

UNIVERSITATEA DIN BUCUREȘTI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

Metode Moderne de Calcul și Simulare
Simularea unui joc de Sudoku pe nivele de dificultate

Studenti - Grupa 451

Ivan Laurențiu-Marian
Voicu Andreea Ana-Maria

Anul universitar 2020-2021

I. Tehnologiile proiectului

- Proiectul este scris în IDE-ul *PyCharm* folosind limbajul de programare *Python*, în versiunea 3.7.9 (poate fi rulat pe orice versiune mai mare sau egala cu 3.0.0). Interfața proiectului a fost realizată folosind librăria *Turtle*.
- Pentru a lansa simularea rulez fișierul *main.py*, în care sunt importate fișierele corespondente, *self_solver.py* și *board.py*.

II. Descrierea proiectului

Proiectul simulează jocul Sudoku pe trei nivele de dificultate: „Easy”, “Medium” și „Hard”. În funcție de nivelul ales de utilizator, se generează o grilă parțial completată cu cifre aleatoare, pe care o va rezolva calculatorul. Durata unui joc și numărul de mutări cresc odată cu creșterea nivelului de dificultate. Calculatorul va atinge obiectivul Sudoku și anume, acela de a umple o grilă 9x9 cu cifre astfel încât fiecare coloană, fiecare rând și fiecare dintre cele nouă subgrile 3x3 care alcătuiesc grila să conțină toate cifrele din 1 la 9.

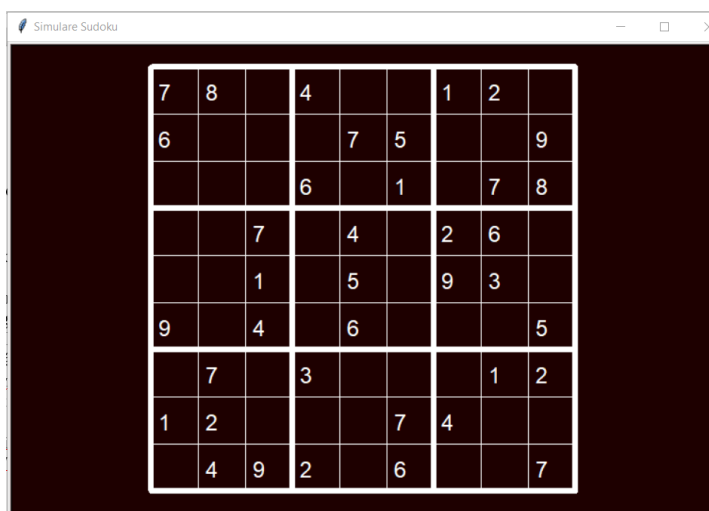
Proiectul se bazează pe un algoritm de backtracking pentru a investiga toate soluțiile posibile ale unei grile date. Având grila inițială generată, testează toate căile posibile către o soluție până când se găsește una. De fiecare dată când o cale este testată, dacă valoarea nu respectă regulile jocului, algoritmul revine la elementul anterior pentru a testa o altă cale posibilă și așa mai departe, până când găsește o soluție sau toate căile sunt testate.

III. Componentele proiectului

Main.py

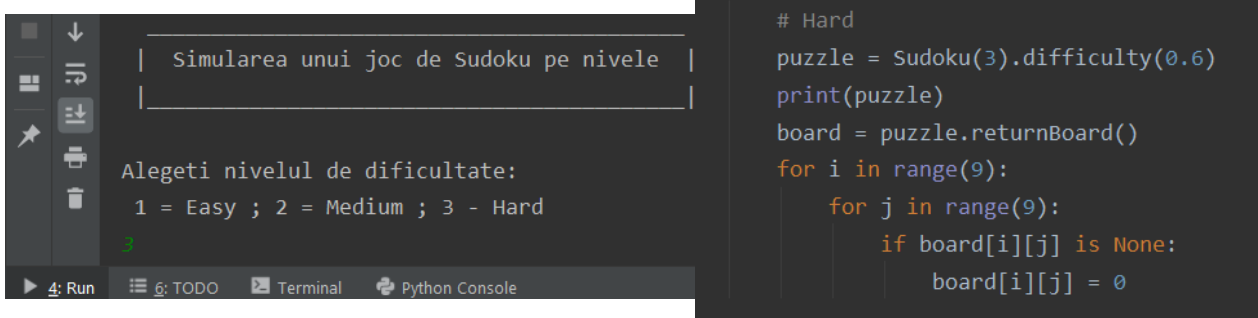
- Din funcția *main()* pornim simularea jocului.

Interfața unui model de tablă inițială :



7	8		4			1	2	
6				7	5			9
			6		1		7	8
		7		4		2	6	
		1		5		9	3	
9		4		6				5
	7		3				1	2
1	2				7	4		
	4	9	2		6			7

- Programul preia de la tastatură cifra corespunzătoare nivelului de dificultate pentru a genera valorile inițiale ale tablei (1/2/3). Generarea cifrelor aleatoare depinde de un coeficient de dificultate setat, din funcția **Sudoku()**, astfel încât nivelele să fie diferențiate, iar algoritmul să fie mai solicitat odată cu creșterea nivelului.

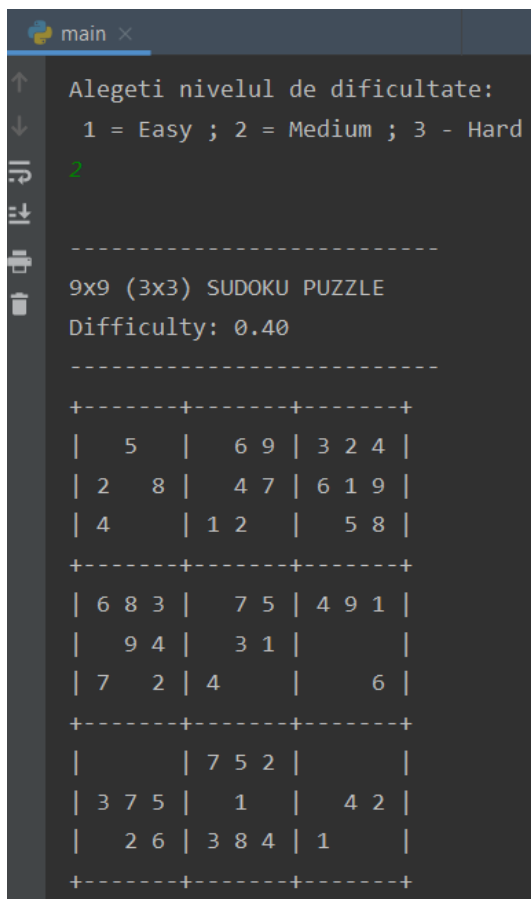


```

# Hard
puzzle = Sudoku(3).difficulty(0.6)
print(puzzle)
board = puzzle.returnBoard()
for i in range(9):
    for j in range(9):
        if board[i][j] is None:
            board[i][j] = 0

```

- Valorile inițiale ale grilei sunt generate aleator și sunt diferite la fiecare simulare a jocului, chiar dacă nivelul de dificultate ales este același. De asemenea, acestea sunt afișate în consolă sub forma unui puzzle, la rularea proiectului.

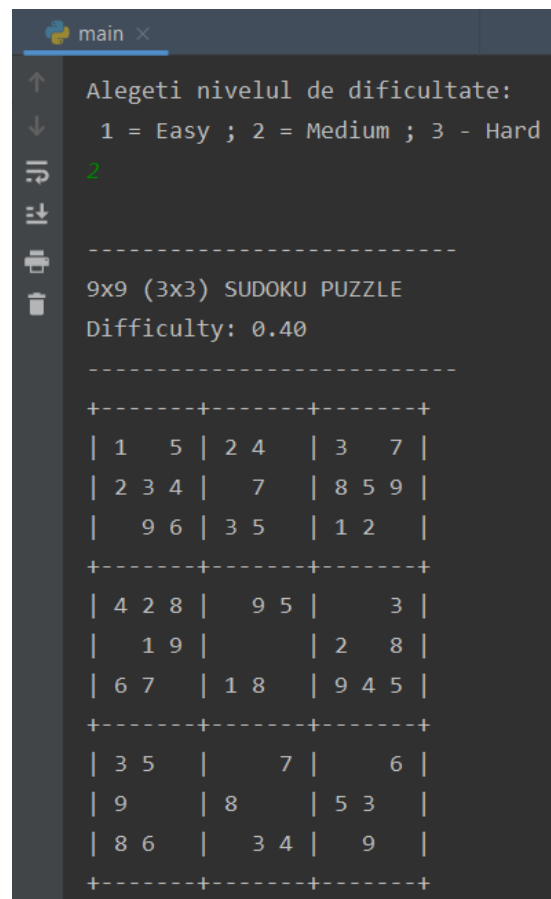


```

Alegeti nivelul de dificultate:
 1 = Easy ; 2 = Medium ; 3 - Hard
2

-----
9x9 (3x3) SUDOKU PUZZLE
Difficulty: 0.40
-----
+-----+-----+-----+
| 5  | 6 9 | 3 2 4 |
| 2  8 | 4 7 | 6 1 9 |
| 4   | 1 2  | 5 8  |
+-----+-----+-----+
| 6 8 3 | 7 5 | 4 9 1 |
| 9 4  | 3 1 |    |
| 7  2 | 4   |    6 |
+-----+-----+-----+
|    | 7 5 2 |    |
| 3 7 5 | 1  | 4 2 |
| 2 6  | 3 8 4 | 1  |
+-----+-----+-----+

```



```

Alegeti nivelul de dificultate:
 1 = Easy ; 2 = Medium ; 3 - Hard
2

-----
9x9 (3x3) SUDOKU PUZZLE
Difficulty: 0.40
-----
+-----+-----+-----+
| 1  5 | 2 4  | 3  7 |
| 2 3 4 | 7   | 8 5 9 |
| 9 6  | 3 5  | 1 2  |
+-----+-----+-----+
| 4 2 8 | 9 5 |    3 |
| 1 9  |    | 2  8 |
| 6 7  | 1 8  | 9 4 5 |
+-----+-----+-----+
| 3 5  |    7 |    6 |
| 9   | 8   | 5 3  |
| 8 6  | 3 4  | 9   |
+-----+-----+-----+

```

- Spațiile goale din schema afișată în consolă corespund celurilor libere din grilă.

- Durata unui joc este calculată cu ajutorul funcției **time()** și este afișată în consolă la finalul fiecărei simulări.

Board.py

- Funcția **drawGrid()** desenează tabla jocului, o **matrice 9x9** de 81 de elemente (**celule**), echivalentă cu o **matrice** de 9 elemente (**subgrilele tablei**) **3x3**. Desenarea grilei pornește de la coordonatele stabilite, **topLeft_x** și **topLeft_y**.

Self_solver.py

- În acest fișier este implementat algoritmul de rezolvare al grilei. Funcția **solveGrid()** încearcă să rezolve, în mod recursiv, tabla cu input-ul aleator generat și verifică toate combinațiile de cifre posibile până găsește o soluție.

Găsește prima celula liberă și, pentru fiecare valoare din intervalul (0,10), verifică dacă valoarea nu se află deja în rândul/coloana în care a fost introdusă. Mai departe, dacă este îndeplinită condiția, algoritmul identifică în care dintre cele 9 „cutii” (subgrile 3x3) se află și verifică dacă valoarea se află deja în pătratul respectiv. În caz afirmativ, valoarea este plasată în celula respectivă, altfel, se verifică următoarea valoare.

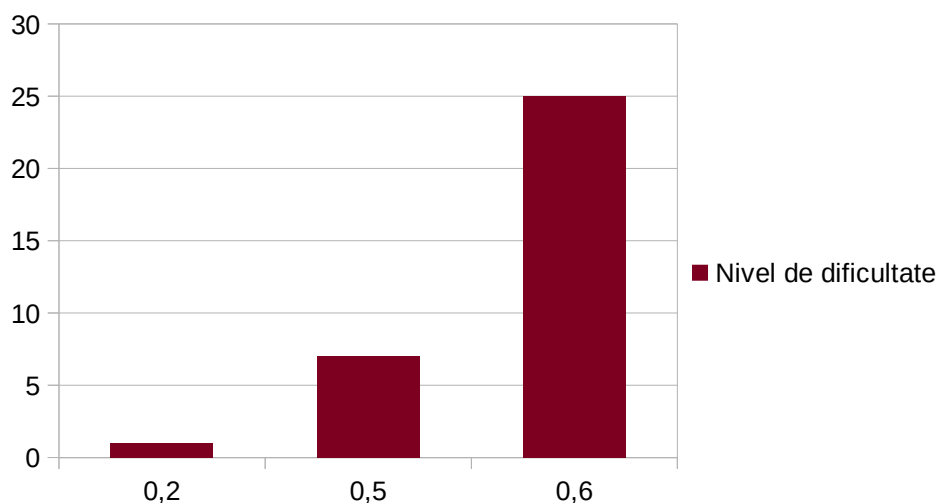
- După ce valoarea a fost găsită, funcția **checkGrid()** verifică dacă grila este completată. Dacă este completată, jocul s-a sfârșit și tabla se afișează, altfel, algoritmul caută următoarea celulă liberă.
- Interfața tabelului este actualizată după fiecare valoare gasită.
- De fiecare data când algoritmul este aplicat, se înregistrează , prin variabila **moves**, numărul total de completări ale calculatorului într-o celulă și , prin variabila **backtracks**, de câte ori calculatorul a recurs la o celulă anterioară pentru a găsi soluția și se afișează în consolă. De asemenea, în consolă se ține evidența fiecărei iterații, afișându-se numărul de completări al fiecărei celule goale, împreună cu rândul și coloana în care aceasta se află și durata până la găsirea soluției complete.

```
main x
↑
↓
Numar de completari in randul 8 - coloana 6 : 2
Numar de completari in randul 8 - coloana 8 : 1
Numar de completari in randul 9 - coloana 2 : 1
Numar de completari in randul 9 - coloana 3 : 1
Numar de completari in randul 9 - coloana 6 : 1
Numar de completari in randul 9 - coloana 8 : 1
Numar de completari in randul 9 - coloana 9 : 1
Gridul este complet si verificat.
Numarul de backtrack-uri realizate: 59

| Sfarsit joc |
Timpul de joc : 9 secunde
```

IV. Backtracking vs. Algebra liniară

Backtracking



- Graficul prezintă evoluția timpului de rezolvare în raport cu gradul de dificultate. Pe axa OX este reprezentat **coeficientul de dificultate** corespunzător fiecărui nivel (Easy – 0,2 ; Medium – 0,4 ; Hard – 0,6), iar pe axa OY este reprezentat **timpul mediu de rulare**. Timpul de rezolvare al primului nivel, de o secunda, rămâne constant, indiferent de generarea cifrelor aleatoare. Timpul pentru un nivel mediu variază între 5 și 10 secunde. Cât despre cel al ultimului nivel, poate ajunge până la 40 de secunde, în funcție de ce cifre sunt generate în tabla inițială.
- Acest algoritm garantează soluția dacă puzzle-ul Sudoku este un puzzle valid.

Sistem liniar

Sudoku este rezolvat ca o problemă de satisfacție a constrângerilor (echivalente cu regulile jocului). Constrângerile Sudoku sunt descrise matematic ca un **sistem liniar** întreg de ecuații care sunt apoi rezolvate prin programare liniară întreagă. Acești algoritmi garantează una sau toate soluțiile puzzle-ului, totuși complexitatea crește exponențial pe măsură ce dimensiunea puzzle-ului tinde să crească.

Formularea problemei

Fie un puzzle sudoku $N \times N$. Conținutul celulei n din S poate fi reprezentat ca $S_n = \{1, 2, \dots, N\}$ pentru $n = \{1, 2, \dots, N^2\}$

Fie $I = [I(S_n = 1), I(S_n = 2), \dots, I(S_n = N)]^T$ este un vector indicator în care $I(S_n = k)$ este o funcție indicator care este dată ca : $I(S_n = k) = \{1, \text{daca } S_n = k$
 $0, \text{altfel.}$

Fie $x = [x_1, x_2, \dots, x_{N^2}]^T$ de mărime N^2 . În plus față de acestea, există câteva indicii date pentru fiecare puzzle care trebuie, de asemenea, satisfăcute. Fiecare dintre constrângerile de mai sus și indicii pot fi scrise ca o combinație liniară la elementele lui x .

Indiciile pot fi, de asemenea, scrise ca o combinație liniară de x . Combinând toate constrângerile putem scrie problema Sudoku în formă mai generică :

$$Ax = \begin{bmatrix} A_{row} \\ A_{col} \\ A_{box} \\ A_{cell} \\ A_{clue} \end{bmatrix} x = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} = b$$

Unde $A_{row}, A_{col}, A_{cell}, A_{clue}$ sunt matricile asociate cu diferite constrângeri. Dimensiunea matricei este $(4N^2 + C) \times N^2$ unde C denotă numărul de indicii.

Algoritm de programare liniară– minimizarea normei $L1$

$A \cdot x = b \quad (1)$ duce la un număr infinit de soluții. Cu toate acestea, nu toate soluțiile sunt soluții sudoku. De fapt, pentru sudoku-ul care are o soluție unică, există o singură soluție care constă doar din 0 și 1. Toate soluțiile fezabile ale (1) trebuie să satisfacă ecuația :

$$\|x\|_0 = \sum_{j=1}^{N^2} \|x_j\|_0 \geq N^2$$

Pentru a găsi cea mai rară soluție a (1) vom lua în considerare următoarea problemă de optimizare:

$$\text{minimize } \|x\|_1 \text{ sub.to } Ax = b \quad (2), \text{ unde}$$

$$\|x\|_1 = \sum_{j=1}^{N^3} |x_j| \text{ este norma 1 a lui } x.$$

Soluția la (2) rezolvă majoritatea puzzle-urilor Sudoku. Ecuația (2) poate fi ușor exprimată ca un model de programare liniară și există un pachet de programare liniară standard pentru a o rezolva.

Concluzii

- Tehnicile în **programarea liniară**, de soluție aproximativă, pentru rezolvarea Sudoku nu au precizie în comparație cu soluțiile exacte, dar câștigul este că complexitatea calculelor pentru rezolvatorii aproximativi este mai bună atunci când dimensiunea puzzle-ului crește. Cu toate acestea, algoritmul bazat pe minimizarea normei L1 are viteză de calcul redusă.
- Spre deosebire de **algoritmul de minimizare**, cel de **backtracking** tinde să ajungă la soluția exactă a problemei. Avantaje ale metodei de backtracking:
 - O soluție este garantată (atâta timp cât puzzle-ul este valid).
 - Algoritmul (și, prin urmare, codul programului) este mai simplu decât alți algoritmi, mai ales în comparație cu algoritmi puternici care asigură o soluție la cele mai dificile puzzle-uri. Așadar, este mai potrivit pentru puzzle-uri dificile.
- Cu toate acestea, dezavantajul metodei de backtracking este că timpul de rezolvare poate fi lent în comparație cu algoritmi modelați după metode deductive. Cu cât solverul face mai multe erori, cu atât trebuie să efectueze mai multe trasee, ceea ce îi scade eficiența generală și îi crește timpul de rulare efectiv.
- În schimb, programarea liniară este un instrument puternic pentru găsirea soluțiilor optime. Cu alte cuvinte, dacă putem scrie o problemă ca:

$$\begin{array}{ll} \text{maximize} & c^T x \\ \text{subject to} & Ax \leq b \\ & x \geq 0 \end{array} \iff \begin{array}{ll} \text{minimize} & b^T y \\ \text{subject to} & A^T y \geq c \\ & y \geq 0 \end{array}$$

(Formulări primare (stânga) și duale (dreapta) ale problemelor de programare liniară - maximizează sau minimizează o funcție obiectivă, supusă constrângerilor de inegalitate unde b, c, x și y sunt vectori și A este o matrice.)

putem găsi parametri optimi x și y.

- Exemplu de rezultate al metodei de backtracking pentru cel mai dificil nivel :

```
main x
Alegeti nivelul de dificultate:
1 = Easy ; 2 = Medium ; 3 - Hard
-----
9x9 (3x3) SUDOKU PUZZLE
Difficulty: 0.60
-----
+-----+-----+-----+
| 8 | 1 | 4 6 |
| 5 9 | 6 4 | 8 7 |
|      |      |      |
+-----+-----+-----+
| 2 7 | 6 |      |
| 5 | 8 7 | 9 |
| 6 | 9 5 | 7 3 |
+-----+-----+-----+
| 1 | 5 3 | 6 |
| 7 | 2 4 1 | 5 |
| 5 |      | 4 |
+-----+-----+-----+

Numar de completari in randul 1 - coloana 1 : 1
Numar de completari in randul 1 - coloana 3 : 1
Numar de completari in randul 1 - coloana 5 : 1
Numar de completari in randul 1 - coloana 6 : 1
Numar de completari in randul 1 - coloana 5 : 2
```

```
Numar de completari in randul 9 - coloana 1 : 1
Numar de completari in randul 9 - coloana 2 : 1
Numar de completari in randul 9 - coloana 4 : 1
Numar de completari in randul 9 - coloana 5 : 1
Numar de completari in randul 9 - coloana 6 : 1
Numar de completari in randul 9 - coloana 7 : 1
Numar de completari in randul 9 - coloana 9 : 1
Gridul este complet si verificat.
Numarul de backtrack-uri realizate: 58

| Sfarsit joc |
Timpul de joc : 29 secunde

Process finished with exit code 0
```

Simulare Sudoku

7	8	3	1	9	2	4	6	5
2	5	9	3	6	4	1	8	7
4	1	6	7	5	8	3	9	2
9	2	7	6	1	3	8	5	4
5	3	1	4	8	7	9	2	6
8	6	4	9	2	5	7	1	3
1	4	2	5	3	9	6	7	8
6	7	8	2	4	1	5	3	9
3	9	5	8	7	6	2	4	1

Sfarsit joc