

# Homework Assignment

Artificial Intelligence

Student: Julia Florea

Calculatoare Engleza

Anul II, Grupa 2.2 A

Facultatea de Calculatoare, Automatică și Electronică

# 1 Introduction

## 1.1 Problem statement

Let us suppose the  $k \times k$  grid presented in Figure 2. The grid is configured with a pattern of walls. You are required to place  $n$  archers on this grid such that they cannot shoot each other. An archer can shoot up, down, left, right and also diagonally and its shoot can reach at most  $w$  locations in all directions, up to the grid edges.

- a. Write a detailed formulation for this search problem.
- b. Identify a search algorithm for this task and explain your choice.

## 1.2 Requirements

Your task is to analyze the problems and fulfill the following requirements:

R1 Implement the appropriate code to solve the assigned problem as a search problem.

R2 Present and comment your experimental results and choices in a meaningful way.

## 1.3 Project perspective

The project aims at developing a software for experimenting with heuristic search algorithms for problem solving. The homework is focused on demonstrating skills for programming search algorithms, covering coding, design, documentation and presentation of results.

The application is created using Python programming language. This particular choice was made due to the fact that I am currently trying to improve my skills and explore different techniques regarding Python, but also due to its flexibility and readability as a high-level programming language.

## 1.4 Objectives

The project is mainly based on the following objectives: creating a project based on heuristic search algorithms; implementing features in Python, focusing on writing clear, readable and efficient code; being familiar with resource reusability by making user defined function; designing appropriate algorithms for solving heuristic problems; creating documentation to keep record of all the steps followed when solving the problem; presenting the obtained results in a meaningful way to the reader; making the program efficient while optimising the algorithm; keeping track of the memory used for the program and the time complexity; learning to develop a complex project aimed at performing different given tasks in a timely manner.

## 2 Approach of the project

For the proposed problem it can be used the backtracking technique, which is a variant of the depth first search algorithm for searching solutions in an implicit graph.

### 2.1 Requirements

a. Formulation of the problem :

In the statement, the input of the problem is a board of size  $k \times k$  on which we must place  $n$  archers such that they cannot shoot each other under certain constraints : an archer can shoot in the directions up, down, left, right and diagonally. Thus, in the output we will print all possible solutions, containing distinct board configurations of the archers.

We have to think about the total number of permutations. If, for instance, we have an  $8 \times 8$  board, we can place the archer into  $8^2 = 64$  possible squares, then 63 for the second archer and so on. But that means the number of permutations would be very large, so we come to the conclusion of using combinations (because the order in which we pick the squares does not matter). This reduces the number of configurations, but it is still not ideal. For the algorithm to be optimal, we have to take into account the constraints given. Each archer must be placed in a different row, so we do not need to consider the previous row after we have placed a certain archer (because the next one has to be on the following row). The safeness of the position where the archer is placed must be checked : first we start with the top leftmost column, try the archer there and have a look where the archers above it are placed. If any of those are in a shooting position, then the current position is not safe. When a position is unsafe in one of the rows, there is no need to check anything beneath it.

b. Choice of algorithm :

I have chosen to use backtracking combined with DFS. In DFS, the current path is explored until there are no successor nodes. In backtracking, the nodes are only explored if they are potential, which means that through them we can get to a solution. For instance, if we have an array of elements  $a[0..n]$ , this is potential if a heuristic indicates that it can be extended to a complete solution. The heuristic is made of the constraints of the problem listed above. We first take a step and then see if this step is correct or not (whether it will give a correct answer or not). If it does not, then we come back and change our first step. This is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further.

Using this approach of backtracking is better than pure depth first search, as we do not need to explore each combination all the time until we reach the end of the grid. It is also better than using random search, regarding the running time, because for larger values of the grid, the random search takes significantly

more time to execute.

## 2.2 Solution idea

The idea is to place the archers one at a time on the board such that no archer is shot by the others. If the algorithm works by placing an archer on the top row, then placing one on the second row, then on the third, we can eliminate combinations as soon as we find a combination of the first few rows that does not work. Then, we backtrack and place the second archer in the second square. Again this combination does not work, so we try the second archer in the third square. This time the archer is safe, so we can progress to the third row and try the third archer in the first square and so on. This composes the backtracking method : exploring all possible valid solutions by working along the solution until we get to an error and then coming back to try an alternative solution.

## 2.3 Steps

1. Start in the leftmost row
2. If all archers are placed  
    return true
3. Loop through all the positions to find safe placement for each one  
    mark the current safe position as part of the solution  
    and check if placing archer here leads to a solution  
    if it leads to a solution  
        return true  
    else  
        unmark it and backtrack to check others
4. If all positions have been tried and did not work  
    return true and trigger backtracking

## 3 Algorithms

Algorithm for the entire program :

1. currentSolution[n]
2. for col0 = 0 to n-1
3.     currentSolution[0] = col0
4. for col1 = 0 to n-1
5.     if isSafe(row1, col1) then
6.         currentSolution[1] = col1
7.         for coln-1 = 0 to n-1
8.             if isSafe(rown-1, coln-1) then
9.                 currentSolution[n-1] = coln-1
10.             end if

Algorithms for the methods used in the program :

```
function isSafe(testRow, testCol) returns a boolean value
1. for row = 0 to testRow - 1
2.   if currentSolution[row] == testCol
3.     return false
4.   end if
5.   if abs(testrow - row) == abs(testCol - currentSolution[row])
6.     return false
7.   end if
8. next row
9. return true
end function
```

```
function placeArcher(row) returns list of all solutions
1. for col = 0 to n
2.   if ! isSafe(row, col)
3.     continue
4.   else
5.     currentSolution[row] = col
6.     if row == n-1
7.       solutions += currentSolution
8.     else
9.       placeArcher(row+1)
10. return solutions
end function
```

## 4 Application outline

### 4.1 Architecture of the program

The application is composed of one module named "n\_archers.py". The module contains the input data, as well as the output data.

The input data consists of a variable "n", which stores the size of the grid where n archers will be placed. The variable is an integer and it is inputted by the user at the beginning of the program.

The output data is represented by a list which stores all the possible solutions to the problem i.e. all the possible configurations of the archers on the grid with the inputted size and the number of solutions found.

## 4.2 Functions included in the module

`isSafe(testRow, testCol)` - The function takes as parameters the position of the archer that we want to test, composed of the row and the column. It checks if that certain position is safe for the archer (if it can not be shot by the others either vertically, horizontally or diagonally). It only needs to check rows that are above the current test row, since any rows beneath it are empty. For a diagonal position, it needs to count how many rows up the archer is and how many columns across it is. Thus, if the absolute values of the differences between the test row or column and the previous archer placed are the same, it means that is a diagonal attack. If the values are different, it means the archer is safe. If the archer is in the same column as the test column, then it is not safe. The function returns a boolean value : true if the position is safe or false if the position is not safe.

`placeArcher(row)` - The function places an archer in a row. It tests each of the columns in the certain row to find a safe position for the archer. It is a recursive function. The parameter is the row in which it will place the archer. It must have access to the list of total solutions (because it will add something to that list). It loops through each column in the row and calls the function " `isSafe` " to check if that position is safe. If it is not safe, we loop again and try the next column, but if it is safe, it saves the safe position in the current solution list. Then, it needs to decide whether it will recurse and go down to the next level or if it has reached the end of the recursion. If the current position is on the last row, the function needs to recurse down one more level, so we call the function with the index of the next row ( $row + 1$ ). The function backtracks, tries to place an archer on the row 0 (after it goes through all the solutions) and it returns the list of all the solutions.

## 5 Experiments and results

### 5.1 Computational complexity

In computer science, the computational complexity or simply complexity of an algorithm is the amount of resources required to run it. Particular focus is given to time and memory requirements. The complexity of a problem is the complexity of the best algorithms that allow solving the problem.

The study of the complexity of explicitly given algorithms is called analysis of algorithms, while the study of the complexity of problems is called computational complexity theory. Both areas are highly related, as the complexity of an algorithm is always an upper bound on the complexity of the problem solved by this algorithm. Moreover, for designing efficient algorithms, it is often fundamental to compare the complexity of a specific algorithm to the complexity of the problem to be solved. Also, in most cases, the only thing that is known about the complexity of a problem is that it is lower than the complexity of the most efficient known algorithms. Therefore, there is a large overlap between

analysis of algorithms and complexity theory.

## 5.2 Notations

Big-O Notation - it gives the worst-case scenario of an algorithm's growth rate; starting from slowest space growth (best) to fastest (worst):

1.  $O(1)$  – constant complexity – takes the same amount of space regardless of the input size
2.  $O(\log n)$  – logarithmic complexity – takes space proportional to the log of the input size
3.  $O(n)$  – linear complexity – takes space directly proportional to the input size
4.  $O(n \log n)$  – log-linear/quasilinear complexity – also called “linearithmic”, its space complexity grows proportionally to the input size and a logarithmic factor
5.  $O(n^2)$  – square complexity - grows proportionally to the square of the input size

Omega Notation – it gives the best-case scenario of an algorithm's complexity, opposite to big-O notation.

Theta Notation – represents a function that is within lower and upper bounds.

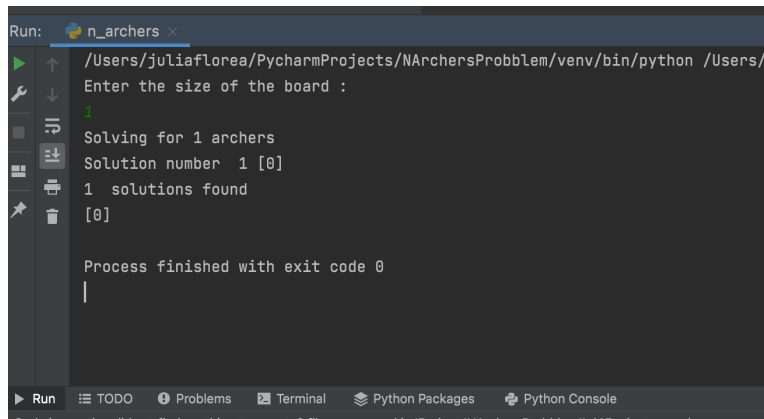
## 5.3 Complexity of the algorithm

The time complexity of the algorithm based on backtracking is the following :  $T(N) = N * T(N-1)$ , which simplifies to  $O(N!)$  Depth-first search with backtracking is significantly faster than using nested for loops (each loop has complexity of  $O(N)$  ) or random search, which would slow down the algorithm for larger values of  $n$ .

## 6 Test data and results

By running the algorithms of every function explained above, the program is expected to display the number of solutions and all the possible configurations of the archers ,taking into consideration the integer number inputted by the user.

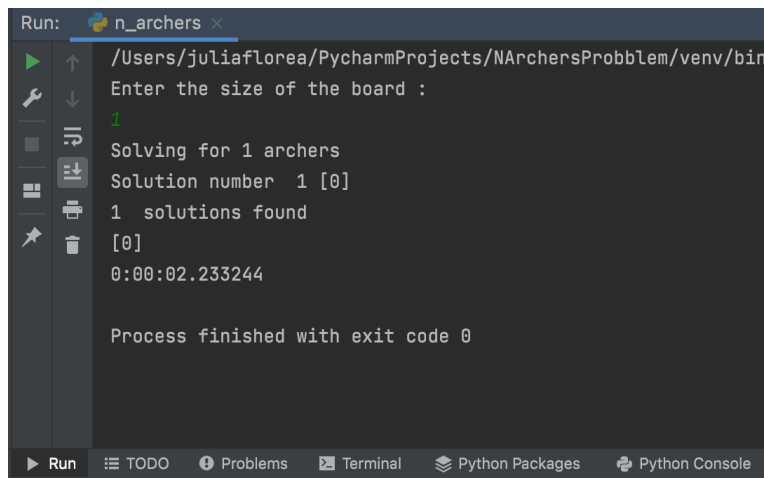
Let us consider a few examples, in which each input data is of a different size, their corresponding outputs and their execution time.

A screenshot of the PyCharm Run console for a program named 'n\_archers'. The console shows the execution path starting from the user's home directory, followed by the prompt 'Enter the size of the board :'. A green cursor indicates input. The program then prints 'Solving for 1 archers', 'Solution number 1 [0]', and '1 solutions found'. The final output is '[0]'. At the bottom, it states 'Process finished with exit code 0'. The interface includes a toolbar with icons for running, debugging, and other actions, and a status bar at the bottom with tabs for Run, TODO, Problems, Terminal, Python Packages, and Python Console.

```
Run: n_archers x
/Users/juliafloreana/PycharmProjects/NArchersProblem/venv/bin/python /Users/
Enter the size of the board :
Solving for 1 archers
Solution number 1 [0]
1 solutions found
[0]

Process finished with exit code 0
```

Figure 1: Size of  $n = 1$ ; Number of solutions found : 1

A screenshot of the PyCharm Run console for the same 'n\_archers' program. It shows the same sequence of outputs as Figure 1, but with an additional line '0:00:02.233244' indicating the execution time. The rest of the output, including 'Process finished with exit code 0', is identical to the previous figure. The interface elements are also the same.

```
Run: n_archers x
/Users/juliafloreana/PycharmProjects/NArchersProblem/venv/bin/
Enter the size of the board :
Solving for 1 archers
Solution number 1 [0]
1 solutions found
[0]
0:00:02.233244

Process finished with exit code 0
```

Figure 2: The execution time of  $n = 1$



```
Run: n_archers x
/Users/juliafloreana/PycharmProjects/NArchersPro
Enter the size of the board :
4
Solving for 4 archers
Solution number 1 [1, 3, 0, 2]
Solution number 2 [2, 0, 3, 1]
2 solutions found
[1, 3, 0, 2]
[2, 0, 3, 1]

Process finished with exit code 0
```

Figure 3: Size of  $n = 4$ ; Number of solutions found : 2

```
Run: n_archers x
/Users/juliafloreana/PycharmProjects/NArchersPro
Enter the size of the board :
4
Solving for 4 archers
Solution number 1 [1, 3, 0, 2]
Solution number 2 [2, 0, 3, 1]
2 solutions found
[1, 3, 0, 2]
[2, 0, 3, 1]
0:00:02.006438

Process finished with exit code 0
```

Figure 4: The execution time of  $n = 4$

```
Run: n_archers x
/Users/juliaflorea/PycharmProjects/NArchersProblem/ven
Enter the size of the board :
8
Solving for 8 archers
Solution number 1 [0, 4, 7, 5, 2, 6, 1, 3]
Solution number 2 [0, 5, 7, 2, 6, 3, 1, 4]
Solution number 3 [0, 6, 3, 5, 7, 1, 4, 2]
Solution number 4 [0, 6, 4, 7, 1, 3, 5, 2]
Solution number 5 [1, 3, 5, 7, 2, 0, 6, 4]
Solution number 6 [1, 4, 6, 0, 2, 7, 5, 3]
Solution number 7 [1, 4, 6, 3, 0, 7, 5, 2]
Solution number 8 [1, 5, 0, 6, 3, 7, 2, 4]
Solution number 9 [1, 5, 7, 2, 0, 3, 6, 4]
Solution number 10 [1, 6, 2, 5, 7, 4, 0, 3]
Solution number 11 [1, 6, 4, 7, 0, 3, 5, 2]
Solution number 12 [1, 7, 5, 0, 2, 4, 6, 3]
Solution number 13 [2, 0, 6, 4, 7, 1, 3, 5]
Solution number 14 [2, 4, 1, 7, 0, 6, 3, 5]
Solution number 15 [2, 4, 1, 7, 5, 3, 6, 0]
Solution number 16 [2, 4, 6, 0, 3, 1, 7, 5]
Solution number 17 [2, 4, 7, 3, 0, 6, 1, 5]
Solution number 18 [2, 5, 1, 4, 7, 0, 6, 3]
Solution number 19 [2, 5, 1, 6, 0, 3, 7, 4]
Solution number 20 [2, 5, 1, 6, 4, 0, 7, 3]
Solution number 21 [2, 5, 3, 0, 7, 4, 6, 1]
```

Figure 5: The size of  $n = 8$ ; The number of solutions found is 92.

```
[6, 1, 5, 2, 0, 3, 7, 4]
[6, 2, 0, 5, 7, 4, 1, 3]
[6, 2, 7, 1, 4, 0, 5, 3]
[6, 3, 1, 4, 7, 0, 2, 5]
[6, 3, 1, 7, 5, 0, 2, 4]
[6, 4, 2, 0, 5, 7, 1, 3]
[7, 1, 3, 0, 6, 4, 2, 5]
[7, 1, 4, 2, 0, 6, 3, 5]
[7, 2, 0, 5, 1, 4, 6, 3]
[7, 3, 0, 2, 5, 1, 6, 4]
0:00:05.511116

Process finished with exit code 0
```

Figure 6: The execution time of  $n = 8$ ; At this point, the running time starts to become larger, as the number of solutions increases.

```
↑ Solution number 2679 [10, 8, 5, 2, 9, 3, 0, 7, 4, 6, 1]
↓ Solution number 2680 [10, 8, 6, 4, 2, 0, 9, 7, 5, 3, 1]
⏮ 2680 solutions found
⏭ [0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
⏮ [0, 2, 5, 8, 1, 7, 10, 3, 6, 4, 9]
⏭ [0, 2, 6, 8, 3, 1, 9, 5, 10, 4, 7]
⏮ [0, 2, 6, 9, 1, 8, 5, 3, 10, 7, 4]
⏭ [0, 2, 6, 9, 7, 10, 1, 3, 5, 8, 4]
⏮ [0, 2, 6, 10, 3, 7, 9, 4, 1, 5, 8]
⏭ [0, 2, 7, 5, 8, 1, 4, 10, 3, 6, 9]
⏮ [0, 2, 7, 9, 3, 8, 10, 4, 6, 1, 5]
⏭ [0, 2, 7, 10, 6, 1, 9, 5, 3, 8, 4]
⏮ [0, 2, 9, 6, 8, 10, 1, 3, 5, 7, 4]
⏭ [0, 2, 9, 7, 3, 10, 8, 5, 1, 4, 6]
⏮ [0, 3, 5, 8, 1, 9, 4, 2, 7, 10, 6]
⏭ [0, 3, 5, 9, 2, 10, 7, 4, 1, 8, 6]
⏮ [0, 3, 6, 8, 10, 1, 9, 5, 2, 4, 7]
⏭ [0, 3, 6, 9, 1, 4, 7, 10, 2, 5, 8]
⏮ [0, 3, 6, 10, 7, 2, 4, 8, 1, 5, 9]
```

Figure 7: The size of  $n = 11$ ; The number of solutions found:2680

```
[10, 8, 3, 5, 2, 9, 8, 0, 7, 4, 1]
[10, 8, 4, 0, 7, 3, 1, 6, 9, 5, 2]
[10, 8, 4, 1, 3, 0, 9, 7, 5, 2, 6]
[10, 8, 4, 1, 9, 2, 5, 7, 0, 3, 6]
[10, 8, 4, 2, 7, 9, 1, 5, 0, 6, 3]
[10, 8, 5, 2, 9, 3, 0, 7, 4, 6, 1]
[10, 8, 6, 4, 2, 0, 9, 7, 5, 3, 1]
0:00:05.630759

Process finished with exit code 0
```

Figure 8: The execution time of  $n = 11$

```
n_archers x
Solution number 6160 [5, 7, 0, 10, 3, 1, 9, 11, 8, 2, 4, 6]
Solution number 6161 [5, 7, 0, 10, 3, 6, 9, 2, 8, 11, 4, 1]
Solution number 6162 [5, 7, 0, 10, 6, 2, 9, 11, 3, 8, 4, 1]
Solution number 6163 [5, 7, 0, 10, 6, 2, 9, 11, 4, 8, 1, 3]
Solution number 6164 [5, 7, 0, 10, 8, 1, 3, 9, 11, 2, 4, 6]
Solution number 6165 [5, 7, 0, 10, 8, 2, 9, 3, 1, 11, 4, 6]
Solution number 6166 [5, 7, 0, 10, 8, 4, 1, 9, 2, 6, 11, 3]
Solution number 6167 [5, 7, 0, 11, 3, 1, 9, 4, 8, 10, 2, 6]
Solution number 6168 [5, 7, 0, 11, 3, 6, 9, 2, 4, 1, 10, 8]
Solution number 6169 [5, 7, 0, 11, 3, 8, 6, 9, 2, 10, 1, 4]
Solution number 6170 [5, 7, 0, 11, 3, 8, 10, 4, 1, 9, 2, 6]
Solution number 6171 [5, 7, 0, 11, 6, 8, 10, 2, 9, 3, 1, 4]
```

Figure 9: The size of  $n = 12$ ; The number of solutions found:6160

```
[11, 9, 6, 0, 2, 10, 1, 7, 4, 8, 3, 5]
[11, 9, 6, 1, 3, 0, 7, 10, 8, 5, 2, 4]
[11, 9, 6, 1, 3, 8, 0, 2, 10, 5, 7, 4]
[11, 9, 6, 4, 1, 7, 0, 2, 8, 5, 3, 10]
[11, 9, 7, 1, 4, 2, 0, 8, 10, 5, 3, 6]
[11, 9, 7, 2, 4, 1, 10, 0, 5, 3, 8, 6]
[11, 9, 7, 2, 4, 1, 10, 0, 6, 3, 5, 8]
[11, 9, 7, 4, 2, 0, 6, 1, 10, 5, 3, 8]
0:00:15.889874
```

Figure 10: The execution time of  $n = 12$

```
n_archers x
Solution number 351 [8, 6, 2, 7, 1, 4, 0, 5, 3]
Solution number 352 [8, 6, 3, 1, 7, 5, 0, 2, 4]
352 solutions found
[0, 2, 5, 7, 1, 3, 8, 6, 4]
[0, 2, 6, 1, 7, 4, 8, 3, 5]
[0, 2, 7, 5, 8, 1, 4, 6, 3]
[0, 3, 1, 7, 5, 8, 2, 4, 6]
[0, 3, 5, 2, 8, 1, 7, 4, 6]
[0, 3, 5, 7, 1, 4, 2, 8, 6]
```

Figure 11: The size of  $n = 9$ ; The number of solutions found: 352

```
[8, 5, 2, 6, 1, 7, 4, 0, 3]
[8, 5, 3, 1, 7, 4, 6, 0, 2]
[8, 5, 3, 6, 0, 7, 1, 4, 2]
[8, 5, 7, 1, 3, 0, 6, 4, 2]
[8, 6, 1, 3, 0, 7, 4, 2, 5]
[8, 6, 2, 7, 1, 4, 0, 5, 3]
[8, 6, 3, 1, 7, 5, 0, 2, 4]
0:00:10.174594
```

Figure 12: The execution time of  $n = 9$

```

[8, 1, 11, 0, 7, 9, 4, 2, 10, 3, 6, 12, 5]
[8, 1, 11, 0, 10, 6, 4, 2, 12, 3, 9, 7, 5]
[8, 1, 11, 2, 0, 7, 10, 4, 9, 12, 5, 3, 6]
[8, 1, 11, 2, 0, 7, 10, 12, 3, 6, 4, 9, 5]
[8, 1, 11, 2, 5, 7, 12, 10, 3, 6, 0, 9, 4]
[8, 1, 11, 2, 5, 9, 0, 10, 3, 6, 12, 7, 4]
[8, 1, 11, 2, 5, 9, 4, 10, 3, 6, 12, 7, 0]
[8, 1, 11, 2, 6, 9, 12, 0, 4, 10, 7, 5, 3]
[8, 1, 11, 2, 6, 10, 3, 5, 9, 12, 4, 0, 7]
[8, 1, 11, 2, 6, 10, 12, 0, 3, 5, 7, 9, 4]
[8, 1, 11, 2, 6, 12, 10, 5, 3, 0, 4, 7, 9]
[8, 1, 11, 2, 7, 12, 3, 0, 4, 10, 5, 9, 6]
[8, 1, 11, 2, 10, 6, 4, 0, 12, 3, 5, 7, 9]

```

Figure 13: The size of  $n = 13$  ; The number of solutions found: 7001

```

[12, 10, 8, 2, 4, 1, 9, 0, 5, 7, 11, 3, 6]
[12, 10, 8, 2, 4, 1, 9, 11, 6, 0, 3, 5, 7]
[12, 10, 8, 3, 0, 4, 9, 1, 5, 7, 11, 6, 2]
[12, 10, 8, 3, 5, 0, 9, 1, 6, 11, 7, 2, 4]
[12, 10, 8, 3, 5, 2, 9, 11, 0, 7, 4, 6, 1]
[12, 10, 8, 3, 11, 4, 1, 9, 0, 5, 7, 2, 6]
[12, 10, 8, 4, 0, 3, 9, 6, 1, 11, 7, 5, 2]
[12, 10, 8, 6, 1, 3, 0, 7, 9, 11, 5, 2, 4]
[12, 10, 8, 6, 4, 2, 0, 11, 9, 7, 5, 3, 1]
[12, 10, 8, 11, 4, 1, 3, 0, 9, 7, 5, 2, 6]
0:01:50.294494

```

Figure 14: The execution time of  $n = 13$

Solution number	2654	[10, 6, 2, 9, 5, 1, 8, 4, 0, 7, 3]
Solution number	2655	[10, 6, 3, 1, 4, 8, 0, 9, 7, 5, 2]
Solution number	2656	[10, 6, 3, 9, 4, 8, 0, 2, 7, 5, 1]
Solution number	2657	[10, 6, 4, 1, 7, 0, 2, 8, 5, 3, 9]
Solution number	2658	[10, 6, 4, 2, 0, 8, 3, 1, 9, 7, 5]
Solution number	2659	[10, 6, 9, 3, 0, 4, 8, 1, 5, 7, 2]
Solution number	2660	[10, 6, 9, 3, 1, 8, 2, 5, 7, 0, 4]
Solution number	2661	[10, 7, 1, 4, 0, 8, 3, 9, 6, 2, 5]
Solution number	2662	[10, 7, 2, 4, 1, 9, 0, 5, 3, 8, 6]
Solution number	2663	[10, 7, 2, 4, 8, 1, 5, 9, 6, 0, 3]
Solution number	2664	[10, 7, 2, 8, 3, 9, 0, 5, 1, 4, 6]

Figure 15: The size of  $n = 22$  ; The number of solutions found: 10224

Number of archers	Number of solutions	Running Time
1	1	0.2
4	2	0.2
8	92	0.05
9	352	0.10
11	2680	0.5
12	6160	0.15
13	7001	0.50
15	9023	0.71
22	10224	2.46
24	10500	out of time

## 7 Conclusions

To conclude, in developing a solution to a problem, it can be quite beneficial to consider the data and the format in which they are presented. It can be a waste of resources (time and memory in the case of computing) to manipulate data that communicates more information than needed to answer a given question. The main purposes of this project are to place  $n$  archers on a grid board, considering all the possible directions from where they can be shot by the others, formulate an efficient search algorithm for this matter and present its results. After the completion of the assignment, I became familiar with a various number of features provided by the Python language and LaTeX. At the same time, I've managed to create a programming work style, which enables me to arrange the source code in a much more readable and organized way.

The most challenging part of the project was to choose the search algorithm which was to be used for the problem, because it had to complete the task, while having a reasonable time and space complexity. For making the right choice, I started by thinking of all the possible solutions and writing a basic approach for each one, until I have found the most efficient one. By using depth first search and backtracking, the algorithm became faster than using random search or nested for loops, since it did not need to search or verify all the positions which were already checked and were not part of the solution.

Thus, the project supports a user-friendly data entry, that can be modified for a wide range of input sizes, based on which it provides the expected output with the number of solutions obtained and all the possible configurations for that specific size.

## 8 References

- [1] Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach. Prentice Hall, 3rd Edition, 2010.
- [2] LATEXproject site, <http://latex-project.org/>, accessed in April 2013.
- [3] David Poole, Alan Mackworth, Artificial Intelligence: Foundations of Computational Agents (2nd ed.), Cambridge University Press, 2017.
- [4] Vaughan, N. (2015, July). Swapping algorithm and meta-heuristic solutions for combinatorial optimization. In Science and Information Conference (SAI).