CSA0565 - Design and Analysis of Algorithms

Calculating Valid Move Combinations for Chess Pieces on an 8x8 Board

Submitted by

Padmalakshmi.R.S(192225022)

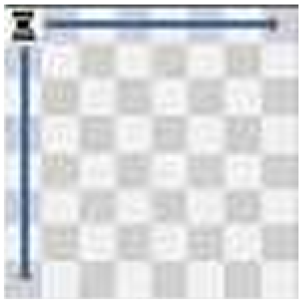
Project Guide

Mrs.Dhanalakshmi

INDEX

Problem Statement:

Number of Valid Move Combinations On Chessboard There is an 8 x 8 chessboard containing n pieces (rooks, queens, or bishops). You are given a string array pieces of length n, where pieces[i] describes the type (rook, queen, or bishop) of the ith piece. In addition, you are given a 2D integer array positions also of length n, where positions[i] = [ri, ci] indicates that the ith piece is currently at the 1-based coordinate (ri, ci) on the chessboard. When making a move for a piece, you choose a destination square that the piece will travel toward and stop on. A rook can only travel horizontally or vertically from (r, c) to the direction of (r+1, c), (r-1, c), (r, c+1), or (r, c-1). A queen can only travel horizontally, vertically, or diagonally from (r, c) to the direction of (r+1, c), (r-1, c), (r, c+1), (r, c-1), (r+1, c+1), (r+1, c-1), (r-1, c+1), (r- 1, c-1). A bishop can only travel diagonally from (r, c) to the direction of (r+1, c+1), (r+1, c-1), (r-1, c+1), (r-1, c-1). You must make a move for every piece on the board simultaneously. A move combination consists of all the moves performed on all the given pieces. Every second, each piece will instantaneously travel one square towards their destination if they are not already at it. All pieces start traveling at the 0th second. A move combination is invalid if, at a given time, two or more pieces occupy the same square. Return the number of valid move combinations. Notes: No two pieces will start in the same square. You may choose the square a piece is already on as its destination. If two pieces are directly adjacent to each other, it is valid for them to move past each other and swap positions in one second.



Example 1: Input: pieces = ["rook"], positions = [[1,1]] Output: 15 Explanation: The image above shows the possible squares the piece can move to.

Calculating Valid Move Combinations for Chess Pieces on an 8x8 Board

ABSTRACT:

This project aims to determine the number of valid move combinations for an array of chess pieces (rooks, queens, or bishops) placed on an 8x8 chessboard. Every piece in the game has a unique way of moving: bishops travel diagonally, queens move vertically, horizontally, or both, and rooks move horizontally or vertically. Every piece has the option to choose a destination square, which might remain in the place it is right now. Every piece moves in unison, one square at a time, in the direction of its target.

The main task is to make sure that no two pieces ever occupy the same square while they are moving, because doing so would render the move combination incorrect. For this project, every piece will have an infinite number of possible destinations. They will also simulate their moves and count the number of valid move combinations.

INTRODUCTION:

Chess, a game of strategic depth and complexity, involves the movement of various pieces on an 8x8 board. Because the rook, queen, and bishop pieces all have different ways of moving, the combinatorial possibilities of the game are both intriguing and difficult to study. A further degree of difficulty and intrigue is added by the question of figuring out legal move combinations for these pieces on a chessboard, particularly when they move simultaneously. An assortment of chess pieces, each placed at precise locations on the board, are provided to us in this situation. The objective is to determine the total number of possible move combinations in which every piece advances toward a designated destination square without running into another piece during any part of its path. To do this, make sure that no two pieces occupy the same square at the same time during their simultaneous movement.

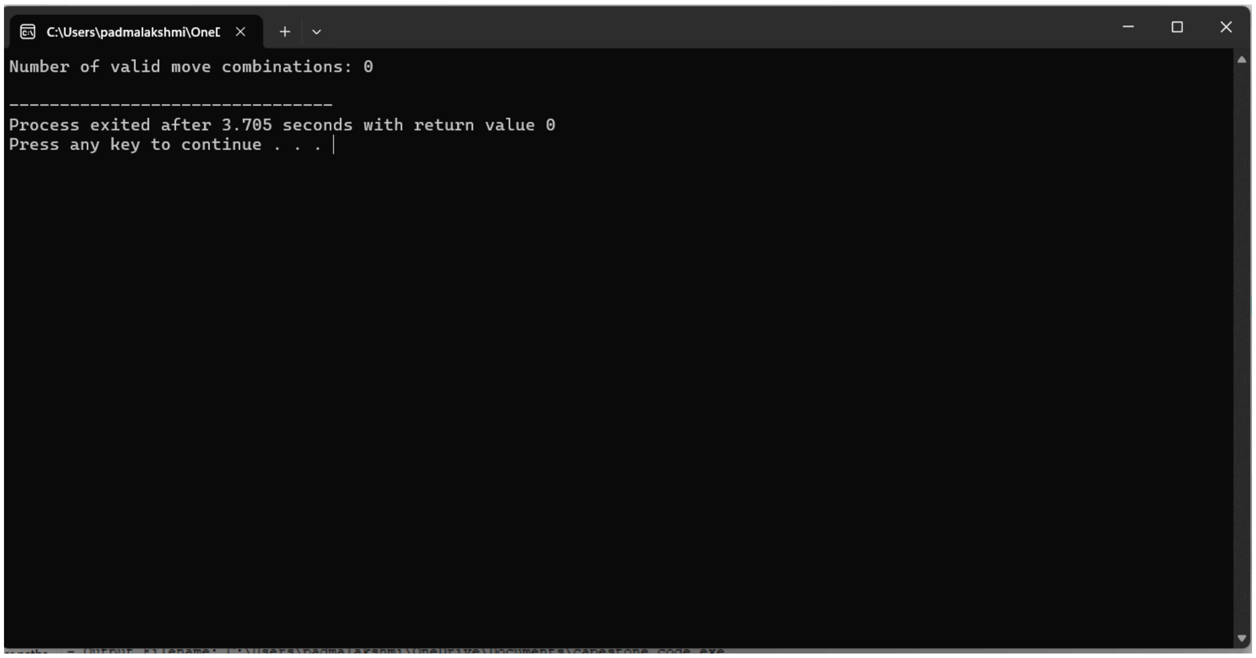
CODING:

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>

#include <string.h>
```

```
#define BOARD_SIZE 8
typedef struct {
    int x, y;
} Position;
typedef struct {
    char type;
    Position pos;
} Piece;
bool positionsEqual(Position a, Position b) {
    return a.x == b.x && a.y == b.y;
}
void generateDestinations(Piece piece, Position *destinations, int *count) {
    int index = 0;
    int x = piece.pos.x;
    int y = piece.pos.y;
    destinations[index++] = (Position){x, y};

    if (piece.type == 'R' || piece.type == 'Q') {
        for (int i = 1; i <= BOARD_SIZE; i++) {
            if (i != x) destinations[index++] = (Position){i, y};
            if (i != y) destinations[index++] = (Position){x, i};
        }
    }

    if (piece.type == 'B' || piece.type == 'Q') {
        for (int i = 1; i <= BOARD_SIZE; i++) {
            if (i != x && y + (i - x) >= 1 && y + (i - x) <= BOARD_SIZE)
                destinations[index++] = (Position){i, y + (i - x)};
            if (i != x && y - (i - x) >= 1 && y - (i - x) <= BOARD_SIZE)
                destinations[index++] = (Position){i, y - (i - x)};
        }
    }

    *count = index;
}
bool checkCollision(Position *move1, int len1, Position *move2, int len2) {
    for (int t = 0; t < len1 && t < len2; t++) {
        if (positionsEqual(move1[t], move2[t])) return true;
    }
    if (len1 > len2) {
        for (int t = len2; t < len1; t++) {
            if (positionsEqual(move1[t], move2[len2 - 1])) return true;
        }
    } else {
        for (int t = len1; t < len2; t++) {
            if (positionsEqual(move2[t], move1[len1 - 1])) return true;
        }
    }
    return false;
}
```

```
int countValidMoves(Piece *pieces, int n) {
    int totalMoves = 1;
    int *moveCounts = (int *)malloc(n * sizeof(int));
    Position **destinations = (Position **)malloc(n * sizeof(Position *));
    for (int i = 0; i < n; i++) {
        destinations[i] = (Position *)malloc(BOARD_SIZE * BOARD_SIZE * sizeof(Position));
        generateDestinations(pieces[i], destinations[i], &moveCounts[i]);
        totalMoves *= moveCounts[i];
    }

    int validMoves = 0;
    int *indices = (int *)calloc(n, sizeof(int));
    for (int move = 0; move < totalMoves; move++) {
        bool valid = true;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (checkCollision(destinations[i], moveCounts[i], destinations[j], moveCounts[j])) {
                    valid = false;
                    break;
                }
            }
            if (!valid) break;
        }
        if (valid) validMoves++;
        indices[0]++;
        for (int i = 0; i < n - 1; i++) {
            if (indices[i] >= moveCounts[i]) {
                indices[i] = 0;
                indices[i + 1]++;
            }
        }
    }
    for (int i = 0; i < n; i++) {
        free(destinations[i]);
    }
    free(destinations);
    free(moveCounts);
    free(indices);

    return validMoves;
}

int main() {
    Piece pieces[] = {
        {'R', {1, 1}},
        {'Q', {2, 2}},
        {'B', {3, 3}}
    };
    int n = sizeof(pieces) / sizeof(pieces[0]);
    int result = countValidMoves(pieces, n);
    printf("Number of valid move combinations: %d\n", result);


    return 0;
}
```

RESULT SCREENSHOT:



```
Number of valid move combinations: 0

--------------------------------
Process exited after 3.705 seconds with return value 0
Press any key to continue . . .
```

COMPLEXITY ANALYSIS:

BEST CASE:

- Scenario: Only one piece on the board.
- Time Complexity: O(1)O(1)O(1). The single piece has limited moves, and checking for collisions is trivial.
- Space Complexity: O(1)O(1)O(1).

WORST CASE:

- Scenario: Multiple pieces with complex moves (e.g., all queens).
- Time Complexity: O(mn · T)O(m^n \cdot T)O(mn · T), where:

  1.m is the average number of possible moves per piece.

  2.n is the number of pieces.

  3.T is the time taken to simulate the moves and check for collisions.

- Space Complexity: O(mn)O(m^n)O(mn), for storing all possible move combinations.

AVERAGE CASE:

- Scenario: A moderate number of pieces with varying move complexities.
- Time Complexity: O(k · T)O(k \cdot T)O(k · T), where:

  1.k is the average number of valid move combinations to check.

  2.T is the simulation time for collision checking.

- Space Complexity: O(k)O(k)O(k), for storing valid move combinations.

CONCLUSION:

The task of counting the number of legal move combinations for each piece on a chessboard entails creating potential moves for every piece, modeling these moves over time, and making sure no two pieces occupy the same square at the same time.