**به نام خدا**

---

# پیاده سازی بازی سودوکو


## دانشگاه صنعتی امیرکبیر

## دانشکده ریاضی و علوم کامپیوتر

## استاد: دکتر مهدی قطعی

## نام دانشجو: امیر حسین رستمی کنگرلو

## شماره دانشجویی:9813016

SUDOKU

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

ANSWER

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

## مقدمه

سودوکو نوعی بازی پازلی است که در یک صفحه N*N ، اعداد 1 تا N را طوری در هر سطر و ستون ها بچینیم که قوانین بازی نقض نشود. این بازی را با استفاده از هوش مصنوعی و مسائل ارضای محدودیت و الگوریتم های آن پیاده سازی کرده ایم.در این بازی برای N های بزرگ مثل 16 یا 25 یا اعداد بزرگتر از آن ، مدت زمان حل آن توسط کامپیوتر افزایش می یابد. اما برای N=9 یا N=4 ، زمان اجرای قابل قبولی دارد.

**توضیح کلی نحوه پیاده سازی**

پروژه دارای دو فایل ( Sudoku.py ,CSP.py ) است :

- Sudoku.py : با اجرای این فایل و تابع Main() ، کانفیگ اولیه بازی انجام میشود. در واقع کاربر با وارد کردن ورودی n ، ابعاد جدول بازی را مشخص میکند (عدد ورودی برای n،باید حتما جذر داشته باشد). برنامه با گرفتن n ، جدول بازی را میسازد که جدول بازی در یک ماتریس n*n ذخیره میشود و مقدار اولیه هر خانه برابر صفر است. سپس Domain هر متغیر که لیستی از اعداد 1تا n هست ، ساخته میشود.

  و سپس یک عدد ورودی دیگر( C) از کاربر گرفته میشود که مشخص میکند چه تعدادی از متغییر های جدول توسط کاربر پر شود. با مقدار دهی c به تعداد c تا متغییر توسط کابر مقدار دهی میشود و پس از مقدار دهی اولیه جدول بازی ، سیستم شروع به حل جدول میکند و اگر غیر قابل حل باشد ، سیستم پیام **(Opps UnSolveable!!)** را چاپ میکند و بازی تمام میشود و در غیر اینصورت نتیجه نهایی حل جدول چاپ میشود که کد مربوط به آن در شکل صفحه بعد آمده است.

```
def Main():
    n=int( input("n : "))
    c=int(input("number of cells you want to fill : "))
    Variables=[[0 for j in range(n)] for i in range(n)]
    Domain=[[[d for d in range(1,n+1)] for i in range(n) ] for j in range(n)]
    initialValidation=True
    for i in range(c):
        filledcell=input("row ,col,value :").split(",")
        row=int(filledcell[0])
        col=int(filledcell[1])
        value=int(filledcell[2])
        Variables[row][col]=value
        if IsConsistent(row,col,value,Variables) ==False:
            initialValidation=False

    print("Your initial Board : ")
    printBoard(Variables)
    input("Enter any key to Solve...")
    if initialValidation:
        result=Backtrack_FC(Variables, Domain)
        if result :
            print("!!!!Solved Successfully\nYour Board :\n")
            printBoard(Variables)
        else:
            print("\n!!!Oooops Unsolveable ")


    else:
        print(" !!!Oooops Unsolveable\n Cells You Filled Are InConsistent ")
```

## 1.Main() Function

- CSP.py : این فایل شامل چند تابع است که برای مسائل ارضای محدودیت پیاده سازی شده اند که شامل توابع ( **IsConsitent** ، **Backtrack_FC** ، **Select_Best_UnAssigned_Variable** ، **IsGameOver**، **GetDomain** ، **PrintBoard** و **ForwardChecking** ) میباشد. در ادامه هر کدام از توابع پیاده سازی شده در فایل CSP.py بررسی میشوند.

## بررسی توابع پیاده سازی شده

**IsConsistent(row,col,value,board) :**

این تابع وظیفه بررسی سازگاری یک متغییر و انتساب انجام شده برای آن یعنی value ، را دارد. در بازی سودوکو سه نوع محدودیت داریم :

**1)** محدودیت سطری : در این محدودیت به ازای هر متغییری که مقدار دهی میشود ،value نباید در سطر row وجود داشته باشد که پیاده سازی ان یه شکل زیر است :

```
for j in range(n):
    if board[row][j] == value and j !=col:
        return False
```

**2)** محدودیت ستونی : در این محدودیت به ازای هر متغییری که مقدار دهی میشود ،value نباید در ستون col ، وجود داشته باشد که پیاده سازی آن یه شکل زیر است :

```
for i in range(n):
    if board[i][col] == value and i !=row :
        return False
```

**3)** محدودیت داخل زیر مربع ها : در این محدودیت به هر متغییری که مقدار دهی میشود ، value نباید در داخل زیر مربع خودش که جدول رادیکال n در رادیکالn است، وجود داشته باشد.

```
# check  SubBox
    n_subBox=int(math.sqrt(n))
    box_row = (row // n_subBox) * n_subBox
    box_col= (col // n_subBox) * n_subBox
    for i in range(n_subBox):
        for j in range(n_subBox):
            if board[box_row + i][box_col + j] == value and row!=box_row + i and col!=box_col + j:
                return False
    return True
```

- در نتیجه اگر یکی از سه محدودیت بالا برای متغییر برقرار باشد ، تابع مقدار False را برمیگرداند.

## Select_Best_UnAssigned_Variable(board,domain)

این تابع با بررسی board (یا همان variables ) ، متغییری را که هنوز مقدار دهی نشده است ، انتخاب کرده و برمیگرداند. در واقع الگوریتم آن اینگونه است که از ابتدا شروع به جستجو در board میکنیم و اگر متغییری دارای مقدار صفر باشد ، آن را

برمیگردانیم.(البته میتوانستیم از توابع هیوریستیک نظیر MRV هم برای یافتن بهترین متغییر برای مقدار دهی ، استفاده کنیم ) . کد مربوط به آن به شکل زیر است:

```
def Select_Best_UnAssigned_Variable(board,domain):
        n=len(board)
        for i in range(n):
            for j in range(n):
                if board[i][j]==0:
                    return i,j
        return None,None
```

**GetDomain(i,j,domain) :**

این تابع به ازای هر متغییر ، دامنه مربوط به آنرا برمیگرداند که اعضای دامنه به صورت صعودی مرتب شده اند.

```
def GetDomain(i,j,domain):
    domain[i][j].sort()
    return domain[i][j]
```

**PrintBoard(board) :** این تابع جدول بازی را می گیرد و جدول را چاپ میکند.

```
def printBoard(board):
        n=len(board)
        for i in range (n):
            print(board[i])
        print("**********\n\n")
```

## ForwardChecking(board ,domain) :

Forward-checking یکی از روش های فیلترینگ در مسائل ارضای محدود است که در این پروژه از این روش برای فیلتر کردن دامنه متغییر ها استفاده میکنیم که باعث افزایش سرعت اجرای بازی شده و تعداد عقب گردی هایمان در جستجوی Backtracking ، کاهش می یابد که الگوریتم آن بدین شرح است :

1. مقدار دهی اولیه Removedvalues= []
2. به ازای هر value در دامنه متغیر ها اگر آن value برای آن متغییر در جدول بازی سازگار نباشد ، آن Value را از دامنه آن متغییر حذف کرده و به Removedvalues اضافه میکنیم.
3. اگر موقع حذف کردن value از دامنه متغییر ، دامنه آن متغییر خالی شود ، آنگاه False را با Removedvalues برمیگردانیم که به معنای ناموفق بودن عملیات فیلترینگ است.
4. در آخر نیز مقدار removeValues , True را برمیگردانیم.

کد مربوط به Forward-Checking در زیر امده است:

```
def ForwardChecking(board:list[list],domain:list[list[list]])->(bool,list,list):
    n=len(board)
    removedvalues=[]
    for i in range(n):
        for j in range(n):
            for value in domain[i][j]:
                if IsConsistent(i, j, value, board)==False:
                    domain[i][j].remove(value)
                    removedvalues.append((i,j,value))
                    if len(domain[i][j])==0:
                        return False,removedvalues
    return True,removedvalues
```

## RestoreRemovedValues(domain, Removedvalues) :

این تابع مقادیری را که از دامنه متغییر ها توسط الگوریتم Forward-checking حذف میشوند را برمیگرداند.

```
def RestoreRemovedValues(domain:list[list[list]],removedValues:list,n:int):

    for item in removedValues:
        domain[item[0]][item[1]].append(item[2])
    for i in range(n):
        for j in range(n):
            domain[i][j].sort()
```

Item[0] و item[1] و item[2] در اینجا به ترتیب همان شماره سطر و ستون و مقدار هر متغییر میباشد که در RemovedValues ذخیره شده است.

## Backtrack-FC(board,domain) :

این تابع همان جستجوی عقب گرد است که برای حل جدول از این الگوریتم استفاده میشود و در این الگوریتم از Forward-Checking هم استفاده شده است.

الگوریتم کلی آن بدین شرح است:

1. ابتدا چک میکنیم که اگر همه متغییر ها مقدار دهی شده باشند به معنای پایان بازی است و مقدار True را برمیگردانیم.
2. یک متغییری که هنوز پر نشده است را انتخاب میکنیم
3. به ازای هر value در دامنه آن متغییر ،اگر انتساب آن مقدار به متغییر سازگار باشد،انتساب را انجام میدهیم.
4. تابع ()Forwardchecking را فراخوانی میکنیم .
5. اگر این تابع مقدار true را برگرداند:
6. دوباره تابع Backtrack_FC(board,domain) را به شکل بازگشتی فراخوانی میکنیم
7. اگر درست بود ، دوباره یک عمق پایین تر میرویم و در غیر اینصورت به حالت قبلی برمیگردیم و مقادیر فیلتر شده در دامنه را دوباره بازیابی میکنیم.

```
def Backtrack_FC(board,domain):
        if IsGameOver(board):
            return True
        i,j=Select_Best_UnAssigned_Variable(board,domain)
        for value in GetDomain(i, j, domain):
            if IsConsistent(i,j,value,board):
                board[i][j]=value
                fc_result,removed_values=ForwardChecking(board, domain)
                if fc_result:
                    if Backtrack_FC(board,domain):
                        return True
                RestoreRemovedValues(domain, removed_values,len(board))
                board[i][j]=0
        return False
```

**: Backtrack(board,domain)**

این تابع دقیقا مثل تابع ()Backtrack-FC عمل میکند اما با این تفاوت که در این تابع ازForward-checking استفاده نشده است.

```
def Backtrack(board,domain):
    if IsGameOver(board):
        return True
    i,j=Select_Best_UnAssigned_Variable(board,domain)
    for value in GetDomain(i, j, domain):
        if IsConsistent(i,j,value,board):
            board[i][j]=value
            if Backtrack(board,domain):
                return True

            board[i][j]=0
    return False
```

**تست برنامه :**

نمونه جدول های زیر را داریم و میخواهیم برنامه را به ازای اینها تست کنیم:

```
board1=[[9,0,0,0,0,0,2,0,0],
        [0,8,0,0,0,7,0,9,0],
        [6,0,2,0,0,0,5,0,0],
        [0,7,0,0,6,0,0,0,0],
        [0,0,0,9,0,1,0,0,0],
        [0,0,0,0,2,0,0,4,0],
        [0,0,5,0,0,0,6,0,3],
        [0,9,0,4,0,0,0,7,0],
        [0,0,6,0,0,0,0,0,0]
       ]
board2=[
    [0,4,0,1],
    [3,0,4,0],
    [1,0,0,4],
    [0,2,1,0]
    ]
```

برنامه را یکبار برای Backtrack و یکبار نیز برای Backtrack-FC اجرا میکنیم:

- به ازای n=4 , board=board2 :

```
def MainTest1():
    n=4
    Variables=[[0 for j in range(n)] for i in range(n)]
    Domain=[[[d for d in range(1,n+1)] for i in range(n) ] for j in range(n)]
    Variables=board2
    result=Backtrack_FC(Variables,Domain)
    print("result : ",result)
    printBoard(Variables)
```

نتیجه: برای board2 ، در زمان بسیار مناسبی حل شد.  اگر با Backtracking هم حل کنیم ، در زمان سریعی به جواب می رسیم.

```
result :  True
[2, 4, 3, 1]
[3, 1, 4, 2]
[1, 3, 2, 4]
[4, 2, 1, 3]
**********
```

- **به ازای n=9 , board=board1 و با استفاده از تابع Backtracking-FC() :**

```
def MainTest2():
    n=9
    Variables=[[0 for j in range(n)] for i in range(n)]
    Domain=[[[d for d in range(1,n+1)] for i in range(n) ] for j in range(n)]
    Variables=board1
    result=Backtrack_FC(Variables,Domain)
    print("result : ",result)
    printBoard(Variables)
```

**نتیجه:** نتیجه به درستی انجام شد ، اما زمان اجرای آن نسبتا طولانی شد و بالای 10 ثانیه بود.

```
result :  True
[9, 5, 7, 6, 1, 3, 2, 8, 4]
[4, 8, 3, 2, 5, 7, 1, 9, 6]
[6, 1, 2, 8, 4, 9, 5, 3, 7]
[1, 7, 8, 3, 6, 4, 9, 5, 2]
[5, 2, 4, 9, 7, 1, 3, 6, 8]
[3, 6, 9, 5, 2, 8, 7, 4, 1]
[8, 4, 5, 7, 9, 2, 6, 1, 3]
[2, 9, 1, 4, 3, 6, 8, 7, 5]
[7, 3, 6, 1, 8, 5, 4, 2, 9]
**********
```

- **به ازای n=9,board=board1 و با استفاده از تابع Backtracking() :**

```
def MainTest3():
    n=9
    Variables=[[0 for j in range(n)] for i in range(n)]
    Domain=[[[d for d in range(1,n+1)] for i in range(n) ] for j in range(n)]
    Variables=board1
    result=Backtrack(Variables,Domain)
    print("result : ",result)
    printBoard(Variables)
```

**نتیجه:** دقیقا مثل نتیجه تست قبل شد با این تفاوت که با استفاده از Backtrack() و بدون استفاده از Forward-Checking() ، زمان اجرای بهتری داشتیم و در کمتر از 10 ثانیه به جواب مورد نظر رسیدیم.

```
result :  True
[9, 5, 7, 6, 1, 3, 2, 8, 4]
[4, 8, 3, 2, 5, 7, 1, 9, 6]
[6, 1, 2, 8, 4, 9, 5, 3, 7]
[1, 7, 8, 3, 6, 4, 9, 5, 2]
[5, 2, 4, 9, 7, 1, 3, 6, 8]
[3, 6, 9, 5, 2, 8, 7, 4, 1]
[8, 4, 5, 7, 9, 2, 6, 1, 3]
[2, 9, 1, 4, 3, 6, 8, 7, 5]
[7, 3, 6, 1, 8, 5, 4, 2, 9]
**********
```

- با استفاده از تابع Main() هم میتوانیم برنامه را طبق ورودی ها و خروجی های خواسته شده در فایل راهنما که در بالا هم توضیح داده شده است، اجرا کنیم.

- **نکته مهم :** بدلیل اینکه در MainTest3() از توابع هیوریستیک برای انتخاب متغییر برای انتساب مقدار استفاده نکرده ایم ، و همچنین ترتیب انتخاب value از دامنه ، بهینه سازی نشده است ، زمان اجرای الگوریتم زیاد بوده و میزان عقبگرد مان برای انتساب متغییر زیاد است.

## نتیجه

برای اینکه به زمان اجرای مناسبی برسیم ، باید از توابع هیوریستیک در توابع Select-Best-UnassignedVariable() و همچنین تابع GetDomain() استفاده کنیم که میزان عقبگرد هایمان در درخت جستجوی تشکیل شده به کمترین حد برسد و سریع تر به جواب برسیم.

## منابع


- https://chat.openai.com
- AmirShahbazi(9812033)
- /https://www.geeksforgeeks.org