Sudoku – Algoritmus és Rendszer Dokumentáció

Készítette:

- ***Wéber Dominik - FL6KPT***
- ***Szappanos Szabolcs - VC2QXL***
- ***Marosi Róbert - E4M7VR***

A projekt célja

A projekt célja egy olyan **komplex, algoritmikus Sudoku-rendszer** kidolgozása, amely képes a Sudoku logikai feladvány teljes életciklusát önállóan kezelni: a megoldástól kezdve a feladvány generálásán át egészen a fájlalapú mentésig és visszatöltésig. A hangsúly nem a grafikus megjelenítésen, hanem a háttérben zajló **matematikai és programozási modelleken** van. Az alkalmazás célja demonstrálni, hogyan lehet egy klasszikus logikai feladatot tisztán algoritmikus módon, strukturált Python-kóddal megvalósítani.

A projekt további célja a **visszalépéses keresés (backtracking)** gyakorlati bemutatása, amely az egyik legfontosabb problémamegoldó algoritmus a mesterséges intelligenciában és a kombinatorikus keresésben. A Sudoku tökéletes példa erre, mivel jól szemlélteti, hogyan lehet egy nagy keresési teret hatékonyan bejárni, hibás próbálkozások esetén visszalépni, majd más útvonalakat kipróbálni. A program célja tehát nemcsak egy működő Sudoku, hanem egy jól átlátható és oktatási célra is kiváló algoritmusmodell létrehozása.

A rendszer harmadik célja egy **egyedi megoldású feladványgenerátor kialakítása**, amely garantálja, hogy minden létrehozott Sudoku puzzle megfelel a szakmai követelményeknek: egyrészt helyesen van felépítve, másrészt pontosan egy megoldása van. Ez a Sudoku-generálás egyik legösszetettebb része, ezért a projekt célkitűzései között szerepelt ennek megbízható és programozottan korrekt megvalósítása.

A projekt negyedik célja a Sudoku táblák és játékállapotok **fájlrendszerben történő tárolásának** kidolgozása. Ez lehetővé teszi a feladványok bővíthetőségét, új Sudoku-k hozzáadását, és a felhasználói állapotok későbbi visszatöltését. A fájlkezelés ezen formája a program tartósságát és bővíthetőségét biztosítja, amely fontos szempont minden komolyabb alkalmazásnál.

Adatmodell – A Sudoku tábla

A Sudoku tábla a program központi adatszerkezete, amely egy **9×9-es kétdimenziós lista (mátrix)** formájában kerül tárolásra. A program minden logikai művelete – legyen az megoldás, generálás, ellenőrzés vagy mentés – ezen az adatszerkezeten keresztül történik. A kétdimenziós lista minden eleme egy-egy mezőt képvisel, amelyek integer típusú értékeket tartalmaznak.

A Sudoku szabályai szerint minden mező 1 és 9 közötti számot vehet fel. A még nem kitöltött mezőket a program **0 értékkel** jelöli. Ez a választás praktikus, mert a 0 kívül esik a Sudoku megengedett értékein, így egyértelműen és gyorsan felismerhető, mely cellák üresek. Ennek köszönhetően a különböző algoritmusok – például az üres mező keresése vagy a backtracking megoldó – könnyedén beazonosíthatják a további feldolgozást igénylő pozíciókat.

Az adatmodell tehát így épül fel:

```
board = [
    [0, 0, 3, 0, 2, 0, 6, 0, 0],
    [9, 0, 0, 3, 0, 5, 0, 0, 1],
    [0, 0, 1, 8, 0, 6, 4, 0, 0],
    ...
]
```

A board[row][col] hivatkozási forma lehetővé teszi, hogy a program gyorsan elérje bármelyik mezőt. A sorok (row) és oszlopok (col) indexei 0-tól 8-ig tartanak, így a Sudoku belső szerkezete tökéletesen illeszkedik a Python listaindexeléséhez.

Az adatmodell nagy előnye a **rugalmasság** és a **könnyű manipulálhatóság**. Az algoritmusok egyszerűen járhatják be soronként, oszloponként vagy 3×3-as blokkok szerint a táblát – ezek mind kulcsfontosságú műveletek a Sudoku ellenőrzésében és megoldásában. A kétdimenziós listaformátum emellett lehetővé teszi, hogy a tábla könnyen összefűzhető legyen fájlmentés előtt (pl. 81 karakter hosszú string formájában), vagy akár visszaalakítható legyen mentésből egy újabb 9×9-es struktúrává.

Üres mező keresése

A Sudoku megoldásának egyik legfontosabb alapművelete az első üres mező megtalálása. Az üres mezőket a program a tábla adatmodelljének megfelelően **0 értékkel** jelöli. A megoldó algoritmus (backtracking) minden egyes lépésénél szükséges meghatározni, hol van a következő olyan pozíció, amely még kitöltésre vár.

A függvény teljes kódja:

```
def üres_mező_keresése(board):
    for r in range(9):
        for c in range(9):
            if board[r][c] == 0:
                return r, c
    return None
```

Kétszintű for-ciklus
A függvény két egymásba ágyazott ciklust használ:

- az első a sorokon (r) megy végig 0-tól 8-ig,
- a második az oszlopokon (c) halad 0-tól 8-ig.

Minden mező ellenőrzése
A board[r][c] kifejezés segítségével minden cellát egyenként vizsgál.
Ha a cella értéke 0, akkor az adott pozíció **üres mezőnek minősül**.

Az első üres mező visszaadása
Amint talál egy 0 értéket, azonnal visszaadja a mező sor–oszlop helyét:

```
return r, c
```

1. Ez hatékony, mert nem kell végignéznie a teljes táblát, ha az első sorban vagy oszlopban található egy üres hely.
2. **Ha nem talál üres mezőt**
   Ebben az esetben a függvény None értékkel tér vissza:

```
return None
```

Ez azt jelenti, hogy **a Sudoku teljesen ki van töltve**, és a megoldó algoritmus számára ez egy jelzés arra, hogy a Sudoku kész.

Szám elhelyezhetősége

A Sudoku megoldásának egyik alapköve annak eldöntése, hogy egy adott mezőbe **szabályosan elhelyezhető-e** egy szám. A Sudoku szabályai három nagy csoportba sorolhatók:

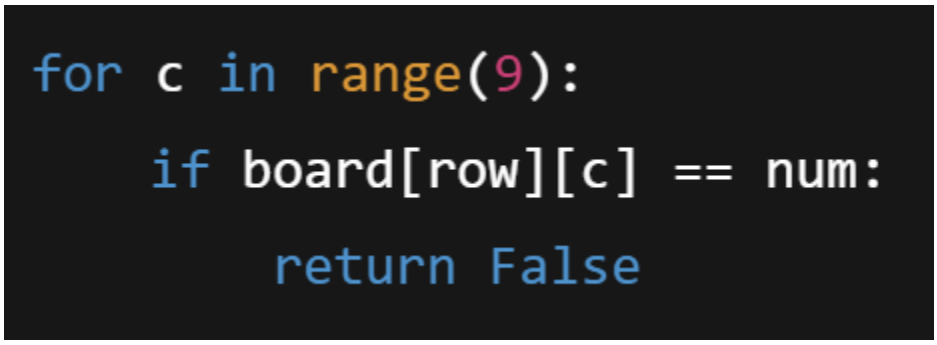
1. Egy sorban minden szám csak egyszer szerepelhet.
2. Egy oszlopban minden szám csak egyszer szerepelhet.
3. Minden 3×3-as blokkban minden szám csak egyszer szerepelhet.

A következő függvény pontosan ezeket a szabályokat ellenőrzi:

```
def szám_elhelyezhető(board, row, col, num):
    for c in range(9):
        if board[row][c] == num:
            return False
    for r in range(9):
        if board[r][col] == num:
            return False
    start_row = (row // 3) * 3
    start_col = (col // 3) * 3
    for r in range(start_row, start_row + 3):
        for c in range(start_col, start_col + 3):
            if board[r][c] == num:
                return False
    return True
```

A sor ellenőrzése:

A függvény első ciklusa végigmegy a kiválasztott soron (row), és megvizsgálja, szerepel-e benne már a num érték:



Az oszlop ellenőrzése:

A második ciklus ugyanígy működik, csak az oszlopokra alkalmazva:

```
for r in range(9):
    if board[r][col] == num:
        return False
```

A 3×3-as blokk meghatározása:

A Sudoku tábla 9 darab 3×3-as blokkra van felosztva. A függvény kiszámítja, hogy a mező melyik blokkba tartozik:

```
start_row = (row // 3) * 3
start_col = (col // 3) * 3
```

Sudoku megoldása – Backtracking

A Sudoku megoldásának központi eleme a **visszalépéses keresés** (backtracking) algoritmus. Ez egy olyan általános problémamegoldó módszer, amely lépésről lépésre próbálja felépíteni a megoldást, és ha egy ponton zsákutcába fut, akkor visszalép az előző állapothoz, és más lehetőséget próbál.

A függvény kódja:

```
def sudoku_megoldása(board):
    üres = üres_mező_keresése(board)
    if üres is None:
        return True
    row, col = üres
    for num in range(1, 10):
        if szám_elhelyezhető(board, row, col, num):
            board[row][col] = num
            if sudoku_megoldása(board):
                return True
            board[row][col] = 0
    return False
```

Következő üres mező keresése:

```
üres = üres_mező_keresése(board)
if üres is None:
    return True
```

A függvény első lépésben meghívja az üres_mező_keresése(board) függvényt. Ha az visszaad egy pozíciót ((row, col)), akkor még **van üres mező**, tehát a Sudoku **nincs kész**. Ha None-t ad vissza, akkor nincs több üres mező → a tábla **teljesen kitöltött, és mivel mindig szabályosan töltünk, ez egy kész megoldás**.
Ezért ilyenkor True-val tér vissza.

Koordináták szétbontása:

```
row, col = üres
```

- A üres változó egy (sor, oszlop) tuple.
- Ez a mező lesz a következő, amelyre megpróbálunk számot tenni.

Ezzel jelöljük ki azt a **konkrét cellát**, amin a következő lépésben dolgozni fogunk.

Számok végigpróbálása 1-től 9-ig:

```
for num in range(1, 10):
    if szám_elhelyezhető(board, row, col, num):
        board[row][col] = num
```

- A ciklus végigmegy az összes lehetséges számon: 1, 2, ..., 9.
- Minden számra ellenőrzi a szám_elhelyezhető függvénnyel, hogy szabályos lenne-e:
    - nem ismétlődik a sorban,
    - nem ismétlődik az oszlopban,
    - nem ismétlődik a 3×3-as blokkban.

Ha egy szám megfelel a szabályoknak:

- **ideiglenesen beírjuk** a táblába: board[row][col] = num.

Rekurzív hívás:

```
if sudoku_megoldása(board):
    return True
```

Miután beírtunk egy számot, meghívjuk **ugyanezt a függvényt** újra, ugyanazzal a táblával.

Az új hívás már a **következő üres mezőt** fogja keresni, és azon dolgozni.

Ha a rekurzív hívás True-t ad vissza, az azt jelenti:

- **onnantól lefelé mindent sikerült szabályosan kitölteni**, tehát a teljes Sudoku megoldódott.

Visszalépés (backtrack), ha zsákutca van:

```
board[row][col] = 0
```

Ez a sor **csak akkor fut le**, ha:

- egy adott num beírása után a rekurzív hívás **nem** tudott megoldást találni (nem lett True),
- vagyis: ez a num érték hosszú távon zsákutcához vezetett.

Mit csinálunk ilyenkor?

- **Visszaállítjuk** a mezőt üresre (0), vagyis „töröljük a próbálkozást".
- Ezt hívjuk **visszalépésnek** (backtracking):
  → „Ez a szám nem működött, próbáljuk a következőt."

A ciklus ezután tovább megy a következő num értékre (pl. 5 helyett 6-ot próbál).

Teljes tábla generálása

A generál_kitöltött_táblát függvény feladata, hogy egy **teljesen kitöltött, szabályos Sudoku táblát** hozzon létre a semmiből. Ez a megoldott tábla később alapul szolgál a feladvány generálásához.

A függvény kódja:

```
def generál_kitöltött_táblát():
    board = [[0] * 9 for _ in range(9)]
    def backtrack_fill():
        üres = üres_mező_keresése(board)
        if üres is None:
            return True
        r, c = üres
        számok = list(range(1, 10))
        random.shuffle(számok)
        for num in számok:
            if szám_elhelyezhető(board, r, c, num):
                board[r][c] = num
                if backtrack_fill():
                    return True
                board[r][c] = 0
        return False
    backtrack_fill()
    return board
```

1. Üres tábla létrehozása
2. A belső backtrack_fill függvény
3. Számok véletlensorrendű kipróbálása
4. Szám elhelyezése és továbbhaladás

Összegzés:

A generál_kitöltött_táblát függvény:

- létrehoz egy üres 9×9-es Sudoku táblát,
- backtracking segítségével **teljesen kitölti azt**,
- véletlenszerűen próbálja a számokat, így változatos megoldott táblákat hoz létre,
- a generált tábla minden esetben **szabályos és teljes**.

Ez a Sudoku motor „alapja”, amire a későbbi feladványgenerálás épül.

Feladvány generálása

Ez a függvény felel azért, hogy egy **teljesen megoldott Sudoku táblából** olyan feladványt készítsen, amelyben már vannak üres mezők, de a feladványnak **továbbra is pontosan egy**

megoldása marad. Ez nagyon fontos tulajdonság: egy „jó” Sudoku-feladvány nem többértelmű, mindig egyértelműen visszafejthető.

A függvény kódja:

```
def készít_feladvány_megoldásból(megoldott, fokozat):
    if fokozat == "könnyű":
        cél_üres = 40
    elif fokozat == "közepes":
        cél_üres = 50
    else:
        cél_üres = 60
    puzzle = tábla_másolása(megoldott)
    pozíciók = [(r, c) for r in range(9) for c in range(9)]
    random.shuffle(pozíciók)
    eltávolítva = 0
    for r, c in pozíciók:
        if eltávolítva >= cél_üres:
            break
        temp = puzzle[r][c]
        puzzle[r][c] = 0
        másolat = tábla_másolása(puzzle)
        db = megoldások_száma(másolat, limit=2)
        if db == 1:
            eltávolítva += 1
        else:
            puzzle[r][c] = temp
    return puzzle
```

Nehézségi szint és üres mezők száma:

Itt határozza meg a függvény, hogy **körülbelül hány mezőt szeretnénk üresen hagyni**, a választott nehézségi szint alapján:

- *könnyű*: kb. 40 üres mező
- *közepes*: kb. 50 üres mező
- *nehéz*: kb. 60 üres mező

Minél több az üres mező, általában annál nehezebb a feladvány, mert kevesebb kiindulási információ áll rendelkezésre.

A megoldott tábla másolása:

- A függvény **nem közvetlenül a megoldott táblát módosítja**, hanem készít róla egy másolatot.
- A puzzle lesz az a tábla, amelyből számokat fogunk kitörölni.

- A megoldott tábla **változatlanul megmarad**, hogy referenciaként használható legyen, és tudjuk:
  „ez az eredeti, helyes megoldás”.

Pozíciók véletlenszerű sorrendbe keverése:

- Létrejön egy lista az összes mező pozíciójáról: (0,0), (0,1), ..., (8,8).
- Ezt a listát random.shuffle segítségével **véletlenszerűen összekeverjük**.
- Így a törlések:
  - nem soronként történnek,
  - nem „szépen rendezett sorrendben”,
  - hanem össze-vissza a táblában → ettől lesz **változatosabb** a feladvány.

Számok eltávolítása a táblából:

Az eltávolítva változó számolja, hány mezőt sikerült eddig üressé tenni.

A ciklus végigmegy a véletlen sorrendű pozíciók listán.

Ha elértük a cél üresmező-számot (cél_üres), akkor break → leállunk.

Unikális megoldás ellenőrzése:

- A jelenlegi puzzle állapotról készül egy **másolat**.
- A megoldások_száma függvény segítségével megnézzük:
  - **hány megoldása van** ennek a táblának.

A limit=2 paraméter miatt a függvény:

- nem számol végtelen sokáig,
- csak azt figyeli, hogy:
  - 0 megoldás,
  - 1 megoldás,
  - vagy **legalább 2 megoldás** van.
- **Csak olyan feladvány jó**, amelynek **pontosan egyetlen megoldása van** → db == 1.

Eldöntés: maradhat-e üresen a mező?

Ha a db == 1, akkor a szám törlése ellenére a feladványnak **még mindig egyedi megoldása van**:

- ebben az esetben a mező maradhat üresen,
- növeljük az eltávolítva számlálót.

Ha db != 1 (azaz 0 vagy legalább 2 megoldás van):

- a törlés tönkretette a feladványt: vagy megoldhatatlanná tette, vagy több lehetséges megoldást eredményez,
- ezért **visszaállítjuk** az eredeti értéket:

Így biztosítjuk, hogy a készülő feladvány **mindvégig helyes marad**, és nem veszti el az egyértelműségét.

Megoldások számolása

Ez a függvény azt vizsgálja, hogy egy adott Sudoku tábla **hány különböző megoldással rendelkezik**.
Ennek a szerepe nagyon fontos a feladványgenerálásban: csak akkor tekintünk egy Sudoku feladványt „helyesnek", ha **pontosan egy megoldása van**. Ha több megoldás is létezik, akkor a feladvány nem egyértelmű, ami minőségromlást jelent.

A függvény kódja:

```
def megoldások_száma(board, limit=2):
    számláló = [0]
    def backtrack():
        if számláló[0] >= limit:
            return
        üres = üres_mező_keresése(board)
        if üres is None:
            számláló[0] += 1
            return
        row, col = üres
        for num in range(1, 10):
            if szám_elhelyezhető(board, row, col, num):
                board[row][col] = num
                backtrack()
                board[row][col] = 0
    backtrack()
    return számláló[0]
```

A megoldások_száma nem csak „vakon" meg akarja oldani a Sudokut, hanem azt szeretné megtudni, hogy:

- 0 megoldása van-e → hibás vagy befejezetlen feladvány
- 1 megoldása van-e → szép, egyedi megoldású feladvány
- legalább 2 megoldása van-e → a feladvány NEM egyedi

Ezért van a limit paraméter, amelynek alapértéke 2.
Mert ha már kettő megoldást találtunk, akkor teljesen mindegy, hogy van-e 3., 4., 5. stb. – a feladvány már nem egyedi.

Összegzés:

A megoldások_száma függvény:

- egy **backtracking-alapú megoldásszámláló**,
- amely **nem áll meg az első megoldásnál**,
- de **nem is járja be az összeset feleslegesen**, mert a limit leállítja,
- a feladvány-generálás minőségbiztosításának kulcseleme,
- garantálja, hogy a kész Sudoku feladvány **egyedi megoldású** legyen.