Chess State Risk

For any problem, we must start by setting up the algorithm for the problem. Thus, we can foresee the problems that may arise. And we determine what constructs to use in parts of the problem.

In Figure 1, 2, 3, I showed the algorithms I made for the program.

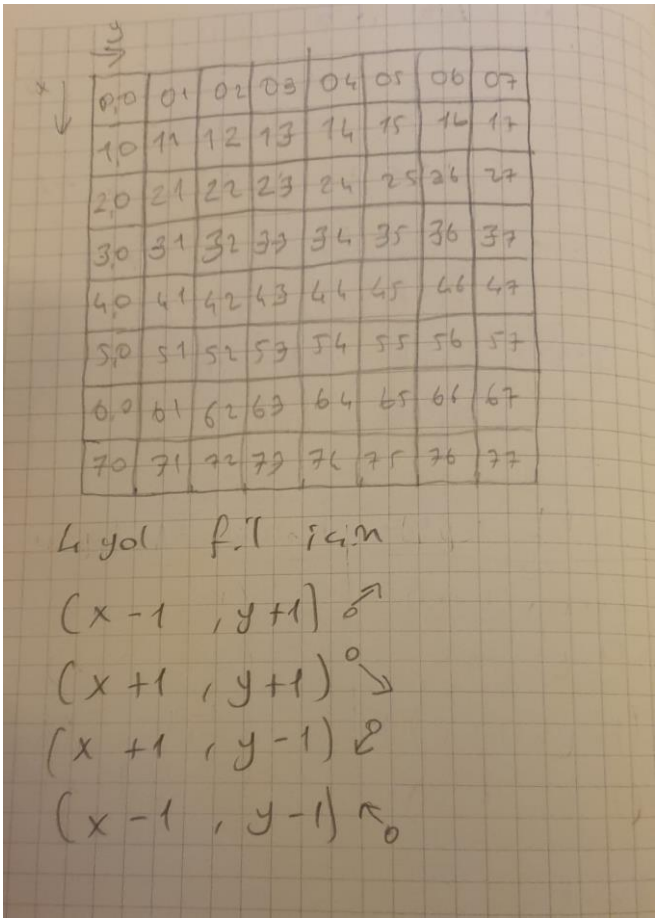

Figure 1

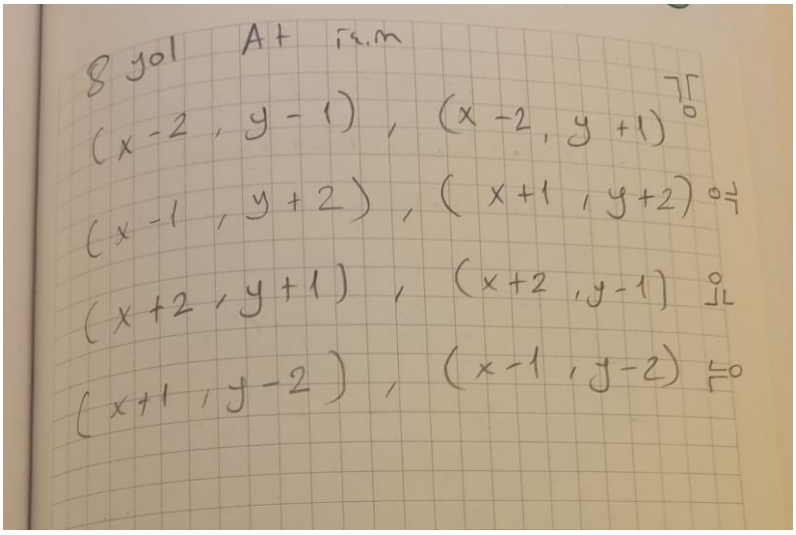

Figure 2

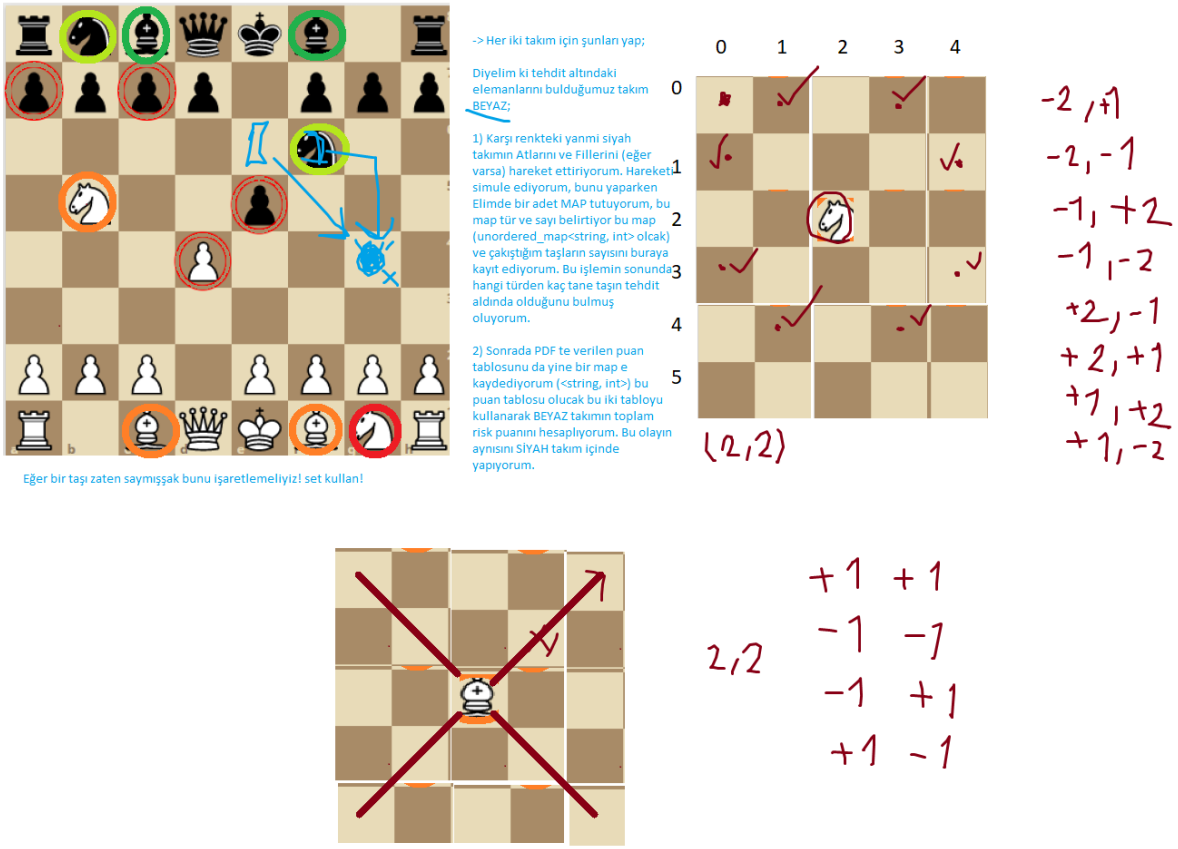

Figure 3

Here I determined the movement directions of knight and elephant stones and extracted their coordinates. And at the same time, I thought of the risks I might face.

```
149  int main() {
150
151      // Data structures
152      vector<vector<string>> chessBoard;
153      vector<string> inputFiles = { "board1.txt", "board2.txt", "board3.txt" };
154
```

Figure 4

```cpp
// Reads board"x".txt and fills the 2D chessBoard vector
void readFromFile(vector<vector<string>>& chessBoard, string &currentInputFile) {
    ifstream file(currentInputFile);
    string line;
    // read file line by line
    while (getline(file, line)) {
        // read line word by word
        string word = "";
        vector<string> row;
        for (int i = 0; i < line.size(); i++) {
            if (line[i] != ' ') {
                word += line[i];
            }
            else {
                row.push_back(word);
                word = "";
            }
        }
        row.push_back(word);
        chessBoard.push_back(row);
    }
    // Close file input
    file.close();
```

Figure 5

As I showed in Figures 4 and 5, I read the file and placed it in the 2D "chessboard" vector. The reason I used vectors was because it had properties.

```cpp
}

// Finds knight and elephand coordinates for both sides, fs as, fb ab
void findKnightAndElephandCoordinates(vector<vector<string>>& chessBoard, vector<vector<int>>& knightCoordinatesBlack,
    vector<vector<int>>& elephantCoordinatesBlack, vector<vector<int>>& knightCoordinatesWhite,
    vector<vector<int>>& elephantCoordinatesWhite) {
    for (int i = 0; i < chessBoard.size(); i++) {
        for (int j = 0; j < chessBoard[i].size(); j++) {
            if (chessBoard[i][j] == "fs") {
                elephantCoordinatesBlack.push_back({ i , j });
            }
            else if (chessBoard[i][j] == "as") {
                knightCoordinatesBlack.push_back({ i , j });
            }
            else if (chessBoard[i][j] == "fb") {
                elephantCoordinatesWhite.push_back({ i , j });
            }
            else if (chessBoard[i][j] == "ab") {
                knightCoordinatesWhite.push_back({ i , j });
            }
        }
    }
}
```

Figure 6

In Figure 6, I found only the knight and elephant coordinates as requested.

```
xx xx xx ks xx xx xx ss
xx ps kb xx xx xx fs xx
ps xx xx xx xx xx pb ps
xx xx xx fb xx xx xx xx
xx xx xx as pb xx xx xx
xx pb xx sb xx xx xx xx
pb xx xx xx ab vs xx xx
xx xx xx xx xx xx xx kb
```

Figure 7

```
Black Side Knight coordinates:
4, 3
Black Side Elephant coordinates:
1, 6
White Side Knight coordinates:
6, 4
White Side Elephant coordinates:
3, 3
```

Figure 8

I printed it on the screen to check that the coordinates are correct, as shown in Figures 7 and 8.

```
// Check if given point is inside
bool inside(int x, int y, vector<vector<string>>& chessBoard) {
    if (x < 0 || x >= chessBoard.size() || y < 0 || y >= chessBoard[x].size()) {
        return false;
    }
    return true;
}

// Move given knight coordinates and fill the map accordingly
void moveKnight(vector<vector<int>>& knightCoordinates, unordered_map<string, int>& stonesRiskCountMap, set<vector<int>>& visited,
    vector<vector<int>>& knightMoves, vector<vector<string>>& chessBoard, unordered_set<string>& stonesSet) {
    // For each knight
    for (vector<int>& currentPoint : knightCoordinates) {
        for (vector<int>& move : knightMoves) {
            int nx = currentPoint[0] + move[0];
            int ny = currentPoint[1] + move[1];
            // Keeps track of current stones that are under threat
            if (inside(nx, ny, chessBoard) && stonesSet.find(chessBoard[nx][ny]) != stonesSet.end()) {
                // Mark current coordiantes as visited
                visited.insert({ nx, ny });
                // Increment current stone by 1
                stonesRiskCountMap[chessBoard[nx][ny]]++;
            }
        }
    }
}
```

Figure 9

```
// All types of white stones
unordered_set<string> whiteStonesSet = { "pb", "kb", "ab", "fb", "vb", "sb" };
// To check if we have already visited a coordinate point
set<vector<int>> visited;
vector<string> whiteStones = { "pb", "kb", "ab", "fb", "vb", "sb" };
for (string& stone : whiteStones) {
    whiteStonesRiskCountMap.insert({ stone, 0 });
}

// Possible moves of the knight
vector<vector<int>> knightMoves = { {-2, 1}, {-2, -1}, {-1, +2}, {-1, -2}, {2, -1}, {2, 1}, {1, 2}, {1, -2} };

// Simulate black knight moves and count overlapping stones
moveKnight(knightCoordinatesBlack, whiteStonesRiskCountMap, visited, knightMoves, chessBoard, whiteStonesSet);
```

Figure 10

As seen in Figures 9 and 10, I moved the knight stone and determined whether the position I was moved was inside. I also checked the situations where the other party encountered the stones and the situation that created another risk for that stone, and I printed the results on the relevant vectors. A stone could threaten 2 of the opposing stones.

```
// Prints the current state of the chess board, Assume color = b -> white, color -> s -> black
void fillCountsMap(vector< vector<string> >& chessBoard, unordered_map<string, int>& countsMap, string color) {
    for (int i = 0; i < chessBoard.size(); i++) {
        for (int j = 0; j < chessBoard[i].size(); j++) {
            string stoneType = "", stoneColor = "";
            stoneType += chessBoard[i][j].front();
            stoneColor += chessBoard[i][j].back();
            if (stoneType != "x" && stoneColor == color) {
                countsMap[stoneType]++;
            }
        }
    }
}
```

Figure 11

```cpp
// Calculate risk table for white side
double whiteSideTotalPoints = 0;
for (auto& tuple : whiteStonesRiskCountMap) {
    string stoneType = "";
    stoneType += tuple.first[0];
    int numberOfStonesUnderThreat = tuple.second;
    //cout << "Stone Type: " << stoneType << " threatStones: " << numberOfStonesUnderThreat << " remainingStones: "
    //  << whiteStoneCounts[stoneType] - numberOfStonesUnderThreat << endl;
    whiteSideTotalPoints += (1.0 * numberOfStonesUnderThreat) * (1.0 * pointsMap[stoneType] / 2);
    whiteSideTotalPoints += 0.0 + ((whiteStoneCounts[stoneType] - numberOfStonesUnderThreat) * (pointsMap[stoneType]));
    //cout << whiteSideTotalPoints << endl;
}

cout << "White Side Risk Point: " << whiteSideTotalPoints << endl;
```

Figure 12

In Figure 11, I placed the relevant elements in the scoreboard vector. In Figure 12, I calculated and printed the points created by the relevant stones. Then I wrote the scores of all the .txt files to the "sonuçlar.txt" file.

I saw an error in the sample results file provided. The score for the black elements of case 3 is incorrect. In this case I showed on Figure 13.

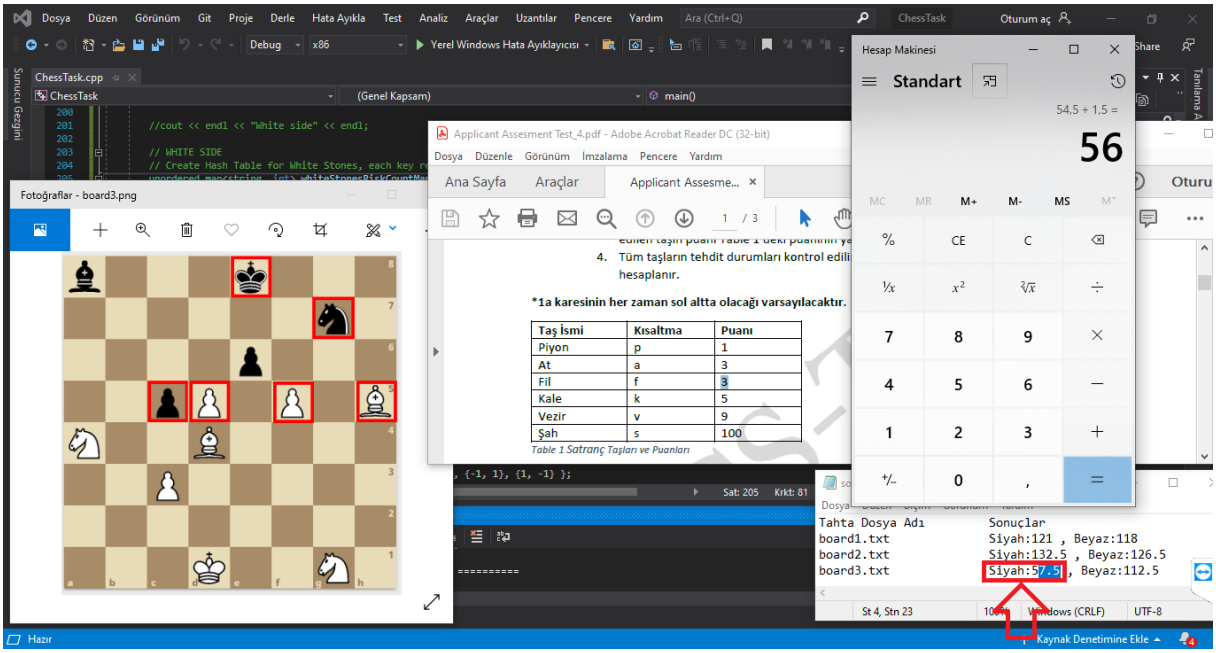

Figure 13

The issues I had difficulties while preparing the program:

1. I have never played chess before.

2. When an elephant travels in a certain plane, it faces the same type of element.

3. A stone threatens 2 stones at the same time from the opposite side.