Inteligență artificială
Rezolvarea integramelor folosind algoritmul Forward Checking

Coordonator:
prof. dr.ing. Leon Florin

Studenți:
Chelea Diana-Maria
Spiridon Bianca

Universitatea Tehnică „Gheorghe Asachi" Iași

Cuprins

1. Descrierea problemei considerate.....**3**
2. Aspecte teoretice privind algoritmul.....**3**
2.1. Probleme de satisfacere a constrângerilor.....**3**
2.2. Algoritmul Forward Checking.....**4**
2.3. Euristici de căutare - Cea mai constrânsă variabilă……..**5**
3. Modalitatea de rezolvare.....**5**
4. Listarea părților semnificative din codul sursă însoțite de explicații și comentarii.....**9**
5. Rezultatele obținute prin rularea programului în diverse situații, capturi ecran și comentarii asupra rezultatelor obținute.....**12**
6. Teste și rezultatele acestora……**13**
7. Concluzii.....**16**
8. Bibliografie.....**16**
9. O listă cu ce a lucrat fiecare membru al echipei.....**17**

1. Descrierea problemei considerate

Ne-am propus să implementăm un program care să fie capabil să rezolve integrame utilizând algoritmul Forward Checking, pe baza unei liste de cuvinte și a unui grid predefinit. O integramă este un joc de tip puzzle, în care trebuie completate spațiile goale, astfel încât cuvintele să se intersecteze corect, iar literele comune să coincidă. De asemenea, fiecare cuvânt din listă poate fi utilizat o singură dată. Pentru a rezolva această problemă, vom utiliza algoritmul Forward Checking, o tehnică utilizată în probleme de satisfacere a constrângerilor. Acesta funcționează prin verificarea anticipată a domeniilor fiecărei variabile, cu scopul de a evita imposibilitățile. Dacă un domeniu devine vid, intervine procesul de backtracking, încercând o altă valoare pentru variabila precedentă.
De asemenea, pentru a optimiza și mai mult procesul de căutare, am folosit euristica celor mai puține valori rămase.

Combinarea algoritmului Forward Checking cu euristica variabilei celei mai constrânse duce la o strategie de căutare mai eficientă pentru rezolvarea integramelor, cu un număr de pași considerabil redus, în comparație cu simplul Backtracking.

2. Aspecte teoretice privind algoritmul

2.1. Probleme de satisfacere a constrângerilor (PSC)

În cadrul inteligenței artificiale, pentru a rezolva probleme ce țin de căutarea unei soluții care respectă toate constrângerile inițiale, sunt folosite diverse tehnici și algoritmi de căutare și optimizare. Scopul este reprezentat de identificarea unei combinații de valori pentru variabilele de intrare ale problemei, astfel încât toate constrângerile să fie satisfăcute.

O problemă de satisfacere a constrângerilor este definită de următoarele atribute:

1. Variabile: o mulțime {X1, X2, …, Xn}
2. Domeniu: reuniunea tuturor domeniilor fiecărei variabile (D1,D2…,Dn); setul de valori posibile pe care o variabilă le poate avea, finit sau infinit; de exemplu, în cadrul aplicației de rezolvare a integramelor, domeniul unei variabile este reprezentat de mulțimea de poziții pe care poate fi așezat un cuvânt în grid-ul curent;
3. Constrângeri: set de reguli pe care valorile variabilelor trebuie să le respecte simultan (C1,C2,....Cn);

Exemple de PSC: problema celor N-regine, Sudoku, ***rezolvarea integramelor,*** planificări.

Dintre algoritmii folosiți în rezolvare PSC, amintim Backtracking și Forward Checking.

2.2. Algoritmul Forward Checking (căutare înainte)

Inițial, în anii 1960-1970, rezolvarea PSC a început cu algoritmi simpli de Backtracking. Totuși, s-a observat că aceste metode consumau mult timp și resurse, explorând soluții inutile. Pentru a optimiza procesul, în anii 1980 a fost introdus algoritmul de Forward Checking (de către Haralick și Elliott), conceput să reducă numărul de ramuri din arborele de căutare. Acest algoritm se bazează pe faptul că odată ce unei variabile Xi îi este asignată o valoare v, o parte din următoarele variabile și valori (Xj, v') devin imposibile. FC verifică imediat cum afectează o alegere curentă celelalte variabile ce urmează a fi rezolvate. Dacă o valoare v' pentru o variabila Xj nu mai respectă constrângerile din cauza unei alegeri actuale, acea valoare este eliminată din posibilitățile variabilei Xj. Aceasta reprezintă principala deosebire dintre Backtracking și FC, cel din urmă algoritm anticipând problemele viitoare și verificând constrângerile doar după ce a atribuit valori tuturor variabilelor până în acel punct, reducând astfel spațiul de căutare.

```
REDUCERE-DOMENIU
    intrări: problemă : PROBLEMĂ
             variabila-curentă : VARIABILĂ » ultima variabilă atribuită
             soluţie : ATRIBUIRE*
             domeniu : (VARIABILĂ, VALOARE*)
    ieşiri:  domeniu : (VARIABILĂ, VALOARE*)
             este-vid : BOOLEAN » dacă domeniul unei variabile a devenit vid
-----------------------------------------------------------------------------------------------------------
variabile-vecine ← problemă.Variabile-neatribuite.Caută(v : Există-constrângere-între(v, variabila-curentă))

pentru-fiecare var din variabile-vecine
    pentru-fiecare val din domeniu[var]
        dacă nu Este-consistentă(ATRIBUIRE(variabila-curentă, valoare-variabilă-curentă), ATRIBUIRE(var, val)) ...
            atunci
            domeniu[var].Elimină(val)
            dacă domeniu[var] = [ ] atunci
                întoarce (Nul, Adevărat)
întoarce (domeniu, Fals)
```

FC combină verificarea anticipată cu mecanismul de Backtracking pentru a gestiona situațiile în care constrângerile blochează soluția.

Pseudocod

```
BACKTRACKING-CU-VERIFICARE-ÎNAINTE
    intrări: problemă : PROBLEMĂ
             soluţie : ATRIBUIRE* » atribuiri parţiale; în apelul iniţial, o listă vidă
             domeniu : (VARIABILĂ, VALOARE*)
                 » valorile posibile ale variabilelor la un moment dat
                 » în apelul iniţial, cele date de funcţia Valori-permise
    ieşiri:  soluţie : ATRIBUIRE* » o atribuire completă
-----------------------------------------------------------------------------------------------------------
dacă problemă.Este-completă(soluţie) atunci
    întoarce soluţie

variabilă ← problemă.Variabile-neatribuite().Primul-element() » sau euristicile E2, E1

pentru-fiecare valoare din problemă.Valori-permise(variabilă)
    » sau euristica E3
    atrib ← ATRIBUIRE(variabilă, valoare)
    dacă pentru-fiecare a din soluţie, problemă.Este-consistentă(atrib, a) atunci
        soluţie.Adaugă(atrib)
        Variabile-neatribuite.Elimină(variabilă)
        (domeniu-nou, este-vid) ← REDUCERE-DOMENIU(problemă, variabilă, soluţie, domeniu)
        dacă nu este-vid atunci
            rezultat ← BACKTRACKING-CU-VERIFICARE-ÎNAINTE(problemă, soluţie, domeniu-nou)
            dacă rezultat ≠ Nul
                întoarce rezultat
        soluţie.Elimină(atrib)
        Variabile-neatribuite.Adaugă(variabilă)
întoarce Nul
```

2.3 Euristici de căutare - Cea mai constrânsă variabilă

Euristicile de căutare reprezintă tehnici folosite pentru a ghida procesul de căutare într-un spațiu mare de soluții, în scopul de a găsi răspunsul mai rapid și eficient. Acestea nu garantează întotdeauna găsirea celei mai bune soluții, dar ajută algoritmul să se concentreze pe soluțiile aparent corecte, evitând explorarea completă a întregului spațiu de căutare.

Dintre euristicile de căutare posibile, problema rezolvării integramelor a utilizat ”Cea mai constrânsă variabilă” (MCV). Se alege variabila cu cele mai puține valori permise rămase (cea mai constrânsă). Concentrarea pe variabilele care sunt cele mai limitate din punct de vedere al valorilor posibile poate duce la descoperirea rapidă a unei soluții sau la detectarea anticipată a unei incompatibilități. Pseudocodul pentru această euristică:

```
PROBLEMĂ::Cea-mai-constrângătoare-variabilă
    ieşiri:   cmcv : VARIABILĂ » cea mai constrângătoare variabilă
-------------------------------------------------------------------------------------------------------------
pentru-fiecare variabilă din Variabile-neatribuite()
    » numărul de variabile neatribuite diferite de variabilă, cu care aceasta are constrângeri
    nr-constrângeri[variabilă] ← Variabile-neatribuite().Numără(v : v ≠ variabilă şi ...
        Există-constrângere-între(v, variabilă))

» se returnează variabila cu numărul maxim de constrângeri
» (funcția Argmax întoarce indexul elementului cu valoare maximă)
întoarce Variabile-neatribuite()[nr-constrângeri.Argmax()]
```

3. Modalitatea de rezolvare

Așa cum am menționat și mai sus, programul nostru folosește algoritmul FC, cu euristica celor mai puține valori rămase.

Tipurile de constrângeri întâlnite în rezolvarea integramelor sunt:

- ocuparea spațiilor disponibile : cuvintele nu trebuie să depășească gridul
- pozițiile trebuie să fie libere sau să coincidă cu literele existente
- pozițiile disponibile trebuie să existe(dacă rămânem fără poziții libere s-a ajuns la soluție)
- dacă mai avem poziții libere, dar s-a ajuns la imposibilitate, se revine la starea anterioară.

De asemenea euristica este utilă în a simplifica procesul, eliminând din start cele mai restrictive părți ale problemei. Această strategie duce la găsirea rapidă a soluției, întrucât, abordând mai întâi variabilele cu cele mai puține opțiuni, se reduc șansele de a întâmpina conflicte în etapele ulterioare ale căutării.

Pentru a înțelege cum se aplică algoritmul, am ales un exemplu simplificat.

GRID:

```
grid = [
    ['0', '0', '0', '0'],
    ['0', '#', '#', '0'],
    ['0', '0', '0', '0'],
    ['0', '#', '#', '0']
]
```

CUVINTE:

```
# Lista de cuvinte posibile (simplificată)
cuvinte = ["TART", "TARS", "RATA", "STAR"]
```

PAȘI:

1. La început, pentru fiecare variabilă se memorează mulțimea curentă de valori permise.

```
TART
[(O, 0, 0), (V, 0, 0), (V, 0, 3), (O, 2, 0)]
TARS
[(O, 0, 0), (V, 0, 0), (V, 0, 3), (O, 2, 0)]
RATA
[(O, 0, 0), (V, 0, 0), (V, 0, 3), (O, 2, 0)]
STAR
[(O, 0, 0), (V, 0, 0), (V, 0, 3), (O, 2, 0)]
```

2. După se alege variabila cu cele mai puține valori permise(în cazul de față, numărul de valori este egal la toate variabilele, așa că se alege prima apariție)

```
['T', 'A', 'R', 'T']
['0', '#', '#', '0']
['0', '0', '0', '0']
['0', '#', '#', '0']
```

3. Când se atribuie o valoare în procesul de căutare, se actualizează mulțimile de valori permise pentru toate variabilele

```
TARS
[(V, 0, 0), (V, 0, 3), (O, 2, 0)]
RATA
[(O, 2, 0)]
STAR
[(O, 2, 0)]
```

4. Se alege variabila cu cele mai puține valori permise

```
['T', 'A', 'R', 'T']
['0', '#', '#', '0']
['R', 'A', 'T', 'A']
['0', '#', '#', '0']
```

5. Când se ajunge în imposibilitate, se șterge cuvântul curent şi se reia variabila anterioară cu altă valoare din posibilitățile ei de mutare.

```
['T', 'A', 'R', 'T']
['0', '#', '#', '0']
['0', '0', '0', '0']
['0', '#', '#', '0']
--------------------
['0', '0', '0', '0']
['0', '#', '#', '0']
['0', '0', '0', '0']
['0', '#', '#', '0']
--------------------
['T', '0', '0', '0']
['A', '#', '#', '0']
['R', '0', '0', '0']
['T', '#', '#', '0']
```

6. Când se atribuie o valoare în procesul de căutare, se actualizează mulțimile de valori permise pentru toate variabilele

```
TARS
[(O, 0, 0), (V, 0, 3)]
RATA
[(V, 0, 3), (O, 2, 0)]
STAR
[(V, 0, 3)]
```

7. Se alege variabila cu cele mai puține valori permise

```
['T', '0', '0', 'S']
['A', '#', '#', 'T']
['R', '0', '0', 'A']
['T', '#', '#', 'R']
```

8. Când se atribuie o valoare în procesul de căutare, se actualizează mulțimile de valori permise pentru toate variabilele

```
TARS
[(0, 0, 0)]
RATA
[(0, 2, 0)]
```

9. Se alege variabila cu cele mai puține valori permise

```
['T', 'A', 'R', 'S']
['A', '#', '#', 'T']
['R', '0', '0', 'A']
['T', '#', '#', 'R']
```

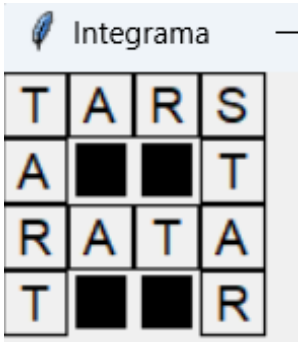
10. Când se atribuie o valoare în procesul de căutare, se actualizează mulțimile de valori permise pentru toate variabilele

```
RATA
[(0, 2, 0)]
```

11. Se alege variabila cu cele mai puține valori permise

```
['T', 'A', 'R', 'S']
['A', '#', '#', 'T']
['R', 'A', 'T', 'A']
['T', '#', '#', 'R']
```

12. Rezultat final :



4. Listarea părților semnificative din codul sursă însoțite de explicații și comentarii

Principalele atribute ale problemei:
Variabilele - vectorul ”cuvinte”, în care se află toate variantele de cuvinte pentru a rezolva integral integrama, cu valori citite din fișier;
Domeniu - vector de ”vectori de poziție” - domenii, aici salvăm toate pozițiile posibile la momentul curent pentru fiecare variabilă în cadrul grid-ului curent;
Constrângeri - verificate în funcția *update:*
1. Lungimea cuvântului pe care vrem să îl adăugăm în grid să nu depășească limitele integramei.
2. Pozițiile pe care vrem să plasăm cuvântul să fie valabile (adică să nu fie marcate în grid cu o altă literă. ci cu ‘0’).
3. Literele cuvântului pe care dorim să-l plasăm la anumite coordonate în grid-ul curent să corespundă cu literele care deja se regăsesc în grid (să nu apară conflicte pe orizontală sau pe verticală).

Euristica de căutare - cea mai constrânsă variabilă:
Urmărind pseudocodul prezentat anterior, funcția de căutare a variabilei cu cele mai puține poziții în gridul actual:

```
# alegere variabila cu cele mai putine valori permise
def cea_mai_constransa_variabila(domenii):
    #numarul de valori permise pentru variabila, consistente cu solutia partiala
    lungime_minima = float('inf')
    for variabila in domenii:
        contor = len(domenii[variabila])
        if contor < lungime_minima:
            cmcv = variabila
            lungime_minima = contor
    #returneaza variabila cu numarul minim de valori posibile ramase
    return cmcv
```

Practic, în codul sursă salvăm domeniul posibil pentru toate variabilele (cuvintele din dicționar), apoi comparăm lungimile fiecărui vector de poziție pentru a determina care este cel mai scurt. Acesta va fi selectat pentru procesare, eficientizând căutarea pe rute care probabil ar duce la incompatibilități ulterioare.

Clasa ”Coordonate”

```
#coordonata pentru pozitie (orizontal/vertical , plasare pe harta(linie,coloana))
class Coordonata:
    def __init__(self, directie, linie, coloana):
        self.directie = directie
        self.linie = linie
        self.coloana = coloana

    def __repr__(self):
        return f"({self.directie}, {self.linie}, {self.coloana})"
```

Pentru a salva pozițiile fiecărei variabile și a le elimina ulterior din domeniul de valori, am

creat o clasă în care memorăm direcția în integramă - orizontală sau verticală, marcate prin caracterele “O”/”V”, dar și coordonatele de început ale cuvântului - linia și coloana corespunzătoare în grid. În vectorul ”domenii” vom avea vectori alcătuiți din obiecte de tip ”Coordonata”.

Funcția update

La fiecare iterație a algoritmului Forward Checking trebuie să actualizăm lista de poziții posibile, deoarece odată plasat un cuvânt în grid, se restrâng variantele de plasare ale viitoarelor cuvinte - acestea sunt constrânse de noile literele din integramă.
Parcurgem fiecare linie și coloană a gridului curent, verificăm constrângerile și marcăm poziția ca fiind liberă prin adăugarea acesteia în vectorul de poziții al cuvântului curent.

```
for i in range(len(grid)):
    for j in range(len(grid[i])):
        if j + len(cuvant) <= len(grid[i]):
            pozitie_libera = True
            for litera in range(len(cuvant)):
                #daca o litera este deja ocupata si e diferita de litera cuvantului curent->
                if grid[i][j + litera] != '0' and grid[i][j + litera] != cuvant[litera]:
                    pozitie_libera = False
                    break
            if pozitie_libera:
                pozitii.append(Coordonata("O", i, j))
```

Funcția forward_checking

Inițial apelăm funcția de modificare a domeniului curent, verificăm dacă soluția este completă, adică dacă toate cuvintele au fost plasate cu succes în grid. Altfel, continuăm prin verificarea în ordinea dată de euristică a tuturor cuvintelor. Dacă se ajunge într-un punct din care nu se mai poate continua rezolvarea, algoritmul se întoarce la pasul anterior, verificând din nou alte poziții disponibile pentru cuvântul respectiv. Pentru a elimina un cuvânt introdus eronat, salvăm o variabilă temporară ce va conține gridul anterior introducerii cuvântului:

```
if coordonata.directie == "O":
    for litera in range(len(cuvant_curent)):
        #se salveaza un temp in cazul in care cuvantul va duce la imposibilitati si se va relua
        temp.append(grid[coordonata.linie][coordonata.coloana + litera])
        grid[coordonata.linie][coordonata.coloana + litera] = cuvant_curent[litera]
```

Vom apela recursiv funcția de forward_checking, până când se returnează ”false”, adică problema a ajuns în imposibilitatea de a fi rezolvată.

```
for cuvant in cuvinte:
    if cuvant != cuvant_curent:
        cuvinte_ramase.append(cuvant)
#recursivitate cu noua lista de cuvinte disponibila
if forward_checking(grid, cuvinte_ramase):
    return True
#se sterge in cazul in care nu ajungem la solutii
for litera in range(len(cuvant_curent)):
    grid[coordonata.linie][coordonata.coloana + litera] = temp[litera]
```

Interfața grafică cu tkinter:

```
for i, row in enumerate(grid):
    for j, celula in enumerate(row):
        if celula == "#":
            label = tk.Label(root, text=" ", width=2, height=1, bg="black", relief="solid")
        else:
            label = tk.Label(root, text=celula, width=2, height=1, borderwidth=1, relief="solid", font=("Arial", 14))
        label.grid(row=i, column=j)
```

Apelul programului în main: citim din fișiere gridul și lista de cuvinte necesare completării, apoi apelăm funcția forward_checking:

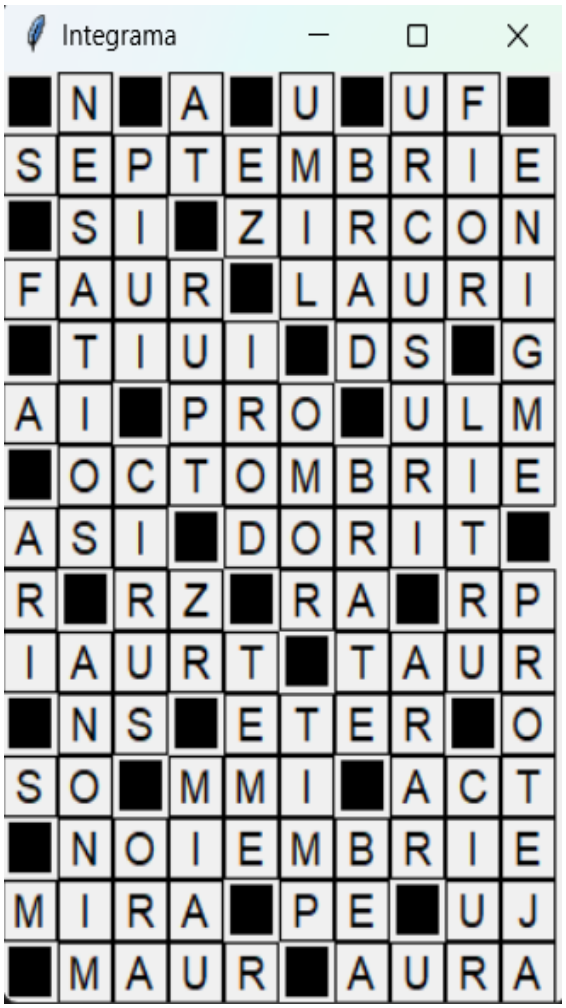
```
if __name__ == "__main__":
    grid=[]
    cuvinte=[]
    with open("grid_exemplu1.txt", "r") as file:
        for i in file:
            grid.append(list(i.strip().split()))

    with open("cuvinte_exemplu1.txt", "r") as file:
        for i in file:
            cuvinte.append(i.strip())

    if forward_checking(grid, cuvinte):
        display_result(grid)
    else:
        print("Nu s-a gasit nici o solutie.")
```

5. Rezultatele obținute prin rularea programului în diverse situații, capturi ecran și comentarii asupra rezultatelor obținute

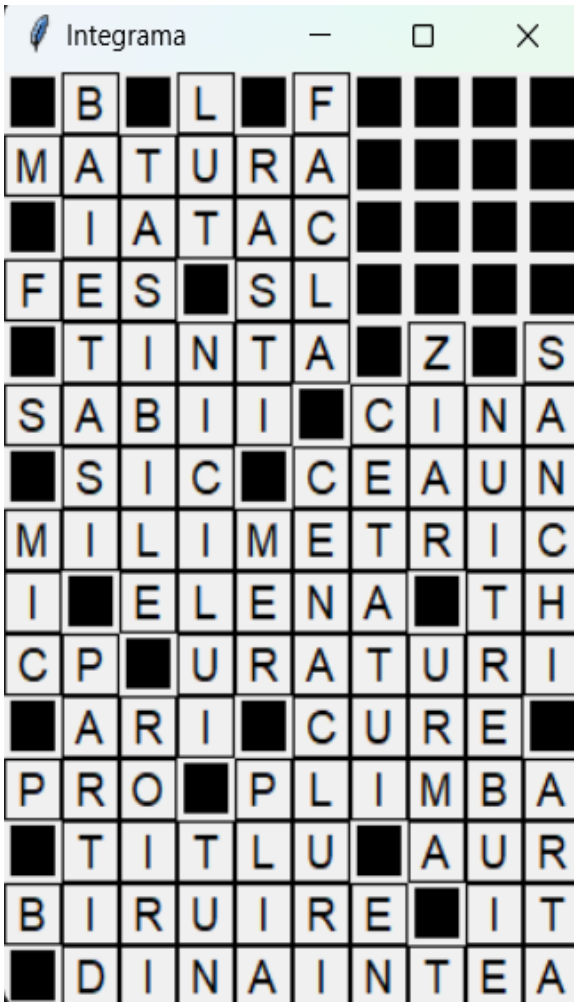
Interfața aplicației conține soluția finală.

Mai jos sunt afișate soluțiile pentru cele 2 exemple prezentate în programul nostru.

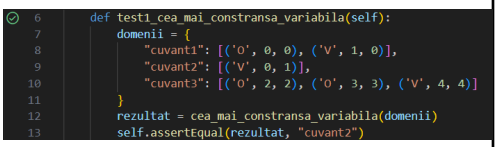
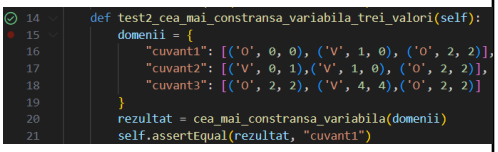
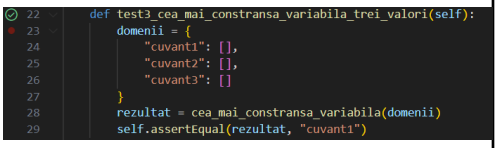
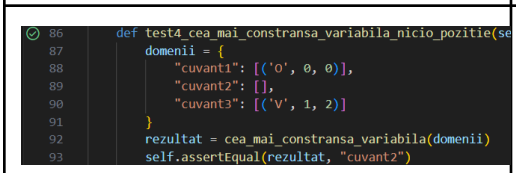
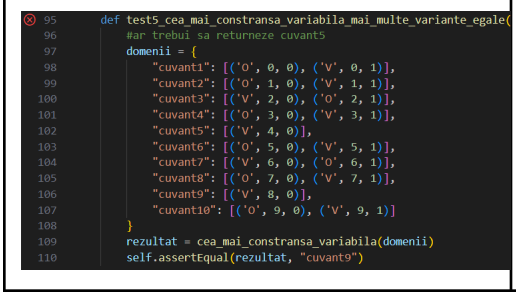
Exemplul 1 :

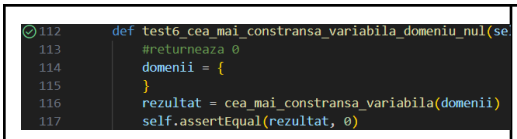
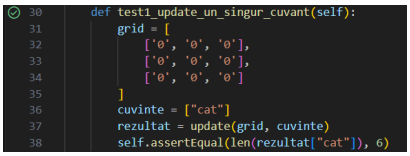
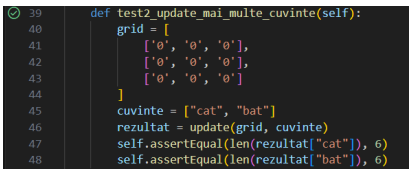
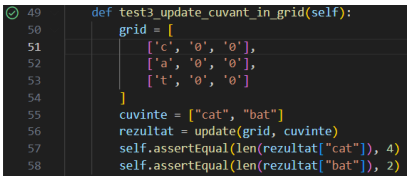
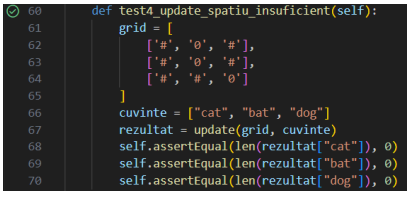


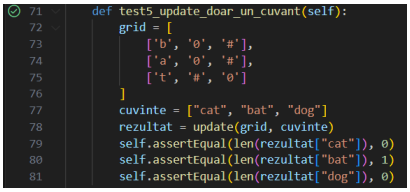
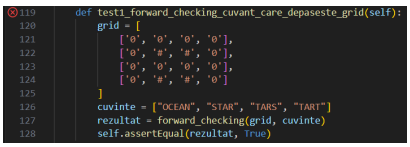
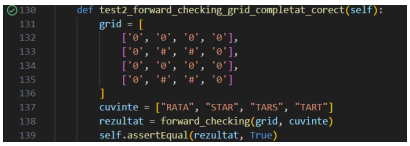
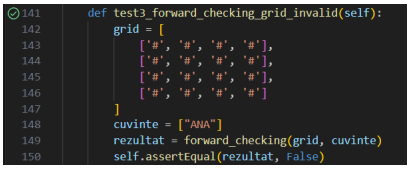
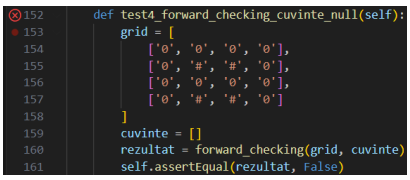
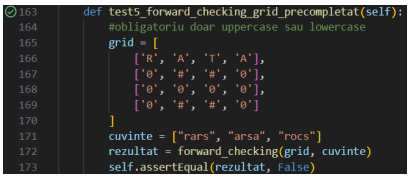
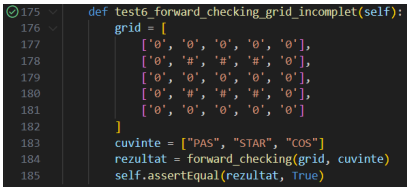
Exemplul 2 :

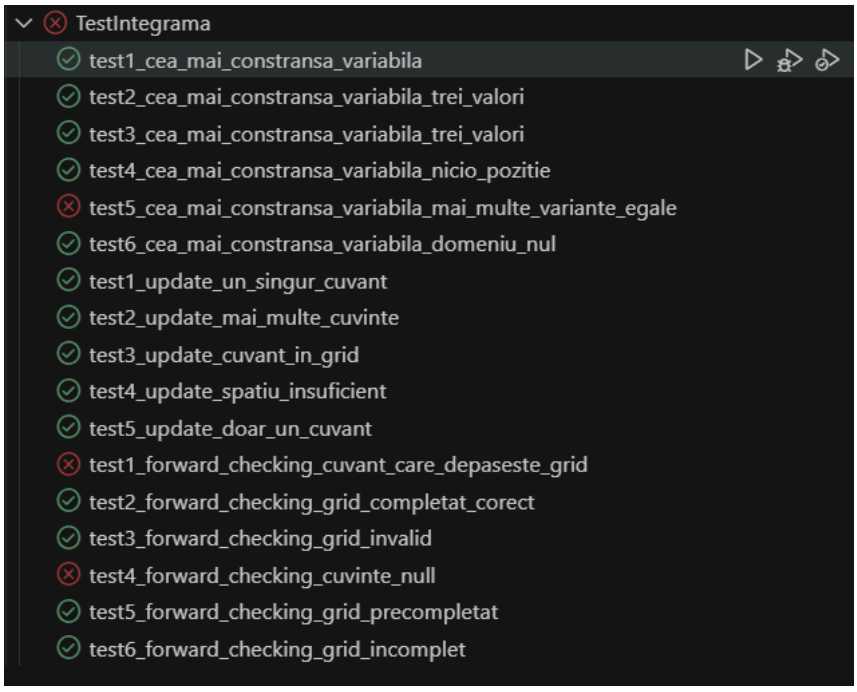


6. Teste și rezultatele acestora

Test	Explicație	Rezultat așteptat	Rezultat primit	OK/Failed
	testează selecția celei mai scurte liste de poziții când fiecare linie are un număr diferit de elemente	cuvant 2	cuvant2	OK
	testează selecția celei mai scurte liste de poziții când liniile au număr egal de elemente, trebuie să selecteze prima linie pe care o întâlnește	cuvant 1	cuvant1	OK
	testează cazul unui vector cu domenii vide, trebuie să selecteze prima linie pe care o întâlnește	cuvant 1	cuvant1	OK
	testează cazul în care una dintre variabile nu are nicio poziție în domeniu, trebuie să o selecteze	cuvant 2	cuvant2	OK
	testează o listă mai mare de variabile cu diferite lungimi de domenii	cuvant 5	cuvant9	Failed

	testează comportamentul funcției în cazul parametrului cu vector vid	0	0	OK
	testează comportamentul funcției în cazul în care grila este necompletată, numărul de poziții disponibile fiind maxim	6	6	OK
	testează comportamentul funcției pentru 2 cuvinte; și pentru cel de-al doilea cuvânt numărul de poziții disponibile va fi egal cu 6, întrucât grid-ul e gol	6,6	6	OK
	testează cazul în care o literă sau mai multe din grid vor fi folosite în scrierea altui cuvânt; în acest caz, pentru ‘cat’ vor fi 4 poziții disponibile, întrucât se va suprapune cu cuvântul completat anterior în grid	4,2	4,2	OK
	testează cazul în care nici un cuvânt nu are spațiu suficient	0,0,0	0,0,0	OK

	testează pentru un grid intermediar, adăugarea de poziții pentru cuvintele rămase	0,1,0	0,1,0	OK
	testează cu o listă de cuvinte pentru care nu există soluție în grid (unul din cuvinte depășește limitele grilei)	False	True	Failed
	testează un grid și o listă de cuvinte pentru un exemplu facil	True	True	OK
	testează un grid alcătuit din spații pe care nu se poate completa	False	False	OK
	testează cazul unui vector de variabile gol - gridul nu se va modifica	True	False	Failed
	testează compatibilitatea cuvintelor lowercase și uppercase în același grid	False	False	OK
	testează dacă gridul este neinițializat și lista de cuvinte este incompletă - va plasa cuvintele respective în pozițiile corespunzătoare	True	True	OK



Rezultatele testelor în Python

7. Concluzii

Așadar programul vine în ajutorul rezolvării integramelor cu o mapă dată și cuvinte predefinite, utilizând algoritmul forward checking cu o euristică de optimizare a numărului de iterații. Prin îmbinarea tradiției jocului de integrame cu tehnicile moderne de inteligență artificială, proiectul nostru oferă o soluție inovatoare și eficientă pentru rezolvarea automată a acestor puzzle-uri.

8. Bibliografie

Curs 3 - http://florinleon.byethost24.com/Curs_IA/IA03_Jocuri_CSP.pdf
Forward Checking - https://cs.cmu.edu/~awm/animations/constraint/
PSC-https://www.geeksforgeeks.org/constraint-satisfaction-problems-csp-in-artificial-intelligence/
Euristica optimizare - https://web.pdx.edu/~arhodes/ai12.pdf

9. O listă cu ce a lucrat fiecare membru al echipei

Chelea Diana-Maria:

- ➔ funcția forward checking
- ➔ funcția display result cu interfața tkinter
- ➔ funcția main
- ➔ Documentație punctele 1, 3, 5, 7
- ➔ teste funcție update și jumătate de la cea_mai_constransa_variabilă

Spiridon Bianca:

- ➔ clasa Coordonata
- ➔ funcția de euristica cea mai constransa variabila
- ➔ funcția update poziții
- ➔ Documentație punctele 2, 4, 6, 8
- ➔ teste funcție forward_checking și jumătate de la cea_mai_constransa_variabilă