Maryna Kolesnykova

Student_ID: 122475

Browsing the state space(F)

The algorithm of blind search (depth-first) can find (all) solutions (under typical computational - time and memory - conditions of a PC) only for chessboards of size 6x6, maximum 7x7. Implement this algorithm for chessboards with dimensions of 5x5 and 6x6, and try to find the first 5 solutions for each chessboard. For the 5x5 and 6x6 chessboards, choose 5 random starting points each (a total of 10 starting points), ensuring that one of these points is the bottom-left corner for each chessboard. Try to find (attempt to find) the first solution for each of these starting points. If you don't find it within the specified limit, signal unsuccessful search. In the discussion, analyze the observed results.

Contant

Solved problem

The goal of this project is to solve the Euler's Knight Problem on chessboards of various sizes (at least from 5x5 to 20x20). The task is to traverse the chessboard using legal moves of a chess knight so that each square of the chessboard is visited exactly once.

Maryna Kolesnykova

Student_ID: 122475

Brief description of the solution and its essential parts

The solution to this problem is implemented using a depth-first search algorithm, where the chessboard serves as the representation of the current state of the game. At the beginning, all squares of the chessboard are initialized to -1, and they are gradually filled with numbers corresponding to the knight's moves. The knight has 8 possible moves in general, unless restricted by the edge of the chessboard or already visited squares. These moves are represented as operators. The depth-first search algorithm proceeds by moving according to these operators while attempting to find the correct sequence of moves that visits all squares of the chessboard exactly once.

At the beginning of the program three values were initialized (mainly they are used in test to show the efficiency of the code):

- move_counter and over_10m_printed – counts the moves that the night has made (it has a limit of 10M moves and being checked at the beginning of the recursion in "knight_moves" function so the program can stop if the limit is reached)

```
if move_counter > 10000000 and not over_10m_printed:
    print("Moves limit of 10M moves is reached")
    over_10m_printed = True
    return False
```

- memory_counter – counts memory in bytes(used for testing the program and is not limited)

```
memory_counter = sys.getsizeof(chessboard)  # Calculate memory usage
print(f"Memory usage (bytes): {memory_counter}")
```

The next function is stay_in_matrix that checks if entered start coordinates fits the entered matrix size:

```
def stay_in_matrix(x_cur, y_cur, mtrx_size):
    if 0 <= x_cur < mtrx_size and 0 <= y_cur < mtrx_size:
        return True
```

Knight_moves is the most important function in the code (here the algorithm was applied). Firstly, it checks move limit and makes first move if the limit wasn't reached. Then it recursively makes following moves from the given set of moves and fills the chessboard with numbers from 1 to matrix_size^2 and signalize either success or failure of the algorithm

Maryna Kolesnykova

Student_ID: 122475

(detailed about the algorithm is written in) “specific algorithm used” paragraph below.

knight_tour calls knight_moves function and measure the time that is used to run the program.

```
def knight_tour(mtrx_size, start_x, start_y):
    chessboard = [[-1 for _ in range(mtrx_size)] for _ in range(mtrx_size)]

    chessboard[start_x][start_y] = 0

    start_time = time.time()
    if knight_moves(chessboard, start_x, start_y, counter: 1, mtrx_size, start_time):
        end_time = time.time()
        execution_time = end_time - start_time
        print("Success! \U0001F44D")
        print(f"Execution time in seconds: {execution_time: .5f}")
        tour(chessboard)
        return execution_time
    else:
        print("Not a successful tour or the time limit is over \U0001F614")
```

The very last function called “tour” prints out the successful tour of the knight:

```
def tour(chessboard):
    for row in chessboard:
        for cell in row:
            print(f'{cell:2}', end=' ')
        print()
```

Example of output:

```
Success! 👍
Execution time in seconds:  0.00900
25 22  9 14  5
10 15  6 23  8
21 24 19  4 13
16 11  2  7 18
 1 20 17 12  3
```

Maryna Kolesnykova

Student_ID: 122475

Data presentation of the problem

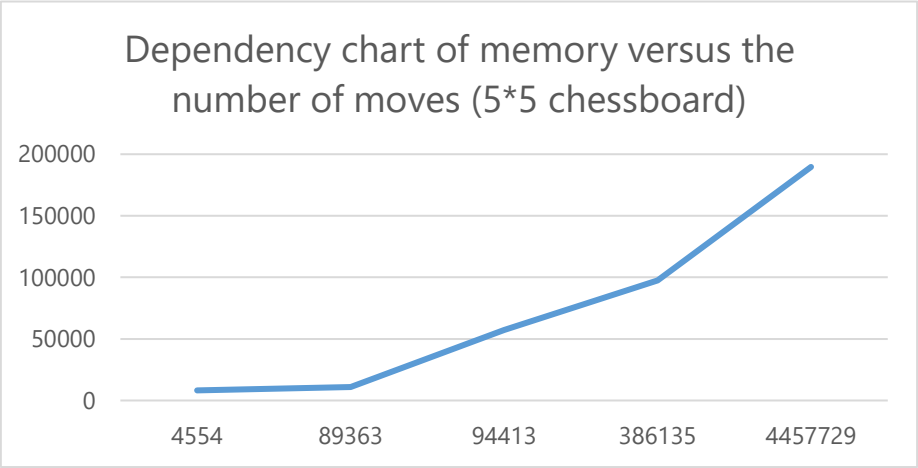
The data representation of the problem is based on the chessboard, where each square contains a number representing the knight's move. The available moves are represented as operators.

Following tables represents data usages and how it changes depending on number of moves

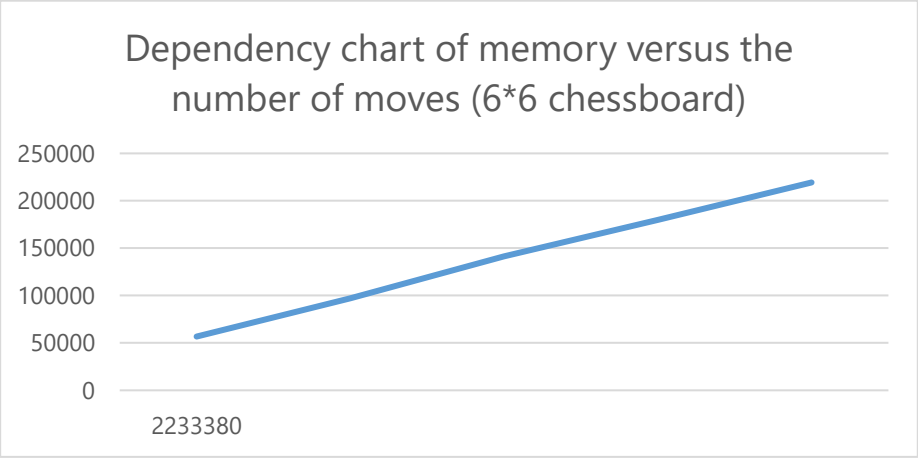
5*5 chessboard	
memory in bytes	number of moves
8192	4554
10846	89363
57098	94413
97456	386135
189654	4457729

6*6 chessboard	
memory in bytes	number of moves
56738	48653
97132	179830
141365	2233380
179856	4268102
219324	5198751

Graph representation:



5*5 chessboard



6*6 chessboard

Maryna Kolesnykova

Student_ID: 122475

Specific algorithms used

The algorithm used to solve the problem is the classic depth-first search algorithm. The algorithm explores all possible knight's moves while trying to find the correct sequence of moves that covers all squares of the chessboard exactly once. If the algorithm does not find a solution within a certain time limit or number of steps, it signals an unsuccessful search. First, all slots in the matrix are being initialized with values of "-1". Then the program tries to fill the -1 slots with positive values from 1 to matrix_size^2 that means that the night found his tour and the success could be signalized.

```
for x, y in moves:
    new_x, new_y = x_cur + x, y_cur + y
    if move_counter > 10000000 and not over_10m_printed:
        print("Moves limit of 10M moves is reached")
        over_10m_printed = True
        return False
    else:
        move_counter += 1
        if stay_in_matrix(new_x, new_y, mtrx_size) and chessboard[new_x][new_y] == -1:
            if knight_moves(chessboard, new_x, new_y, counter + 1, mtrx_size, start_time):
                return True

chessboard[x_cur][y_cur] = -1
return False
```

If the program cannot make a move, it goes a step back and tries another combination from the following list of moves:

```
    moves = [
        (2, 1), (1, 2), (-1, 2), (-2, 1),
        (-2, -1), (-1, -2), (2, -1), (1, -2)
    ]
```

If the program cannot find a tour for defined starting position it prints out a message that tells that the tour is impossible to build.

Testing approach and evaluation of the solution

To test the solution, we implemented the algorithm for chessboards of sizes 5x5 and 6x6. For each chessboard, we randomly selected 5 different starting points. One of these starting

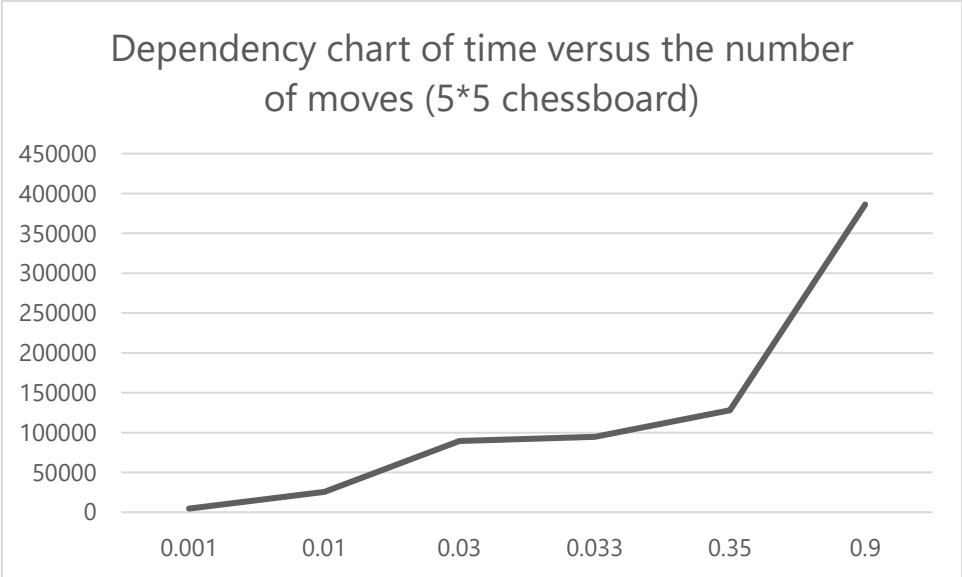
points was always the bottom-left corner of the chessboard. The goal was to find the first 5 solutions for each chessboard. In case a solution was not found within the defined limit (10 million steps or entered number of seconds), we marked the search as unsuccessful.

The first test was provided for chessboard 5*5. On the following table the results are shown including start and finish coordinates, time measurements, number of moves and memory usage.

start coordinates		finish coorditanes		time and memory complexity		
x	y	x	y	time in seconds	memory in bytes	number of moves
0	0	4	0	0.03	12650	94413
0	2	4	0	0.8	189654	4457729
4	0	0	0	0.001	8192	4554
4	2	4	0	0.09	97456	386135
2	2	0	4	0.032	10846	89363

Table 1 (chessboard 5*5)

The next diagram shows the dependency of time vs move number:



As we can see the dependency is represented as linear graph (just as we expected). It means that to make more move more time is needed.

Maryna Kolesnykova

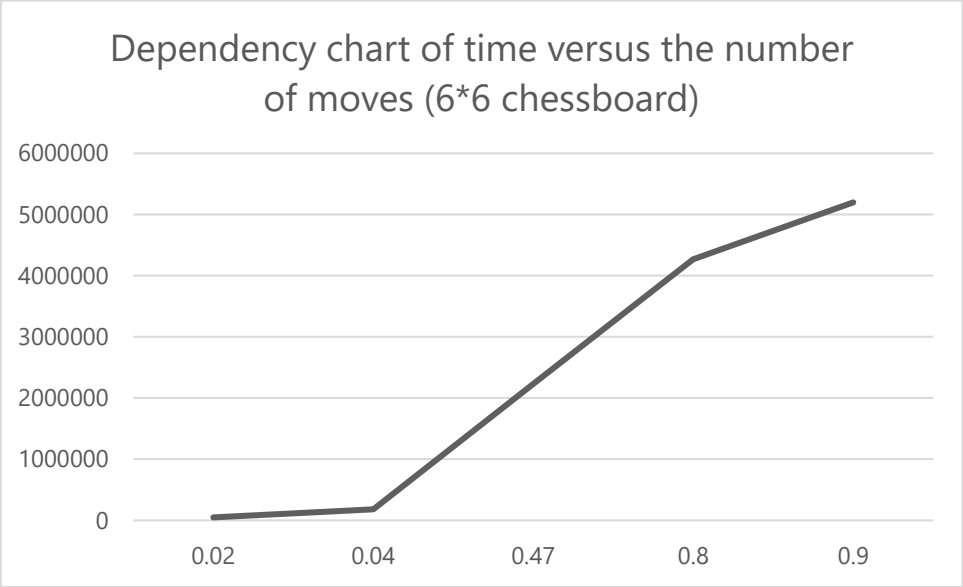
Student_ID: 122475

The same test has been provided for the 6*6 chessboard with the following results

start coordinates		finish coorditanes		time and memory complexity		
x	y	x	y	time in seconds	memory in bytes	number of moves
0	0	5	0	0.47	141365	2233380
0	4	0	5	0.8	179856	4268102
5	0	0	0	0.02	56738	48653
2	0	0	1	0.04	97132	179830
3	3	5	0	0.9	219324	5198751

Table 1 (chessboard 6*6)

The next diagram shows the dependency of time vs move number:



As we can see both graphs are linear. Memory usage growth with the increase of number of moves both for 5*5 and 6*6 matrix and easily to see from the tables 1 and 2 that the representation of memory consumption is also linear function.

Maryna Kolesnykova

Student_ID: 122475

Comparison of method properties for different solution lengths

1. Memory Usage: As the size of the chessboard increases, the memory usage of the algorithm also grows, especially when storing the entire chessboard state and move history. For very large chessboards, it may be necessary to optimize memory usage to prevent running out of memory.
2. Algorithm Complexity: The complexity of the algorithm increases with the size of the chessboard. The number of possible moves and the branching factor of the search tree both grow as the chessboard size increases. Therefore, more advanced algorithmic techniques, such as heuristics or intelligent pruning, may be required for larger chessboards.
3. Search Time: The time required to find a solution on larger chessboards can be substantial. The time limit for searching should be carefully chosen, balancing the desire for accurate solutions with the practicality of computation time.

Conclusion

In conclusion, the Euler's Knight Problem is a challenging puzzle that can be effectively solved using a depth-first search algorithm for smaller chessboards. However, for larger chessboards, the algorithm's efficiency and memory usage become significant concerns, and advanced optimization and parallelization techniques may be necessary to find solutions within a reasonable timeframe. Careful consideration of algorithmic complexity and search time limits is essential when tackling larger instances of the problem. Overall, this project provides a solid foundation for solving the Euler's Knight Problem and serves as a starting point for further exploration and optimization of the algorithm.