



**Universitatea din Craiova**  
**Facultatea de Automatică, Calculatoare**  
**și Electronică**

4 Iunie 2021



Student : Nedianu Gabriel-Cătălin  
Specializarea: Calculatoare Română  
Anul II  
Grupa: CR 2.2 B

## Cuprins

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Enunțul Problemei</b>                                    | <b>3</b>  |
| 1.1      | Formularea Detaliată a Problemei . . . . .                  | 3         |
| 1.2      | Alegerea algoritmului de căutare și al limbajului . . . . . | 3         |
| <b>2</b> | <b>Algoritmii</b>   | <b>4</b>  |
| 2.1      | Algoritmul DFS . . . . .                                    | 4         |
| 2.2      | Algoritmul de verificare . . . . .                          | 6         |
| <b>3</b> | <b>Date Experimentale</b>                                   | <b>7</b>  |
| <b>4</b> | <b>Proiectarea aplicației</b>                               | <b>8</b>  |
| 4.1      | Structura de nivel înalt a aplicației . . . . .             | 8         |
| 4.2      | Descrierea Mulțimii Datelor de intrare . . . . .            | 9         |
| 4.3      | Descrierea Mulțimii Datelor de ieșire . . . . .             | 9         |
| 4.4      | Lista modulelor aplicației . . . . .                        | 10        |
| 4.5      | Funcțiile aplicației . . . . .                              | 10        |
| 4.5.1    | main.py . . . . .   | 10        |
| 4.5.2    | functii_arcasi.py . . . . .                                 | 10        |
| 4.5.3    | generator.py . . . . .                                      | 12        |
| <b>5</b> | <b>Rezultate</b>  | <b>13</b> |
| 5.1      | Organizarea datelor de ieșire . . . . .                     | 14        |
| 5.2      | Testarea datelor de ieșire . . . . .                        | 14        |
| 5.3      | Timpii de execuție pentru date mici si mari . . . . .       | 15        |
| 5.4      | Observații si Concluzii . . . . .                           | 15        |

# 1 Enunțul Problemei

## *Problema Arcașilor*

Let us suppose the  $k \times k$  grid presented. 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.

## 1.1 Formularea Detaliată a Problemei

Problema este una de **satisfacere a constrângerilor** prin care trebuie să fie poziționați pe un grid un număr de arcași.

Constrângerile problemei sunt:

- arcașii așezați nu trebuie să se poată nimeri între ei
- arcașii pot să tragă până la o distanță predefinită ( $w$ )
- există anumite ziduri poziționate pe grid prin care săgețile nu pot să treacă

Starea inițială a problemei va fi grid-ul fără arcași și cu zidurile poziționate pe el și un vector cu pozițiile arcașilor de pe grid inițial cu valori nule.

Starea finală a problemei este grid-ul ce are pe el poziționați toți arcașii astfel încât aceștia nu se pot ataca.

## 1.2 Alegerea algoritmului de căutare și al limbajului

Algoritmul ales de mine este un **DFS** euristic, l-am ales pe acesta deoarece are o funcționalitate destul de bună și este destul de rapid, dar are și anumite minusuri pe care le voi discuta în secțiunea de concluzii.

Stările sunt reprezentate de faptul că un arcaș a fost poziționat cu succes, iar următoarea stare este validă numai atunci când poziționarea următorului arcaș nu aduce probleme.

Pentru ca o stare să fie validă aceasta trebuie să îndeplinească condițiile DFS-ului meu: arcașul poziționat să nu îi încurce cu nimic pe restul (mai multe detalii se vor prezenta în cadrul funcțiilor dar și a algoritmului)

Limbajul unde am ales să implementez problema este python deoarece pentru o astfel de problema este mult mai ușor de implementat, sunt mult mai ușor de înțeles acțiunile din spate (față de o implementare în Java sau alte limbaje) și pentru că în cadrul laboratorului acest limbaj a fost aprofundat mai mult pentru acest tip de probleme.

## 2 Algoritmii

### 2.1 Algoritmul DFS

Algoritmul DFS reprezintă algoritmul principal, recursiv, care găsește pozițiile arcașilor și la final printează grid-ul în fișierul primit ca argument.

Pseudocoul lui este următorul:

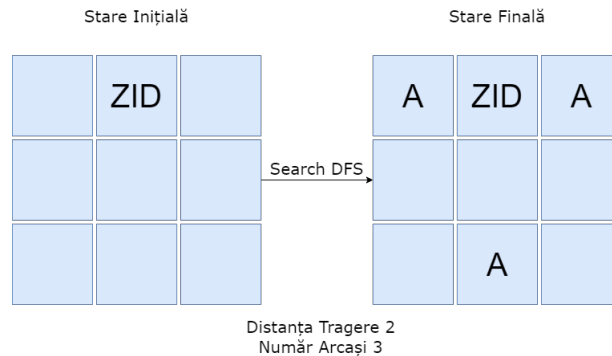
**funcția** SEARCH-DFS(*variabile*) parcurge grid-ul și salvează pozițiile arcașilor  
**variabile:** *arc\_pos*, pozițiile pe coloane unde sunt așezați arcașii, inițial nule  
*table*, grid-ul nostru unde sunt așezate inițial doar zidurile ( $N \times N$ )  
*ok*, devine 1 atunci când toți arcașii sunt așezați corespunzător  
*correct*, devine 1 dacă ultimul arcaș este așezat corect, ajută la trecerea spre următorul nivel  
*index*, variabilă utilizată pentru parcurgerea în adâncime și pentru a avea în vedere la al câtelea arcaș de poziționat s-a ajuns

1. **dacă:** s-au poziționat toți arcașii pe grid și nu este vreo problemă
2.     **afișează** grid-ul cu arcași în fișierul destinat
3.      $ok \leftarrow 1$
4. **dacă:** ultimul arcaș este poziționat corect și nu s-au găsit toți arcașii
5.     **for** line in range(N)
6.     SEARCH-DFS(*următorul\_nivel*)

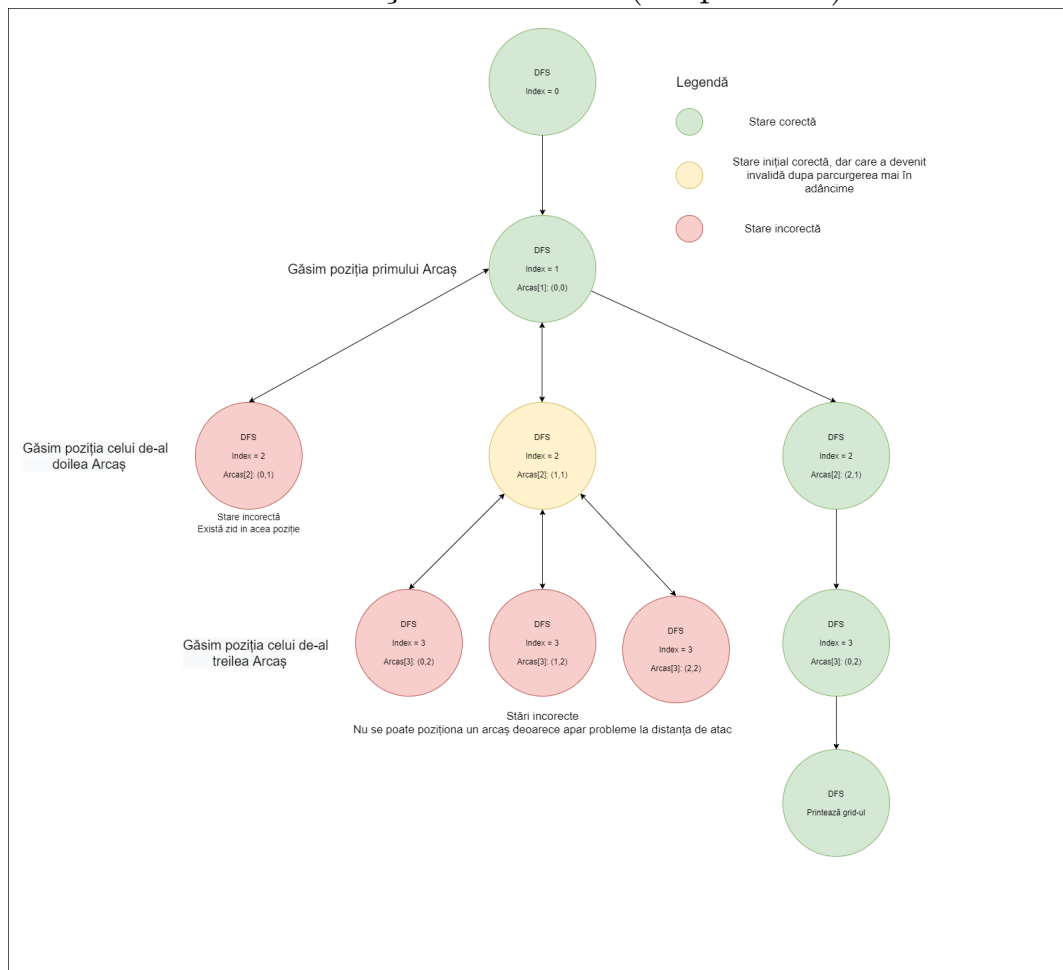
În prima parte, algoritmul verifică, în cazul în care au fost așezați toți arcașii, dacă toți aceștia sunt așezați corect, fără a se putea lovi ( algoritmul de verificare va fi explicat în subsecțiunea următoare. ) apoi dacă este totul corespunzător afișează în fișier grid-ul final și oprește recursivitatea DFS-ului.

În a doua parte, algoritmul verifică ( în cazul în care încă nu au fost așezați toți arcașii ) dacă ultimul arcaș a fost poziționat corespunzător, iar în cazul favorabil, începe căutarea următorului arcaș în adâncimea grid-ului.

Pentru exemplificare, am introdus schematic recursivitatea în cazul unui grid  $3 \times 3$  cu un zid poziționat în punctul (0,1).



## Schema recursivității cazului 3x3 (simplificată)



## 2.2 Algoritm de verificare

În această subsecțiune voi prezenta pe scurt funcționalitatea algoritmului ce verifică în cadrul fiecărei stări validitatea acestuia.

**funcția** de VERIFICARE(*index*) verifică dacă poziționarea ultimului arcaș este în regulă

1. **dacă** ( *check\_last(index) == True* )
2.       *correct*  $\leftarrow$  1
3. **altfel**
4.       *correct*  $\leftarrow$  0
5. **dacă** ( *check(index) == True* și *correct == 1* )
6.       *correct*  $\leftarrow$  1
7. **altfel**
8.       *correct*  $\leftarrow$  0
9. **dacă** ( *correct == 1* )
10.       **dacă** ( *pe pozitia arcasului din grid == Z* )
11.           *correct*  $\leftarrow$  0
12. **dacă** ( *ok == 1* și *correct == 1* )
13.       continue DFS search

Acest algoritm este prezentat pe scurt, el fiind folosit ca o procedură prin care se verifică dacă ultimul arcaș poziționat pe grid este așezat fără a crea probleme.

Mai întâi se compară dacă ultimul arcaș este o amenințare pentru penultimul așezat ( se folosește funcția **threatens**(coord. 2 arcași) care returnează TRUE când este o problema și FALSE când totul este în regulă la compararea pozițiilor a doi arcași).

Apoi se compară dacă ultimul arcaș este o amenințare pentru oricare dintre ceilalți arcași. ( se intră în acest pas doar dacă nu s-au găsit probleme la pasul anterior).

Am ales să compar mai întâi cu penultimul deoarece foarte multe cazuri au doar această amenințare și s-ar pierde mult timp comparându-i mai întâi pe restul arcașilor (experimentând, este cu aproximativ 20% mai rapid algoritmul astfel pe cazurile mari).

Apoi se verifică dacă arcașul ce urmează să fie așezat va fi poziționat pe un zid.

Dacă nu este descoperită nici o neregulă, algoritmul principal poate avansa în adâncime.

### 3 Date Experimentale

Ca date de intrare, problema are nevoie de :

- Numărul de linii și coloane ale grid-ului :  $N \times N$
- Numărul de arcași
- Distanța până la care poate să tragă un arcaș
- Numărul de ziduri
- Coordonatele fiecărui zid ce este amplasat pe grid

Toate datele sunt generate cu un generator automat aleator, aceste sunt împărțite în trei categorii: date mici, medii și mari și foarte mari.

Totuși pentru numrele foarte mari, am limitat la maxim o matrice de aproximativ  $600 \times 600$

Funcția generatoare de teste a fost rulată și aceasta a generat 10 fișiere cu formatul: *test{i}.txt* în folderul *Fisiere\_Input*.

Generatorul funcționează astfel: mai întâi generează numărul de coloane al grid-ului ( $N$ ), apoi generează numărul de arcași, după aceea este generată distanța până la care un arcaș poate să tragă ( $w$ ), iar la final sunt generate numărul de ziduri și coordonatele acestora ( am considerat că maxim un zid va fi plasat pe o coloană, toate variabilele generate au limitări în funcție de  $N$  ) .

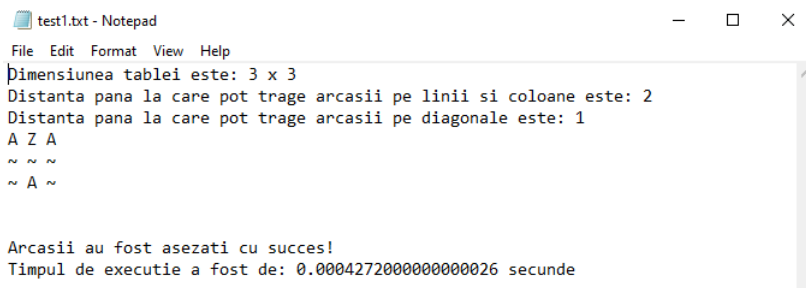
Primele 4 teste generate sunt mici, următoarele 4 sunt medii și mari, iar ultimele 2 sunt foarte mari.

Mai multe explicații legate de acest generator vor fi furnizate în Secțiunea dedicată funcțiilor aplicației.

De asemenea, fișierul de output este și el aranjat într-un mod plăcut și intuitiv (sunt precizate în interiorul lui: dimensiunea, distanța de atac în linie dreaptă și în diagonală, grid-ul final și timpul de execuție al rulării).

Diagonalele sunt mai mici decât liniile, așa că am adăugat o limitare matematică (distanța de tragere pe diagonale este mai mică decât pe linii și coloane și este calculată cu o funcție, dar mai mare ca 0 dacă și  $w$  este mai mare ca 0).

Atașez mai jos un exemplu de fișier de output pentru exemplul recursivității prezentat mai sus.



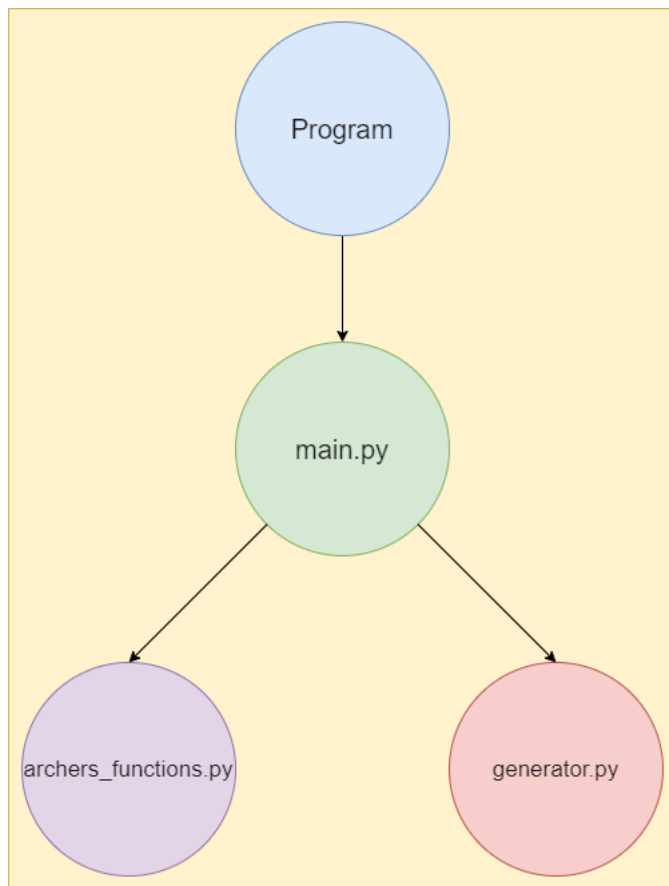
```
test1.txt - Notepad
File Edit Format View Help
Dimensiunea tablei este: 3 x 3
Distanța pana la care pot trage arcașii pe linii si coloane este: 2
Distanța pana la care pot trage arcașii pe diagonale este: 1
A Z A
~ ~ ~
~ A ~

Arcasii au fost asezati cu succes!
Timpul de executie a fost de: 0.000427200000000026 secunde
```

## 4 Proiectarea aplicației

### 4.1 Structura de nivel înalt a aplicației

Aplicația conține următoarele fișiere în limbajul Python: `main.py`, `functii_arcași.py` și `generator.py` conectate astfel:



În `main` a fost folosită mai întâi funcția `generează()` din `generator.py` prin care am generat testele.

Apoi am folosit funcția `startTesting()` care execută fiecare fișier de test și creează unul nou în folderul *FisiereOutput*.

Există și o funcție `startCustomTest()` ce poate fi utilizată pentru a rula un test custom, fără ziduri, cu datele introduse ca argument al funcției (dimensiune grid și distanța până la care arcașii pot să tragă).



## 4.2 Descrierea Mulțimii Datelor de intrare

Datele de intrare pentru algoritm sunt următoarele:

- dimensiune grid
- număr arcași
- distanța de atac
- numărul zidurilor și pozițiile acestora

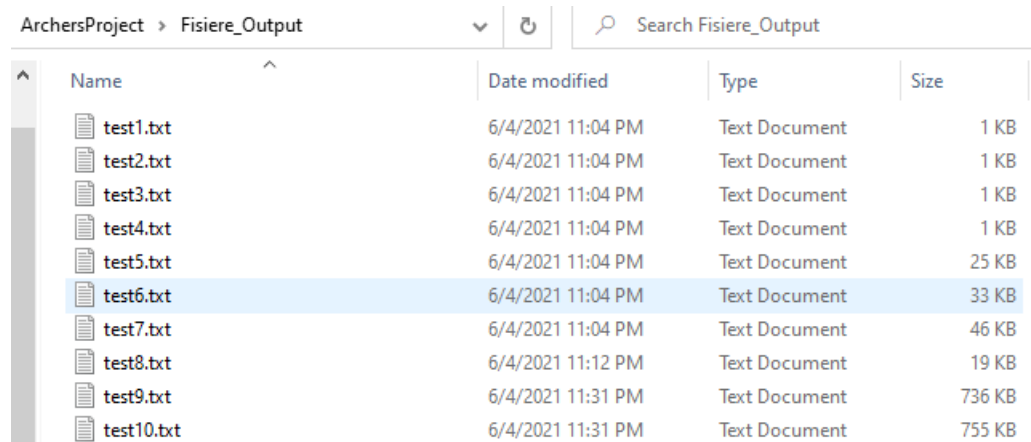
Toate aceste date sunt de tipul integer și sunt salvate într-un folder special numit `Fisiere_Input`.

## 4.3 Descrierea Mulțimii Datelor de ieșire

Datele de ieșire ale problemei vor fi stocate în mod implicit într-un folder numit `Fisiere_Output` (în timpul testelor, aceste date vor fi stocate în fișiere de tipu *test.i*, *i* fiind cuprins între 1 și 10).

Fișierul de ieșire va cuprinde pe primele rânduri mesajele inițiale, care vor arăta variabilele testului curent, pe următoarele *N* rânduri va fi afișat grid-ul cu arcașii poziționați pe el, pe penultimul rând va fi un mesaj ce va arăta dacă arcașii au fost poziționați cu succes, iar pe ultimul rând va fi afișat timpul de execuție (în secunde).

Mai jos am încărcat o fotografie cu folder-ul de Output.



| ArchersProject > Fisiere_Output |                   |               |        |
|---------------------------------|-------------------|---------------|--------|
| Search Fisiere_Output           |                   |               |        |
| Name                            | Date modified     | Type          | Size   |
| test1.txt                       | 6/4/2021 11:04 PM | Text Document | 1 KB   |
| test2.txt                       | 6/4/2021 11:04 PM | Text Document | 1 KB   |
| test3.txt                       | 6/4/2021 11:04 PM | Text Document | 1 KB   |
| test4.txt                       | 6/4/2021 11:04 PM | Text Document | 1 KB   |
| test5.txt                       | 6/4/2021 11:04 PM | Text Document | 25 KB  |
| test6.txt                       | 6/4/2021 11:04 PM | Text Document | 33 KB  |
| test7.txt                       | 6/4/2021 11:04 PM | Text Document | 46 KB  |
| test8.txt                       | 6/4/2021 11:12 PM | Text Document | 19 KB  |
| test9.txt                       | 6/4/2021 11:31 PM | Text Document | 736 KB |
| test10.txt                      | 6/4/2021 11:31 PM | Text Document | 755 KB |

## 4.4 Lista modulelor aplicației

Pentru problema , modulele sunt:

- `main.py` -modulul principal unde sunt apelate funcțiile specifice
- `functii_arcasti.py` -modulul în care sunt scrise majoritatea funcțiilor utilizate
- `generator.py` -modulul care conține funcțiile de generare de input

## 4.5 Funcțiile aplicației

### 4.5.1 `main.py`

Acesta este modulul principal al programului în Python, pentru algoritmul creat, aici se pot apela funcții specifice: de generare de cazuri (*genereaza()*), de executare a cazurilor (*startTesting()*) și de executare a unui test custom (*startCustomTest()*).

Funcțiile și import-urile sunt comentate, acestea pot fi decommentate în momentul în care se dorește reutilizarea uneia dintre ele.

În acest modul sunt astfel conținute doar funcțiile principale, restul fiind implementate în interiorul celorlalte module .

### 4.5.2 `functii_arcasti.py`

Acest modul conține majoritatea funcțiilor utilizate de algoritmul principal, *searchDFS(index, correct, nr\_test)*, algoritmul prezentat în secțiunea de algoritmi, el primește ca argumente *index*-ul arcașului la care s-a ajuns (al câtelea), variabila *correct* (variabilă care transmite faptul că o stare a fost corectă sau incorectă) și *nr\_test* variabilă care este utilizată pentru a stoca fișierul în care se va printa grid-ul (în formatul prezentat: A → arcaș, Z → zid, *tilda* → poziție goală).

Am folosit câteva variabile globale pentru a salva argumentele generale și a le utiliza în recursivitatea DFS.

- *N* dimensiunea grid-ului
- *table* matricea grid ce este inițializată în faza de load a fiecărui test
- *arc\_pos* pozițiile arcașilor
- *ok* (este 0 predefinit, devine 1 după ce găsim soluția)
- *w* distanța de tragere a arcașilor

Urmează prezentarea celorlalte funcții aflate în acest modul.

Funcțiile *check\_last(index)* și *check(index)* sunt funcțiile ce verifică arcașii introdusi dacă au o poziționare fără amenințări pentru ceilalți, acestea primesc ca argument indicele arcașului la care s-a ajuns.

Funcția `check_last(index)` îl compară pe ultimul arcaș introdus cu penultimul cu ajutorul funcției `threatens()` și returnează FALSE dacă este găsită vreo problemă sau TRUE dacă totul este în regulă.

Funcția `check(index)` funcționează asemănător, doar ca aceasta îi compară pe toți, valorile returnate sunt la fel.

Funcția `threatens(coordonate)` primește ca argument linia și coloana a doi arcași și verifică dacă aceștia se pot lovi, returnează TRUE dacă sunt probleme cu poziționarea acestora sau FALSE dacă nu. În prima parte verifică dacă aceștia sunt poziționați pe aceeași linie, coloana sau diagonală, iar dacă sunt, apar 3 cazuri de verificare:

- I Dacă sunt pe aceeași linie, se verifică prima oară dacă cumva distanța dintre ei este mai mare ca  $w$  (atunci returnează FALSE) apoi se verifică dacă este un zid între ei cu funcția `is_wall_linie()` (se returnează tot FALSE).
- II Dacă sunt pe aceeași coloană, se verifică prima oară dacă cumva distanța dintre ei este mai mare ca  $w$  (atunci returnează FALSE) apoi se verifică dacă este un zid între ei cu funcția `is_wall_col()` (se returnează tot FALSE).
- III Dacă sunt pe aceeași diagonală, se verifică prima oară dacă cumva distanța dintre ei este mai mare ca valoarea de tragere pe diagonală (atunci returnează FALSE) apoi se verifică dacă este un zid între ei cu funcția `is_wall_diag()` (se returnează tot FALSE).

Dacă nu se returnează FALSE până aici înseamnă că este o problemă și se returnează astfel TRUE, iar dacă nu s-a intrat deloc în această condiție logică returnăm automat FALSE (nu este nicio amenințare).

Funcțiile `is_wall_linie()`, `is_wall_col()`, `is_wall_diag()` funcționează asemănător, acestea primesc în general argumente legate de poziția celor 2 arcași pe grid și caută dacă între cei doi există un zid, se returnează TRUE dacă se găsește unul.

Funcția `is_wall_linie()` primește coloanele celor 2 arcași și linia pe care ei se află și iterează prin pozițiile dintre cei doi, `is_wall_col()` primește liniile celor 2 arcași și coloana pe care ei se află și iterează prin pozițiile dintre cei doi, iar `is_wall_diag()` primește coordonatele pe care ei se află și iterează prin pozițiile pe diagonală dintre cei doi (funcție care funcționează oricum ar fi plasați ei)

Funcția `load_test(nr_test)` este folosită pentru a încărca din fișierul dat de variabila primită toate variabilele globale și pentru a inițializa matricile necesare, se parcurge linie cu linie fișierul și se salvează totul predefinit, apoi se așază pozițiile zidurilor pe grid.

Funcția `load_custom_test(variabile)` este folosită pentru a încărca un test custom ce primește argumentele specifice testului și funcționează asemănător cu cea dinainte.

Funcția `startTesting()` este folosită pentru a rula rând pe rând toate testele, aceasta iterează de 10 ori și fiecare iterație este alcătuită din următorii pași: încărcarea testului, rularea funcției principale de căutare (se măsoară **timpul de execuție** ) apoi se scrie în fișierul curent (în funcție de testul la care s-a ajuns) timpul în care s-a realizat execuția.

Funcția `startCustomTest()` este funcția folosită pentru a rula un test custom, funcționează asemănător cu cea de mai sus, doar că execuția se realizează doar o dată, iar variabilele sunt primite ca argument.

Ultimele funcții sunt cele utilizate pentru a afișa grid-ul, funcția `print(nr_test)` salvează în matricea `table` arcașii apoi apelează funcția `printTable(nr_test)` care primește ca argument numărul fișierului de test în care se vor afișa primele linii corespunzătoare formatului fișierului de output, apoi se va afișa grid-ul și un mesaj corespunzător dacă testul a fost finalizat cu succes). Funcțiile acestea sunt apelate în cadrul funcției implementării algoritmului de search dacă totul a fost în regulă.

### 4.5.3 generator.py

În acest modul avem implementarea algoritmului care generează testele noastre cu ajutorul a trei funcții.

Funcția `generator(nr_test, dimensiune_start, dimensiune_stop)` generează variabilele pentru numărul testului primit, variabilele dimensiune start și stop sunt utilizate pentru a primi dimensiunile între care este generată dimensiunea  $N$  a grid-ului (tot ele vor ajuta la crearea tipurilor de teste). Mai întâi generăm dimensiunea grid-ului random, apoi numărul de arcași între  $N$  și  $2 \times N$ , apoi generăm random punctul de tragere și numărul de ziduri. La final se apelează funcția `generator(nr_ziduri, dimensiune, test)`.

Funcția `generator(nr_ziduri, dimensiune, test)` este funcția ce generează în fișierul `test` coordonatele (linie, coloană) pentru un număr de ziduri dat de variabila `nr_ziduri`, am considerat să fie generate aleatoriu, maxim câte unul pe o coloană. Funcția generează zidurile între 0 și dimensiunea maximă a grid-ului, este ajutată de un vector de variabile boolene pentru a vedea pe ce coloana s-a generat un zid și pe care nu.

Ultima funcție, `genereaza()` este funcția care prin 10 iterații ne generează teste. Acesta va genera complexitatea testelor în principal în funcție de dimensiunea grid-ului. Am adăugat dimensiunile start și stop în funcție de iterația curentă:

- grid-ul pentru primele 4 cazuri este generat între  $2 \times i$  și  $4 \times i$
- grid-ul pentru următoarele 4 cazuri este generat între  $10 \times i$  și  $40 \times i$
- grid-ul pentru primele cazuri este generat între  $40 \times i$  și  $70 \times i$

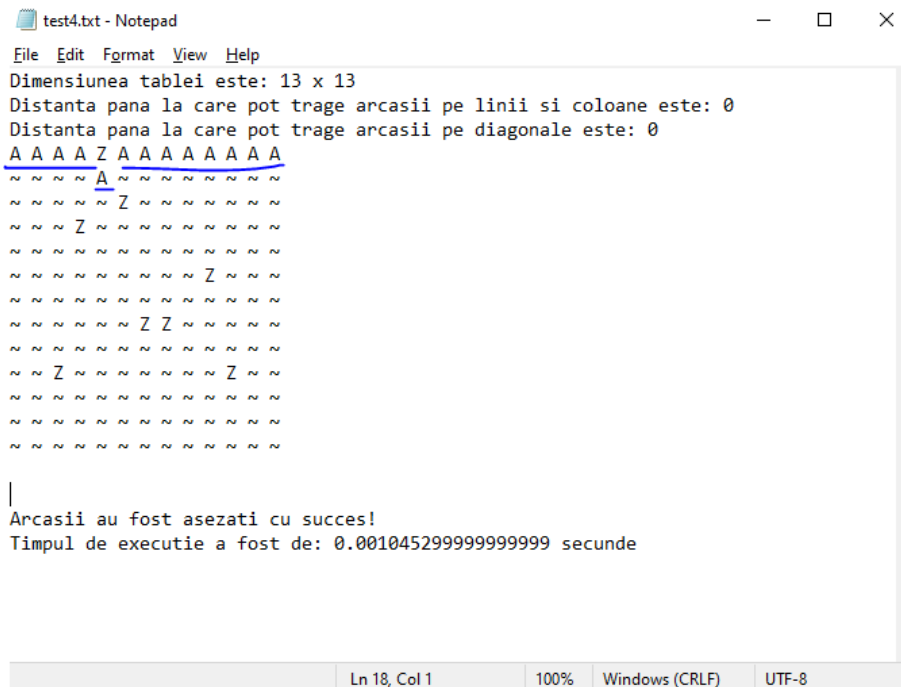
Valorile generate sunt salvate în folderul `Fișiere_Input`.

## 5 Rezultate

Aici voi arăta rezultatele testelor și anumite observații pe baza implementărilor și ale executării codului în timpul testelor.

O prima observație ar fi că algoritmul este foarte rapid pentru cazurile mici, medii și mari care au distanța de tragere relativ mică (sub 30% din dimensiunea grid-ului), dar având în vedere că este un algoritm de tip DFS, acesta devine foarte lent pentru cazurile mari care au o distanță de tragere mare deoarece se verifică foarte multe stări care sunt incorecte și se întoarce de foarte multe ori.

Cu toate acestea, este un algoritm foarte ușor de înțeles, parcurgerea grid-ului în adâncime și poziționarea arcașilor le-am exemplificat în testul numărul 4 în care sunt așezați 13 arcași pe un grid de  $13 \times 13$  în care distanța de tragere este.



```
test4.txt - Notepad
File Edit Format View Help
Dimensiunea tablei este: 13 x 13
Distanța pana la care pot trage arcasii pe linii si coloane este: 0
Distanța pana la care pot trage arcasii pe diagonale este: 0
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A
A A A A Z A A A A A A A A

Arcasii au fost asezati cu succes!
Timpul de executie a fost de: 0.00104529999999999999 secunde
```

În acest exemplu, se parcurge grid-ul în adâncime, primii 4 arcași sunt poziționați fără probleme, apoi se întâlnește un zid unde nu poate fi plasat un arcaș și se coboară pe următorul nivel, apoi restul de 8 arcași sunt poziționați în continuare fără probleme.

Se observă și că algoritmul în cazul acesta este foarte rapid.

Un exemplu contra este testul numărul 10 în care pentru a poziționa pe un grid de  $621 \times 621$  cu distanța de tragere 100 timpul de execuție este de aproximativ 1000 de secunde, algoritmul fiind foarte lent aici.

## 5.1 Organizarea datelor de iesire

Pentru vizualizarea testelor, rezultate sunt afișate în fișierele de tip `test%i.txt`,  $i \in \{1, 10\}$ , aflate în folderul *Fisiere\_Output*.

Toate fișierele au același format, pe care îl voi mai explica o dată aici, și voi explica și de ce am ales să fie salvate astfel.

Prima linie dintr-un fișier de output arată dimensiunea  $N$  a grid-ului pentru care a fost executat programul.

A doua linie arată distanța de trage a arcașilor pe linii și coloane pentru care a fost executat programul.

A treia linie arată distanța de trage a arcașilor pe diagonale pentru care a fost executat programul.

În următoarele  $N$  linii este afișat grid-ul cu arcașii poziționați pe el.

Penultima linie arată dacă arcașii au fost așezați cu succes.

Ultima linie prezintă timpul de execuție.

Primele și ultima linie sunt importante deoarece se vor folosi pentru a prezenta într-un tabel rezultatele.

Observație: fișierele sunt create dacă nu există deja, sau suprascrise dacă există, pentru a nu avea scrieri adăugate la final în cazul rulării testelor iar.

Am ales acest format pentru a putea fi organizate cât mai frumos datele și pentru a putea fi înțelese cât mai ușor. Exemple cu poze cu fișierele de ieșire au mai fost prezentate de-a lungul raportului.

## 5.2 Testarea datelor de ieșire

Pentru a testa dacă algoritmul salvează poziționările corect în fișierele de output, pe lângă apelarea funcției `check()` am și citit anumite fișiere pentru a vedea manual poziționarea (pentru cazurile mici).

Acest lucru m-a ajutat de exemplu în cazul în care nu era completă funcția de verificare, iar unii arcași puteau fi așezați deasupra zidului (am rezolvat acest bug ulterior).

Datele de ieșire s-au dovedit a fi corecte, astfel chiar și lent, algoritmul a fost corect.

### 5.3 Timpii de execuție pentru date mici si mari

Am organizat datele pentru timpii de execuție într-un tabel care conține toate datele necesare pentru o analiză a soluției.

Tabel Comparare Teste

| Nr.Test | Dimensiune grid | Distanța tragere | Timp Execuție |
|---------|-----------------|------------------|---------------|
| 1       | 3               | 2                | 0.0004272     |
| 2       | 8               | 6                | 0.0009306     |
| 3       | 11              | 5                | 0.0009711     |
| 4       | 13              | 0                | 0.0010452     |
| 5       | 111             | 64               | 4.132687      |
| 6       | 128             | 124              | 14.1135649    |
| 7       | 152             | 14               | 2.3782        |
| 8       | 97              | 39               | 1.57475089    |
| 9       | 613             | 50               | 458.9768847   |
| 10      | 621             | 102              | 1185.7126315  |

### 5.4 Observații si Concluzii

Observăm că pentru primele 8 teste timpul de execuție este foarte bun, dar pentru ultimele 2 timpul a crescut exponențial. Deoarece numărul arcașilor este în general apropiat de dimensiunea grid-ului, observăm că distanța de tragere cauzează o creștere a timpului de execuție aproximativ liniară, pe când creșterea grid-ului una exponențială.

Acest lucru se explică deoarece cu creșterea dimensiunii grid-ului este crescut și numărul de teste posibile exponențial, creșterea distanței cauzând doar o creștere liniară a acestora, dar este crescut și numărul recursivităților foarte mult, ceea ce cauzează o încetinire destul de mare.

O altă observație interesantă este că pentru griduri asemănătoare ( testele 9 & 10 sau testele 5 & 6 & 7), timpul de execuție a crescut aproximativ liniar în funcție de raportul dintre distanțele de tragere, este doar aproximativ deoarece există ziduri poziționate ce pot ajuta plasarea arcașilor mult mai rapid. Aceasta este una dintre limitările unui algoritm de tip DFS (timpul de execuție).

Acest proiect a fost foarte util în învățarea multor lucruri noi și în aprofundarea celor deja știute, iar implementările în limbajul Python au fost foarte utile deoarece m-au ajutat la avansare cunoștințelor.

O parte provocatoare în implementarea acestui algoritm din cadrul temei de casă a fost determinarea matematicii din spatele verificărilor si amplasării arcașilor, multe dintre aceste funcții determinându-le scriind si analizând pe o tablă, lucru ce s-a dovedit foarte util în înțelegerea logicii din spatele algoritmului.

Unele achievement-uri realizate cu acest proiect, pe lângă aprofundarea cunoștințelor, înțelegerea a multor concepte din spatele inteligenței artificiale, implementarea unei probleme de satisfacere a constrângerilor, implementarea unui generator complex si bine structurat ce ar putea fi folosit si pe viitor la anumite proiecte, dar și multe altele.

Proiectul ar putea fi dezvoltat si utilizat ca partea de inteligență artificială în cadrul unor jocuri de tip Tower Defense, pentru amplasarea anumitor structuri, poate în viitor voi implementa acest lucru experimental.



## Referințe

- [1] Peter Norvig and Stuart J. Russell, *Artificial Intelligence: A Modern Approach*. Pearson Education Limited, 3rd Edition, 2016.
- [2] [https://www.overleaf.com/learn/latex/Learn\\_LaTeX\\_in\\_30\\_minutes](https://www.overleaf.com/learn/latex/Learn_LaTeX_in_30_minutes)
- [3] <https://www.latex-project.org/>
- [4] [https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-9863-7\\_875](https://link.springer.com/referenceworkentry/10.1007%2F978-1-4419-9863-7_875)
- [5] [https://en.wikipedia.org/wiki/Depth-first\\_search](https://en.wikipedia.org/wiki/Depth-first_search)
- [6] <http://personales.upv.es/misagre/papers/sara07.pdf>
- [7] <https://www.reddit.com/r/Python/>