2 python
3 pieces = ["rook", "bishop"]
4 positions = [[1, 1], [2, 1]]
```

1. First, we call the `getCombMoves` function with this input to generate all possible move combinations for the given pieces. The move combinations will be: [[(1, 0), (1, 1)], [(1, 0), (1, -1)], [(1, 0), (-1, 1)], [(1, 0), (-1, -1)], [(1, 0), (1, 1)], [(1, 0), (1, -1)], [(1, 0), (-1, 1)], [(1, 0), (-1, -1)], [(0, 1), (1, 1)], [(0, 1), (1, -1)], [(0, 1), (-1, 1)], [(0, 1), (-1, -1)], [(0, 1), (1, 1)], [(0, 1), (1, -1)], [(0, 1), (-1, 1)], [(0, 1), (-1, -1)]]
2. Next, we call the `dfs` function with the generated move combinations and the initial board configuration. The function will consider all the move combinations one by one and explore the new configurations.
3. For each move combination, the `dfs` function moves the active pieces on the board based on the bitmask. If the new configurations are valid, it continues exploring by making new moves.
4. After exploring all possible move combinations using DFS, the function calculates the total number of unique configurations. In this example, the function will return the count value.

Solution

```
1
2 python
3 from typing import List
4 from itertools import product
5
6 class Solution:
7     def countCombinations(self, pieces: List[str], positions: List[List[int]]) -> int:
8         n = len(pieces)
9         comb_moves = self.get_comb_moves(pieces)
10         board = [[x - 1, y - 1] for x, y in positions]
11
12         ans = set()
13         for move in comb_moves:
14             self.dfs(move, n, board, (1 << n) - 1, ans)
15
16         return len(ans)
17
18     def get_comb_moves(self, pieces: List[str]) -> List[List[tuple]]:
19         pieces_move = [self.get_move(piece) for piece in pieces]
20         return list(product(*pieces_move))
21
22     def get_move(self, piece: str) -> List[tuple]:
23         moves = {"rook": [(1, 0), (-1, 0), (0, 1), (0, -1)],
24                  "bishop": [(1, 1), (1, -1), (-1, 1), (-1, -1)],
25                  "queen": [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (1, -1), (-1, 1), (-1, -1)]}
26         return moves[piece]
27
28     def dfs(self, move: List[tuple], n: int, board: List[tuple], active_mask: int, ans: set):
29         if active_mask == 0:
30             return
31
32         ans.add(self.hash(board))
33         for next_active_mask in range(1, 1 << n):
34             if not (active_mask & next_active_mask):
35                 continue
36
37             # Make a copy of the board
38             next_board = list(board)
39
40             # Move active pieces
41             for i in range(n):
42                 if (next_active_mask >> i) & 1:
43                     next_board[i] = (next_board[i][0] + move[i][0], next_board[i][1] + move[i][1])
44
45             # Check the validity of the new configuration
46             if self.is_valid(next_board):
47                 self.dfs(move, n, next_board, next_active_mask, ans)
48
49     def is_valid(self, board: List[tuple]) -> bool:
50         # Check boundary and uniqueness of the positions
51         positions = set()
52         for x, y in board:
53             if x < 0 or x >= 8 or y < 0 or y >= 8:
54                 return False
55             if (x, y) in positions:
56                 return False
57             positions.add((x, y))
58         return True
59
60     def hash(self, board: List[tuple]) -> int:
61         hashed = 0
62         for x, y in board:
63             hashed = hashed * 64 + x * 8 + y
64         return hashed
```

The solution is implemented in Python, but the same approach can be used and implemented in other programming languages like Java, JavaScript, C++, or C#.# Solution in JavaScript

```
1
2 javascript
3 class Solution {
4     countCombinations(pieces, positions) {
5         const n = pieces.length;
6         const combMoves = this.getCombMoves(pieces);
7         const board = positions.map((x, y) => [x - 1, y - 1]);
8
9         const ans = new Set();
10         for (const move of combMoves) {
11             this.dfs(move, n, board, (1 << n) - 1, ans);
12         }
13
14         return ans.size;
15     }
16
17     getCombMoves(pieces) {
18         const piecesMove = pieces.map(piece => this.getMove(piece));
19         return product(...piecesMove);
20     }
21
22     getMove(piece) {
23         const moves = {
24             "rook": [[1, 0], [-1, 0], [0, 1], [0, -1]],
25             "bishop": [[1, 1], [1, -1], [-1, 1], [-1, -1]],
26             "queen": [[1, 0], [-1, 0], [0, 1], [0, -1], [1, 1], [1, -1], [-1, 1], [-1, -1]]
27         };
28         return moves[piece];
29     }
30
31     dfs(move, n, board, activeMask, ans) {
32         if (activeMask === 0) {
33             return;
34         }
35
36         ans.add(this.hash(board));
37         for (let nextActiveMask = 1; nextActiveMask < (1 << n); nextActiveMask++) {
38             if (!(activeMask & nextActiveMask)) {
39                 continue;
40             }
41
42             const nextBoard = [...board];
43
44             for (let i = 0; i < n; i++) {
45                 if ((nextActiveMask >> i) & 1) {
46                     nextBoard[i] = [nextBoard[i][0] + move[i][0], nextBoard[i][1] + move[i][1]];
47                 }
48             }
49
50             if (this.isValid(nextBoard)) {
51                 this.dfs(move, n, nextBoard, nextActiveMask, ans);
52             }
53         }
54     }
55
56     isValid(board) {
57         const positions = new Set();
58         for (const [x, y] of board) {
59             if (x < 0 || x >= 8 || y < 0 || y >= 8) {
60                 return false;
61             }
62
63             const key = x * 8 + y;
64             if (positions.has(key)) {
65                 return false;
66             }
67
68             positions.add(key);
69         }
70
71         return true;
72     }
73
74     hash(board) {
75         let hashed = 0;
76         for (const [x, y] of board) {
77             hashed = hashed * 64 + x * 8 + y;
78         }
79         return hashed;
80     }
81 }
82
83 function product(...arrays) {
84     const f = (a, b) => [].concat(...a.map(x => b.map(y => [].concat(x, y))));
85     return arrays.reduce(f, [[]]);
86 }
```

Solution in Java

```
1
2 java
3 import java.util.*;
4
5 class Solution {
6     public int countCombinations(List<String> pieces, List<List<Integer>> positions) {
7         int n = pieces.size();
8         List<List<int[]>> combMoves = getCombMoves(pieces);
9         List<int[]> board = new ArrayList<>();
10         for (List<Integer> pos : positions) {
11             board.add(new int[] {pos.get(0) - 1, pos.get(1) - 1});
12         }
13
14         Set<Integer> ans = new HashSet<>();
15         for (List<int[]> move : combMoves) {
16             dfs(move, n, board, (1 << n) - 1, ans);
17         }
18
19         return ans.size();
20     }
21
22     private List<List<int[]>> getCombMoves(List<String> pieces) {
23         List<List<int[]>> piecesMove = new ArrayList<>();
24         for (String piece : pieces) {
25             piecesMove.add(getMove(piece));
26         }
27
28         return cartesianProduct(piecesMove);
29     }
30
31     private List<int[]> getMove(String piece) {
32         Map<String, int[][]> moves = new HashMap<>();
33         moves.put("rook", new int[][] {{1, 0}, {-1, 0}, {0, 1}, {0, -1}});
34         moves.put("bishop", new int[][] {{1, 1}, {1, -1}, {-1, 1}, {-1, -1}});
35         moves.put("queen", new int[][] {{1, 0}, {-1, 0}, {0, 1}, {0, -1}, {1, 1}, {1, -1}, {-1, 1}, {-1, -1}});
36         return Arrays.asList(moves.get(piece));
37     }
38
39     private void dfs(List<int[]> move, int n, List<int[]> board, int activeMask, Set<Integer> ans) {
40         if (activeMask == 0) {
41             return;
42         }
43
44         ans.add(hash(board));
45         for (int i = 1; i < (1 << n); i++) {
46             int nextActiveMask = i;
47             if ((activeMask & nextActiveMask) == 0) {
48                 continue;
49             }
50
51             List<int[]> nextBoard = new ArrayList<>();
52             for (int[] pos : board) {
53                 nextBoard.add(pos.clone());
54             }
55
56             for (int j = 0; j < n; j++) {
57                 if ((nextActiveMask >> j) % 2 == 1) {
58                     nextBoard.set(j, new int[] {nextBoard.get(j)[0] + move.get(j)[0], nextBoard.get(j)[1] + move.get(j)[1]});
59                 }
60             }
61
62             if (isValid(nextBoard)) {
63                 dfs(move, n, nextBoard, nextActiveMask, ans);
64             }
65         }
66     }
67
68     private boolean isValid(List<int[]> board) {
69         Set<Integer> positions = new HashSet<>();
70         for (int[] pos : board) {
71             int x = pos[0], y = pos[1];
72             if (x < 0 || x >= 8 || y < 0 || y >= 8) {
73                 return false;
74             }
75
76             int key = x * 8 + y;
77             if (positions.contains(key)) {
78                 return false;
79             }
80
81             positions.add(key);
82         }
83
84         return true;
85     }
86
87     private int hash(List<int[]> board) {
88         int hashed = 0;
89         for (int[] pos : board) {
90             int x = pos[0], y = pos[1];
91             hashed = hashed * 64 + x * 8 + y;
92         }
93         return hashed;
94     }
95
96     private static <T> List<List<T>> cartesianProduct(List<List<T>> lists) {
97         List<List<T>> result = new ArrayList<>();
98         if (lists.size() == 0) {
99             result.add(Collections.emptyList());
100             return result;
101         }
102
103         List<T> firstList = lists.get(0);
104         List<List<T>> subresult = cartesianProduct(lists.subList(1, lists.size()));
105
106         for (T item : firstList) {
107             for (List<T> subitem : subresult) {
108                 List<T> newItem = new ArrayList<>();
109                 newItem.add(item);
110                 newItem.addAll(subitem);
111                 result.add(newItem);
112             }
113         }
114
115         return result;
116     }
117 }
```

These solutions for JavaScript and Java use the same approach as the Python solution explained earlier. The main difference between the implementations is the syntax and how the data structures are used in each language.

It is important to understand the core idea of the approach, which is a Depth-First Search (DFS) to build all possible move combinations for the given pieces. Then, the solution calculates the total unique configurations by iterating through the move combinations and applying them to the board. The key is to implement the functions getCombMoves (to calculate all possible move combinations) and dfs (to explore the move combinations and calculate the unique configurations).