**Proiect Evaluarea Performanțelor - Realizarea jocului Sudoku în limbajul C#, folosind algoritmul Forward Checking**

Studenți: Florea Alexandra, Grupa 1408A

Pintilie Justinian, Grupa 1408A

**1) Enunțarea temei și motivarea ei**

Tema proiectului este rezolvarea jocului Sudoku în limbajul de programare C#, folosind algoritmul Forward Checking.

Am ales această temă deoarece se pliază foarte bine pe algoritmul pe care doream să îl implementăm - Forward Checking.

**Principiile jocului:**

Sudoku este un joc logic popular, în care jucătorul trebuie să completeze un grid 9x9 astfel încât:

* Fiecare rând să conțină cifrele de la 1 la 9 fără repetări.
* Fiecare coloană să conțină cifrele de la 1 la 9 fără repetări.
* Fiecare dintre cele 9 sub-grile 3x3 să conțină cifrele de la 1 la 9 fără repetări.

Problema poate fi considerată o problemă de satisfacere a constrângerilor (CSP - *Constraint Satisfaction Problem*), ceea ce o face potrivită pentru aplicarea algoritmilor precum Backtracking sau Forward Checking.

**2) Arhitectura generală**

#### **Structura aplicației**

Aplicația a fost construită în C# folosit Windows Form. Pe parcursul dezvoltării s-a decis implementarea a două clase principale care includ:

* SudokuForm - reprezintă interfața care gestionează interacțiunea cu utilizatorul.
* TableGenerator - care este responsabilă cu generarea tabelelor Sudoku.

Deoarece proiectul principal este realizat în .NET Framework 4.7.2., iar proiectul de testare este realizat în .NET 8.0, s-a creat o problemă de compatibilitate în momentul ruării testelor. Astfel, a fost necesară adăugarea unui nou proiect, utilizând .NET Standard 2.0, în care s-a adăugat codul pentru generarea tablei, **TableGenerator.cs**, urmând ca acesta să fie adăugat ca referință în celelalte două proiecte.

#### **Aspecte teoretice privind algoritmul Forward Checking:**

Algoritmul Forward Checking este o tehnică de optimizare aplicată în problemele de CSP pentru reducerea spațiului de căutare.  
**Principiul de bază**:

După atribuirea unei valori unei variabile, se verifică imediat efectele acestei atribuiri asupra variabilelor neasignate. Dacă cel puțin o variabilă rămâne fără opțiuni valide, se efectuează backtracking.

**Pași principali**:

1. Se selectează o variabilă de asignat (euristică MRV - *Minimum Remaining Values* este des utilizată).
2. Se atribuie o valoare variabilei din domeniul său curent.
3. Se verifică impactul asupra variabilelor neasignate.
4. Dacă domeniul unei variabile devine gol, se revine la pasul anterior (backtracking).
5. Se continuă până când gridul este complet sau s-a determinat că soluția nu există.

Avantaje:

* Reducerea semnificativă a numărului de verificări inutile.
* Îmbunătățirea eficienței față de backtracking-ul clasic.

**3) Funcționalitatea aplicației**

Aplicația oferă următoarele funcționalități esențiale:

##### **Generarea unei tabele noi**

Pentru a se crea o tablă nouă, se apelează funcția

**TableGenerator.SudokuTableGenerator(int difficulty)**.

În momentul apelării acesteia se poate seta dificultatea în momentul creării tabelei, în prezent doar de către dezvoltator. Algoritmul din spate presupune crearea unei matrice complet valide, create cu algoritmul Forward Checking din care se vor șterge un număr egal cu parametrul **difficulty** de valori.

Valorile generate vor fi afișate în celule gri cu scris verde, cu proprietatea ReadOnly.

##### **Validarea valorilor introduse în celule**

Funcția de validare este **IsMoveValid(int row, int column, int value)**.

Aceasta verifică, după ce utilizatorul a introdus câteva valori, dacă acestea sunt valide sau nu. După verificare, valorile valide vor fi de culoare verde, iar cele invalide de culoare roșie. De asemenea se va semnala și printr-un mesaj informativ stadiul jocului după validare (numărul de valori greșite, dacă valorile sunt în regulă, dacă nu mai sunt posibile mutări etc.).

##### **Curățarea tabelei**

În cazul în care utilizatorul dorește reluarea jocului, poate ori să genereze o nouă tabelă, ori să “curețe” valorile introduse de el.

##### **Oferirea de hint-uri**

În cazul în care utilizatorul nu știe cum să rezolve jocul sau are nevoie de ajutor, acesta poate apela la metoda care poate genera o valoare validă. Numărul de hint-uri poate fi setat doar de dezvoltator, în prezent acesta fiind 10. Această funcționalitate este oferită de funcția

**GetPossibleValuesFromForwardChecking(row, column)**

##### **Funcție de rezolvare automată a jocului**

Jocul are și o funcție automată de rezolvare, prin care jucătorul poate vedea rezolvarea unei table în mod automat, pas cu pas. Aceasta are implementat algoritmul de Forward Checking.

##### **Ajutor pentru utilizator**

Dacă utilizatorul nu știe regulile jocului, acesta poate beneficia de o funcție de help care explică regulile jocului.

**4) Rolul fiecarui membru al echipei**

* Creare proiect Github - **Pintilie Justinian**
* Adăugare funcții de generare a matricei pentru Sudoku - **Florea Alexandra**
* Adăugare constrângeri pentru utilizator și validări pentru celule - **Pintilie Justinian**
* Crearea unei schițe a documentației. Scrierea primei părți a documentației- **Pintilie Justinian**
* Corectarea erorilor. Crearea funcției GetHint - **Florea Alexandra**
* Redactarea primei părți a documentației(cap. 1,2,3,5) - **Pintilie Justinian**
* Adăugare funcție de generare soluții cu backtracking - **Florea Alexandra**
* Analiza complexității funcției *private void validateMoveButton\_Click(object sender, EventArgs e)* (punctul 5)- **Pintilie Justinian**
* Analiza corectitudinii funcției *private async Task<bool> SolveSudokuWithForwardCheckingAndBacktrackingAsync()*  (punctul 6) - **Florea Alexandra**
* Crearea și explicarea testelor automate - **Florea Alexandra**
* Redactarea părții a doua a documentației(cap. 4,6,7,8) - **Florea Alexandra**

**5) Complexitatea funcției *private void validateMoveButton\_Click(object sender, EventArgs e)***

**Scopul funcției**

Funcția verifică validitatea valorilor din grila Sudoku, oferind feedback utilizatorului. Marchează celulele incorecte cu roșu, cele corect completate cu verde, și afișează mesaje corespunzătoare progresului (toate completate corect, valori incorecte, sau celule necompletate).

**Complexitatea în cazul cel mai favorabil**

Cazul cel mai favorabil este întâlnit atunci când toate celulele sunt corect completate din prima verificare (countAllGreen == 81). Numărul de operații va fi O(n²), unde n=9 (dimensiunea grilei), deoarece se parcurge întreaga grilă fără ramificații suplimentare.

Complexitatea pentru cazul cel mai favorabil va fi O(n^2) deoarece avem 2 bucle for.

**Complexitatea în cazul cel mai defavorabil**

Cazul cel mai defavorabil este întâlnit atunci când o celulă conține o valoare invalidă și funcția execută un *return* imediat. În cel mai rău caz, funcția trebuie să parcurgă câteva celule până la cea invalidă (de exemplu, prima celulă din grilă poate fi corectă, iar a doua invalidă). Dacă valoarea invalidă este găsită devreme, parcurgerea se oprește imediat. Totuși, în cel mai rău caz (ultima celulă invalidă), se parcurg toate cele 81 de celule.

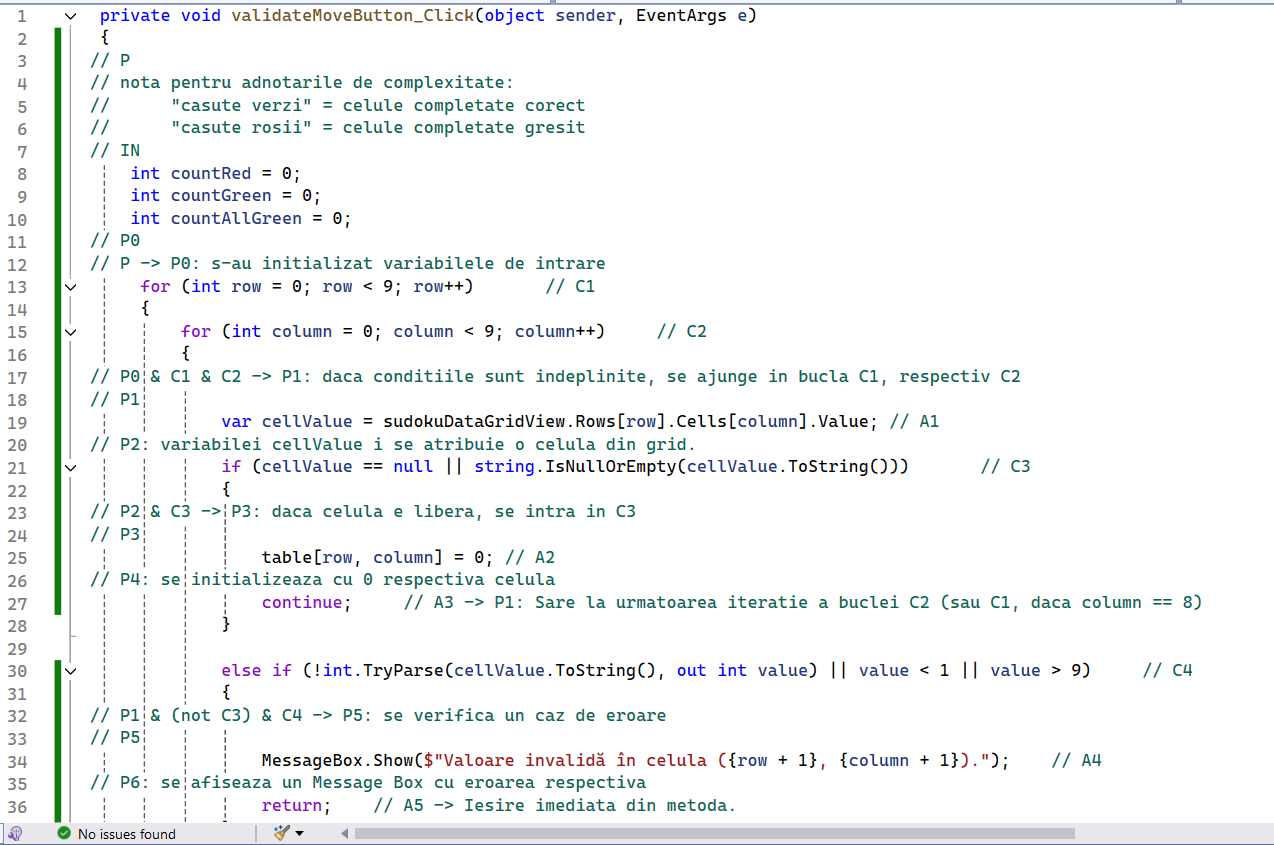
Complexitatea pentru cazul cel mai defavorabil va fi tot O(n^2).

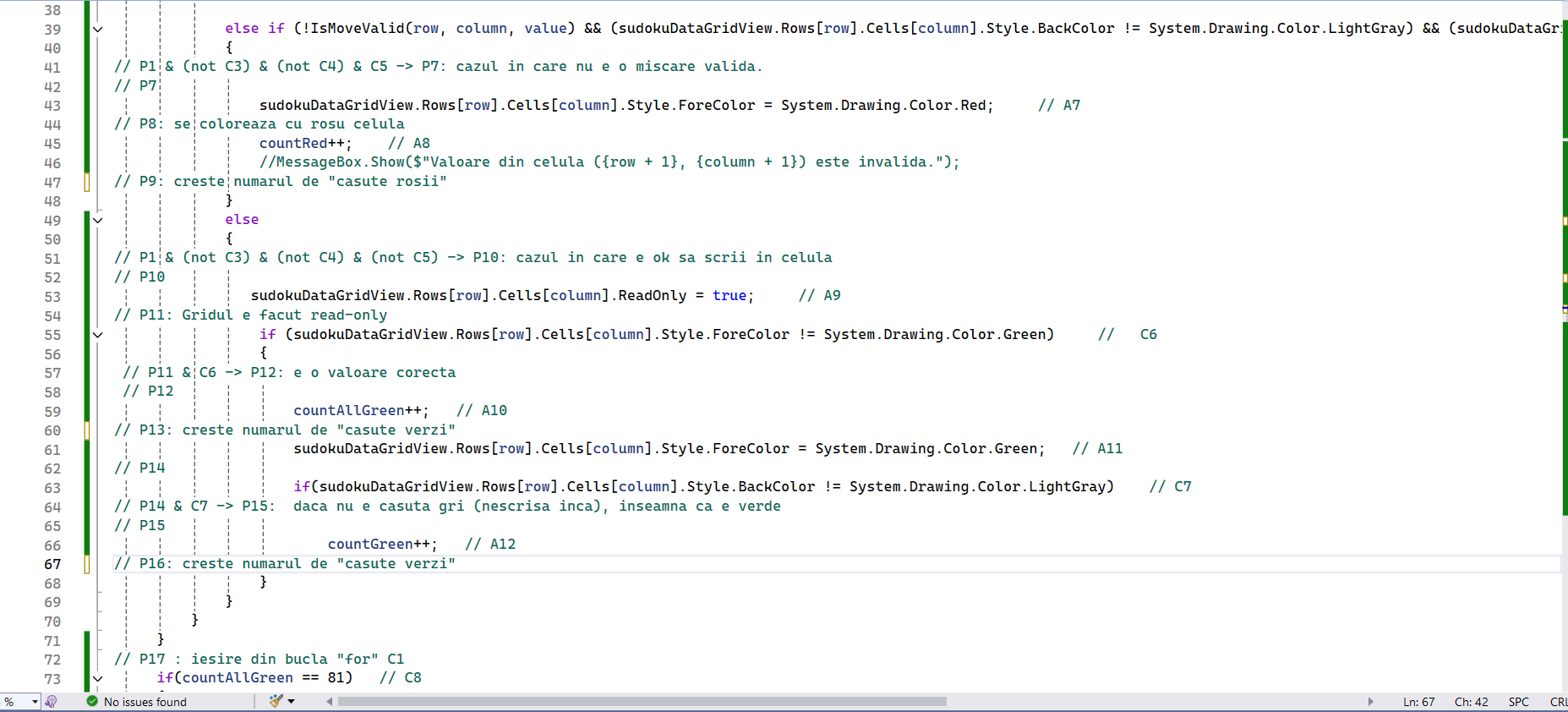
**Complexitatea în cazul mediu**

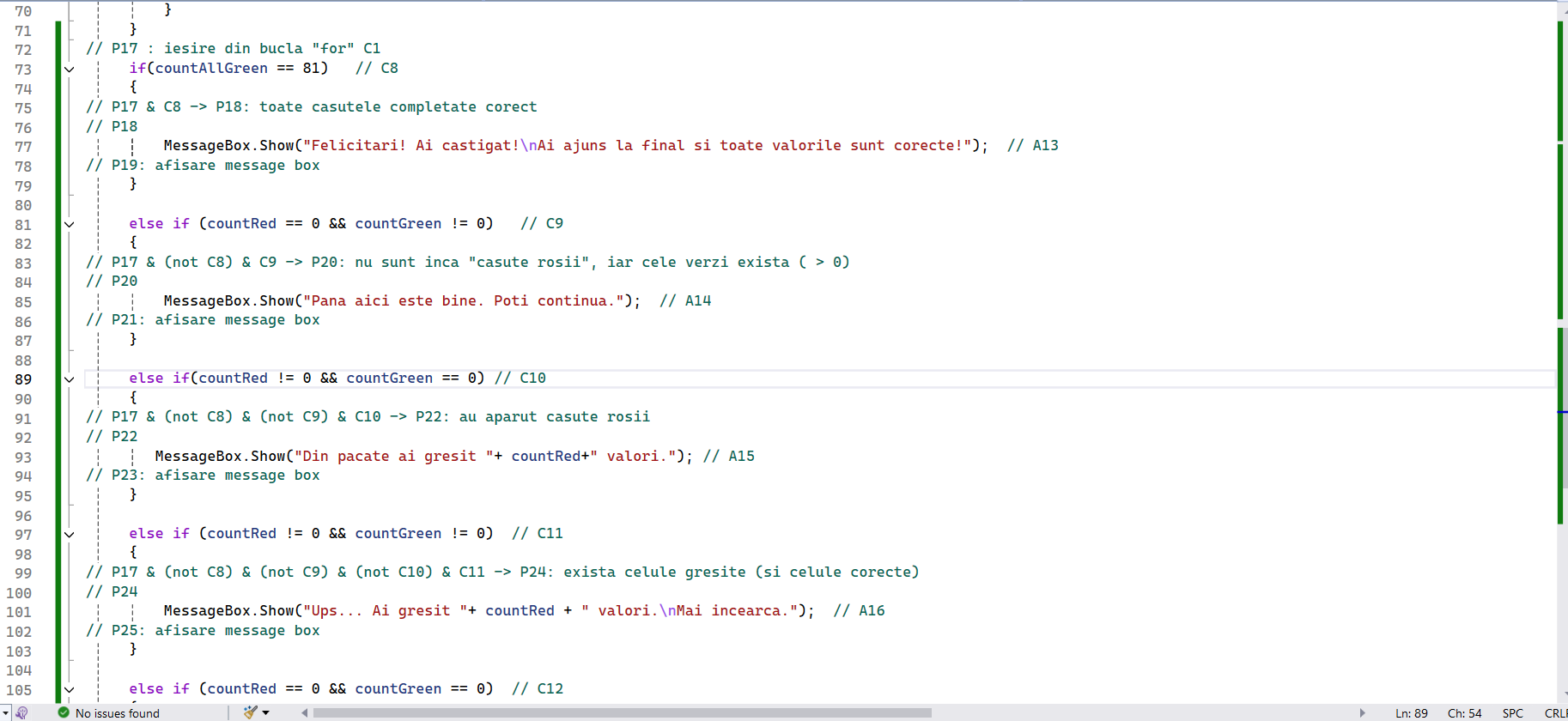
Cazul mediu este întâlnit atunci când grila conține celule goale, corecte și incorecte. Toate celulele sunt verificate și sunt efectuate operații adiționale pentru a marca celulele valide (verzi) sau invalide (roșii). Bucla parcurge întreaga grilă (n^2 = 81 iterații pentru o grilă standard de Sudoku). Fiecare celulă implică verificări adiționale și poate să marcheze stiluri sau să incrementeze contorii, dar aceste operații sunt constante.

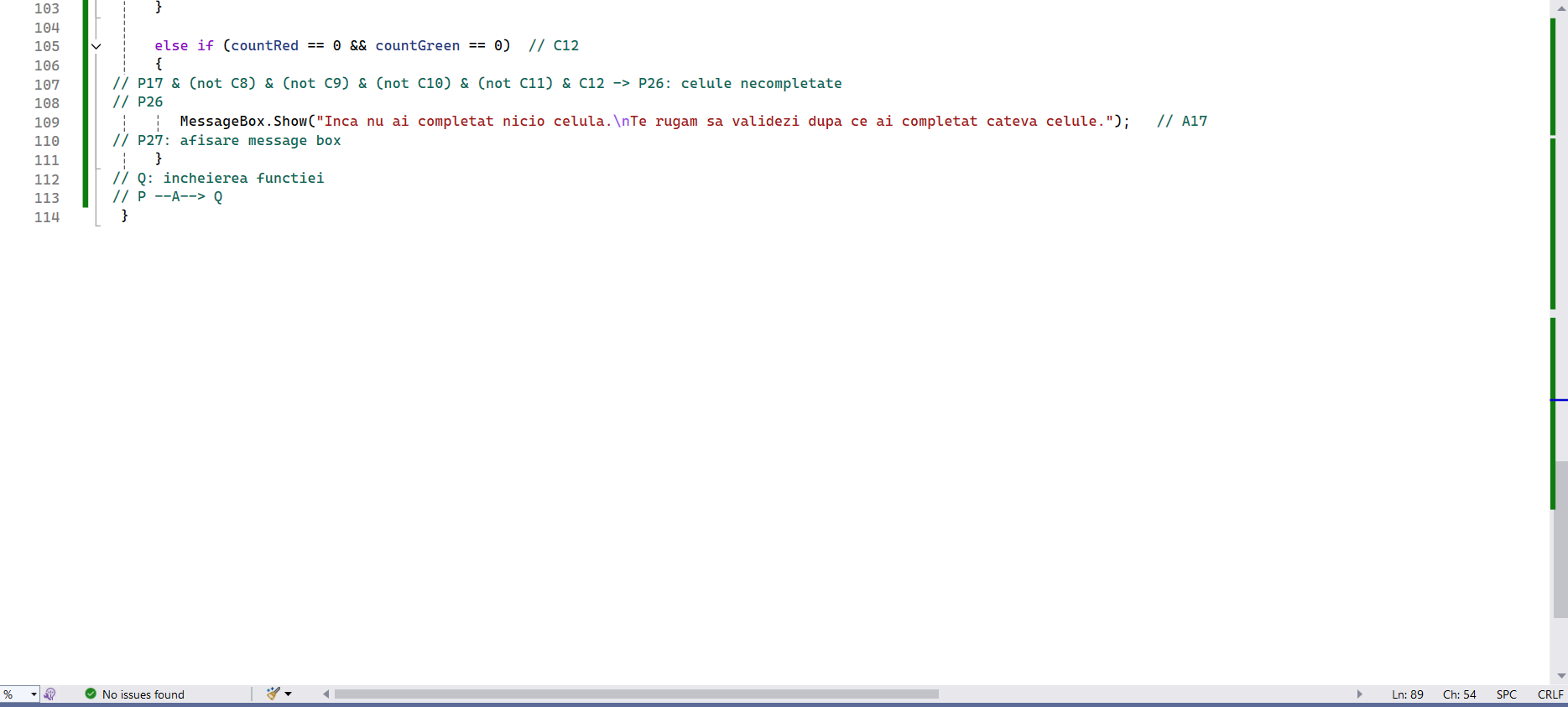
Complexitatea pentru cazul mediu va fi tot O(n^2).

**Codul secvenței analizate**

****

****

****

****

**Demonstrația corectitudinii funcției**

Precondiții:

● *countRed, countGreen, countAllGreen* sa fie 0

Postcondiții:

* Afișarea unui MessageBox, în funcție de cazul obținut.
* Cazuri posibile ce se pot obține:
  + Caz de eroare
  + countRed == 0 && countGreen != 0
  + countRed != 0 && countGreen == 0
  + countRed != 0 && countGreen != 0
  + countRed == 0 && countGreen == 0

Invariantul din bucla 1:

● Denumire: row

● Semnificație: Arată numărul liniei la care se află celula al cărei conținut va fi analizat.

Invariantul din bucla 2:

● Denumire: column

● Semnificație: Arată numărul coloanei la care se află celula al cărei conținut va fi analizat.

**Demonstrație bucla 1 bine definită:**

Invariantul *row* poate avea valori între 0 și 8, care se atribuie corespunzător în interiorul *for*-ului. Condiția *row < 9* este satisfăcută, deci bucla este bine definită.

**Demonstrație bucla 2 bine definită:**

Invariantul *column* poate avea valori între 0 și 8, care se atribuie corespunzător în interiorul *for*-ului. Condiția *column < 9* este satisfăcută, deci bucla este bine definită.

**6) Corectitudinea funcției *private async Task<bool> SolveSudokuWithForwardCheckingAndBacktrackingAsync()***

**Adnotarea metodei:**

**A screenshot of a computer

Description automatically generated**

**Precondiții:**

* **table** – matricea Sudoku de 9x9, în care valorile inițiale sunt deja completate, căsuțele goale au valoarea 0 în matrice.
* **sudokuGridView** – grid-ul cu care interacționează utilizatorul
* Există **cel puțin un spațiu necompletat** în grid, sau **o valoare egală cu 0** în matrice
* Funcția **GetPossibleValuesFromForwardChecking(row, column**) este corect implementată și returnează valorile posibilepentru o anumită celulă

**Postcondiții:**

* Matricea **table** este corect completată sau nu s-a ajuns la o completare validă.
* Toate celulele sunt colorate corespunzător
* Dacă se returnează **true**, Sudoku a fost rezolvat, altfel nu s-a găsit o soluție validă

**Demonstrație formală:**

**P-> P0 –** se trece prin **C1 și C2** care au rolul de căuta în matrice celule necompletate.

**NOT C3 ^P0->C1,C2 -** Dacă celula este deja completată, se trece la următoarea.

**C3 ^P0->P1 –** Se verifică dacă celula este completată si in caz pozitiv se trece la P1

**P1- A1 -> P2 -** Dacă celula este necompletată, se creaza o lista cu valorile posibile

**NOT C4 ^P2 -> P12 –** Se returneaza false daca nu exista o valoare posibila pentru celula

**C4^P2 ->P3** – se intra in bucla in care se trece prin lista cu valori posibile

**P3- A2 -> P4** – se atribuie valoarea din lista tabelului Sudoku si se afiseaza

**C5^P4 –> P5**

**P5- A3,A4**-> **P6 -** In cazul in care valoarea nu era deja generate, se scrie in grid valoarea celulei si se coloreaza cu albastru

**NOT C5 ^P4 -> P6**

**P6 – A5 -> P7**-Se creeaza un delay de 500s pentru a vedea in timp real modificarile facute de algoritm

**P7^C6 ->P8**

**P8 - A6 ->P9** Se trece resursiv la urmatorul pas, daca este totul in regula, se returneaza true.

**NOT C6^ P7 -> P9**

**P9 - A7 -> P10** – daca rezultatul nu este corect, vom pune valoarea 0 in tabel.

**P10 ^C7 ->P11**

**P11-A8,A9 ->P12** – cum rezultatul nu e correct, resetam valoarea, verificam sa nu fie casuta gri si stergem valoarea.

**Demonstratie invariant:**

Invariant:

La fiecare executie se afirma sunt adevarate:

* Fiecare celulă completată din matricea table respectă regulile Sudoku: valorile din rând, coloană și regiunea 3×3 sunt unice.
* Dacă se revine asupra unei celule (backtracking), matricea table revine la starea sa anterioară, păstrând toate constrângerile Sudoku valabile.

La inceput matricea **table** conține valorile inițiale ale Sudoku-ului, date ca intrare. Acestea respectă constrângerile Sudoku, deoarece problema este bine definită. De asemenea,toate celulele necompletate (table[row,column]= 0) sunt evaluate pentru valori posibile prin **forward checking**, care elimină toate valorile ce ar încălca regulile Sudoku.

**Demonstratie:**

Presupunem că invariantul este adevărat după completarea a **k** celule:

* Toate celulele completate respectă constrângerile Sudoku.
* Dacă se revine asupra unei celule în timpul backtracking-ului, matricea revine la starea anterioară, menținând validitatea.

**Dem. Pas k+1:**

Se consideră celula curenta (row,column) care trebuie completată:

1. Se generează lista **possibleValues** folosind **GetPossibleValuesFromForwardChecking(row, column).**
2. Algoritmul alege o valoare **value** care apartine **possibleValues** și o atribuie celulei: **table[row,column]=value**.
3. Algoritmul încearcă să completeze următoarele celule apelând recursiv funcția **SolveSudokuWithForwardCheckingAndBacktrackingAsync**.
   * Dacă recursivitatea duce la o soluție validă (toate celulele sunt completate), funcția returnează true, iar invariantul este menținut.
   * Dacă recursivitatea nu găsește o soluție, algoritmul resetează valoarea celulei curente la 0 (backtracking): **table[row,column]=0** si matricea table revine la starea anterioară, păstrând toate constrângerile Sudoku.

**Concluzie pentru pasul k+1:**

* Dacă o valoare validă este atribuită celulei curente, iar recursivitatea duce la o soluție, toate celulele respectă regulile Sudoku (invariantul este adevărat).
* Dacă nu există soluții, backtracking-ul revine la starea anterioară fără a încălca regulile Sudoku (invariantul este menținut).

**Concluzia demonstratiei:**

Prin inducție matematică, am demonstrat că invariantul este adevărat la inițializarea algoritmului si la fiecare pas k și k+1, fie că o valoare este atribuită, fie că algoritmul revine prin backtracking.

Astfel, algoritmul produce fie o soluție validă care respectă toate regulile Sudoku, fie confirmă că nu există soluții, păstrând invariantul pe tot parcursul execuției.

**Finitudinea:**

Buclele for sunt de la 0 la 8, deci in cazul in care indicii ajung la valoarea 8 se incheie iteratiile. Cat despre functia recursive, aceasta se repeat atata timp cat exista valori posibile, deci in cazul in care prima valoare din tabel si mai apoi toate cele de dupa au luat toate valorile posibile din lista, algoritmul returneaza false si se incheie.

**7) Explicația testelor automate și rezultatele acestora**

Astfel, am decis să creăm următoarele teste pentru metodele din clasa **TableGenerator.cs**:

* **TestGenerateValidTable()**

Metoda testată:

A screenshot of a computer code

Description automatically generated

Testele:

**A screenshot of a computer code

Description automatically generated**

Aceste teste au rolul de a verifica dacă generarea unui tabel nu returneaza null și dacă tabelul generat este unul valid din punct de vedere al condițiilor Sudoku.

****

În urma rulării testelor, rezultatele au fost corecte, lucru care redă corectitudinea metodei.

Alte teste:

**A screen shot of a computer

Description automatically generated**

Acesta verifică dacă tabelul generat nu este null și dacă după generarea unei table întregi nu mai rămâne nicio valoare egală cu 0.



În urma rulării, testele au ieșit corecte

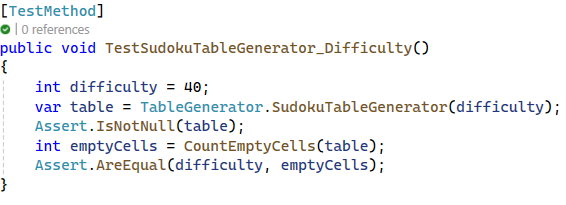
* **TestSudokuTableGenerator\_Difficulty()**

Metoda testată:

A screenshot of a computer program

Description automatically generated

Testele:

****

Aceste teste au rolul de a valida metoda care setează numărul de celule goale din tabelul generat. În acest caz se verifică inițial dacă după crearea unei tabele cu dificultatea 40 nu se creează un tabel null și dacă într-adevăr numărul de căsuțe goale din tabel coincide cu numărul atribuit dificultății.



După rularea testelor se observă că acestea au returnat valorile corecte.

* **TestUpdatePossibleValues()**

Metoda testată:

A screenshot of a computer program

Description automatically generated

Testele:

A screenshot of a computer program

Description automatically generated

Prin aceste teste se dorește a se pune în vedere corectitudinea alegerii momentului în care se face update-ul. Astfel, inițial se pune numarul 5 la coordonatele (0,0), iar apoi se testeaza dacă este posibil ca numărul 5 să fie plasat la coordonatele (0,1), (1,0), (1,1), lucru care nu este permis conform constrângerilor.

Rezultate:

A screenshot of a computer

Description automatically generated

**8)Bibliografie**

● https://sandipanweb.wordpress.com/2017/03/17/solving-sudoku-as-a-constraint-satisfaction-problem-using-constraint-propagation-with-arc-consistency-checking-and-then-backtracking-with-minimum-remaning-value-heuristic-and-forward-checking/

● https://www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/

● https://www.youtube.com/watch?v=ejhmLHd3n\_4&list=LL&index=11&t=70s&ab\_channel=BrandonAdame

● https://cs188ai.fandom.com/wiki/Forward\_Checking

● https://www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/