BackTracking

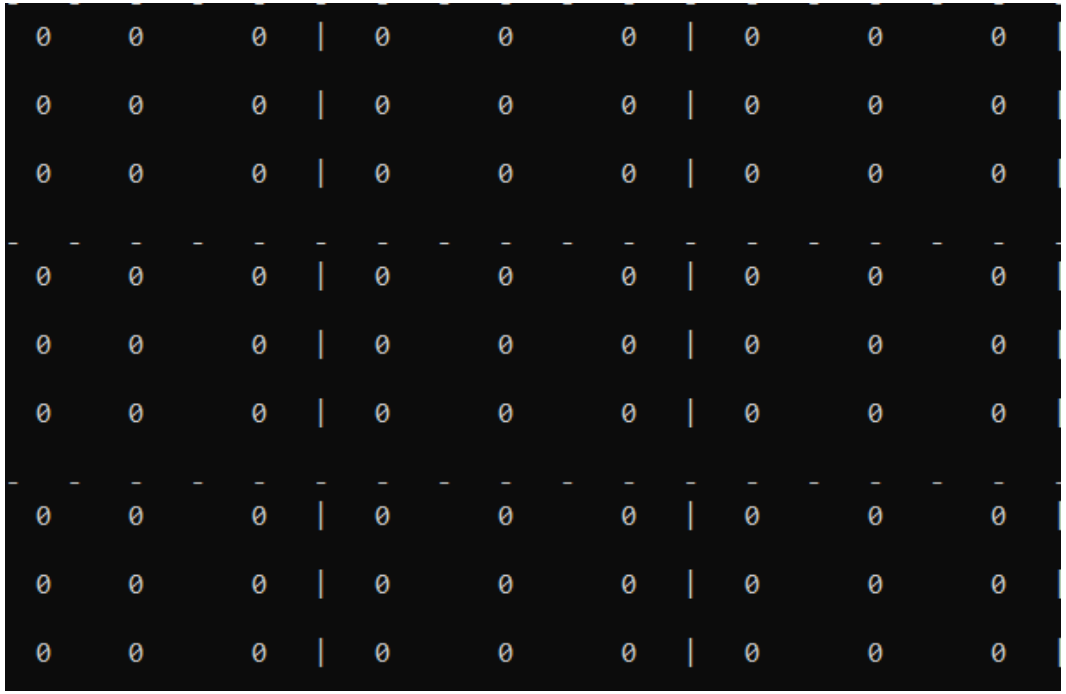
تابع اولیه:

```
def __init__(self, dim) -> None:
        self.dim = dim
        self.expandedNodes = 0
        self.board = [['0' for i in range(self.dim)] for j in range(self.dim)]
```

برای پیاده سازی سودوکو ابتدا `dimension` آن را مشخص می کنیم(قائدتا مضربی از ۳ می باشد؛ برای مثال `dim = 9` در نظر میگیریم)

برای شروع `expandedNodes` برابر صفر می باشد.

همچنیم هنگام شروع صفحه سودوکو یک صفحه (آرایه دو بعدی) خالی از اعداد می باشد که این را با `'0'` قرار دادن در تمام خانه ها نشان میدهیم.



BackTracking

تابع getNextLocation :

```
def getNextLocation(self):
        for x in range(self.dim):
            for y in range(self.dim):
                if self.board[x][y] == '0':
                    return (x, y)
        return (-1, -1)
```

این تابع از بالا و چپ ترین خانه از سودوکو شروع میکند و اولین خانه ی خالی را بر میگرداند تا بعدا برای پر کردن این خانه تصمیم گیری شود.

در صورتی که تمام خانه ها پر شده باشند این تابع مختصات `(-1, -1)` برمیگرداند؛ که بعدا از این موضوع میتوان برای تشخیص پر بودن خانه های صفحه سودوکو استفاده کرد.

BackTracking

تابع isSafe :

```
def isSafe(self, x, y, value):
        for i in range(self.dim):
            if self.board[x][i] == str(value):
                return False
        for i in range(self.dim):
            if self.board[i][y] == str(value):
                return False
        boxRow = x - x%3
        boxCol = y - y%3
        for i in range(3):
            for j in range(3):
                if self.board[boxRow + i][boxCol + j] == str(value):
                    return False
        return True
```

BackTracking

تابع isSafe :

این تابع از سه لوپ تشکیل شده است؛

- لوپ اول مشخص میکند مقدار داده شده برای قرارگیری داخل خانه ی مورد نظر، آیا قبلا داخل آن سطر می باشد یا نه.
- لوپ دوم مانند لوپ اول می باشد ولی این کار را برای ستونی که خانه در آن قرار گرفته است بررسی میکند.
- قبل از لوپ سوم ابتدا اولین خانه از مکعبی که خانه مورد نظر در آن قرار گرفته است را پیدا میکنیم و سپس بررسی میکنیم که مقدار داده شده آیا داخل ۹ خانه ی این مکعب قبلا قرار گرفته است یا نه.

در صورتی که یکی از شرط های داخل لوپ های بالا برقرار باشد یعنی آن مقدار برای این خانه مناسب نیست در نتیجه تابع `False` برمیگرداند در غیر این صورت یعنی این مقدار می تواند داخل این خانه قرار بگیرد.

BackTracking

تابع solveSimpleBackTracking :

```
def solveSimpleBackTracking(self):
        location = self.getNextLocation()
        x = location[0]
        y = location[1]
        if x == -1:
            return True
        else:
            self.expandedNodes += 1
            for choice in range(1, self.dim + 1):
                if self.isSafe(x, y, choice):
                    self.board[x][y] = str(choice)
                    if self.solveSimpleBackTracking():
                        return True
                    self.board[x][y] = '0'
        return False
```

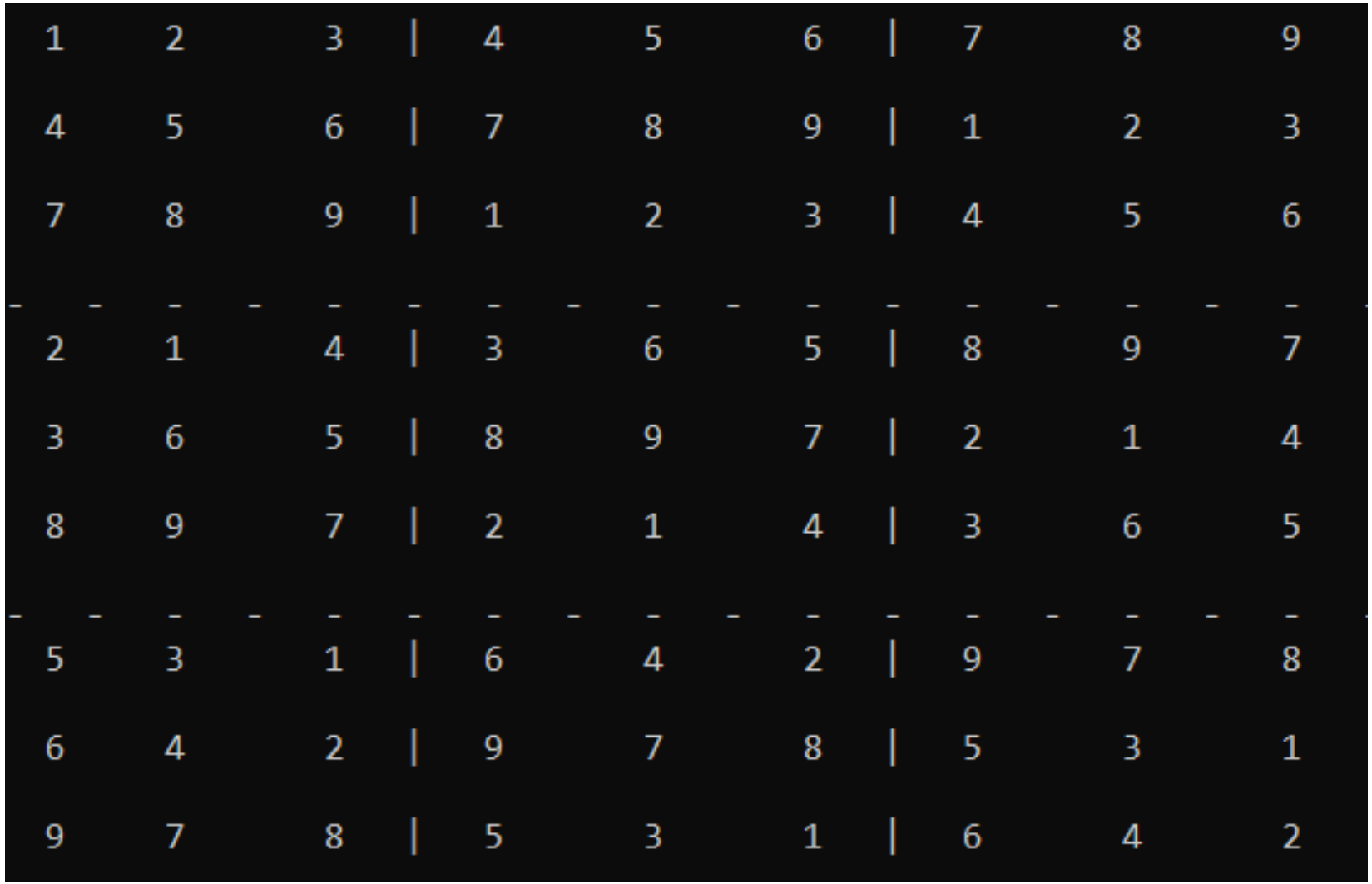
BackTracking

تابع solveSimpleBackTracking :

این تابع در هر مرحله که صدا زده میشود ابتدا اولین خانه ی خالی را پیدا میکند. اگر `(-1, -1)` برگردانده شود یعنی تمام خانه ها به طور کامل و درست پر شده اند و تمام `constraint` های سودوکو برقرار می باشد و `True` برمیگرداند.

اگر هنوز خانه ای خالی باشد ابتدا به تعداد `expandedNodes` یکی اضافه میکند و سپس برای تمام اعداد بازه ی `[1, 9]` با صدا زدن تابع `isSafe` چک میکند که آیا این مقدار برای خانه ی خالی ای که داریم مناسب می باشد یا نه؛ در صورتی که مناسب باشد (با صدا زدن دوباره خود تابع) این مقدار را داخل این خانه قرار میدهد و با صدا زدن دوباره `solveSimpleBackTracking` سعی میکند بقیه خانه های خالی را پر کند و اگر در این هنگام به شرایطی رسیدیم که `consistent` نباشد، این مقدار را از آن خانه پاک میکند و دوباره یک مقدار دیگر را بررسی میکند.

BackTracking



CSP BackTracking

تابع getDomain :

```
def getDomain(self, row, col):
        RVCell = [str(i) for i in range(1, self.dim + 1)]
        for i in range(self.dim):
            if self.board[row][i] != '0':
                if self.board[row][i] in RVCell:
                    RVCell.remove(self.board[row][i])
        for j in range(self.dim):
            if self.board[j][col] != '0':
                if self.board[j][col] in RVCell:
                    RVCell.remove(self.board[j][col])
        boxRow = row - row%3
        boxCol = col - col%3
        for i in range(3):
            for j in range(3):
                if self.board[boxRow + i][boxCol + j] != '0':
                    if self.board[boxRow + i][boxCol + j] in RVCell:
                        RVCell.remove(self.board[boxRow + i][boxCol + j])
        return RVCell
```

این تابع مقادیر مجاز را با توجه به وضعیت

فعلی صفحه سودوکو، برای یک خانه

مخصوص انتخاب میکند.

(منطق آن براساس تابع `isSafe` در بخش قبلی می باشد.)

CSP BackTracking

تابع getRemainingValues :

```
def getRemainingValues(self):
        RV = []
        for row in range(self.dim):
            for col in range(self.dim):
                if self.board[row][col] != '0':
                    RV.append(['x'])
                else:
                    RV.append(self.getDomain(row, col))
        return RV
```

این تابع خانه هایی از صفحه سودوکو که هنوز پر نشده اند را انتخاب میکند و مقادیر دامنه ی آن را با تابع `getDomain` به دست می آورد و داخل RV میریزد.

CSP BackTracking

تابع solveCSPBackTracking :

```
def solveCSPBackTracking(self):
        assign_all_variable = True
        for i in range(len(self.rv)):
            if self.rv[i] != ['x']:
                assign_all_variable = False
                x = int(i/9)
                y = i % 9
                domain = self.rv[i]
        if assign_all_variable:
            return True
        else:
            self.expandedNodes += 1
            for choice in domain:
                self.board[x][y] = choice
                self.rv = self.getRemainingValues()
                if self.solveCSPBackTracking():
                    return True
                self.board[x][y] = '0'
```

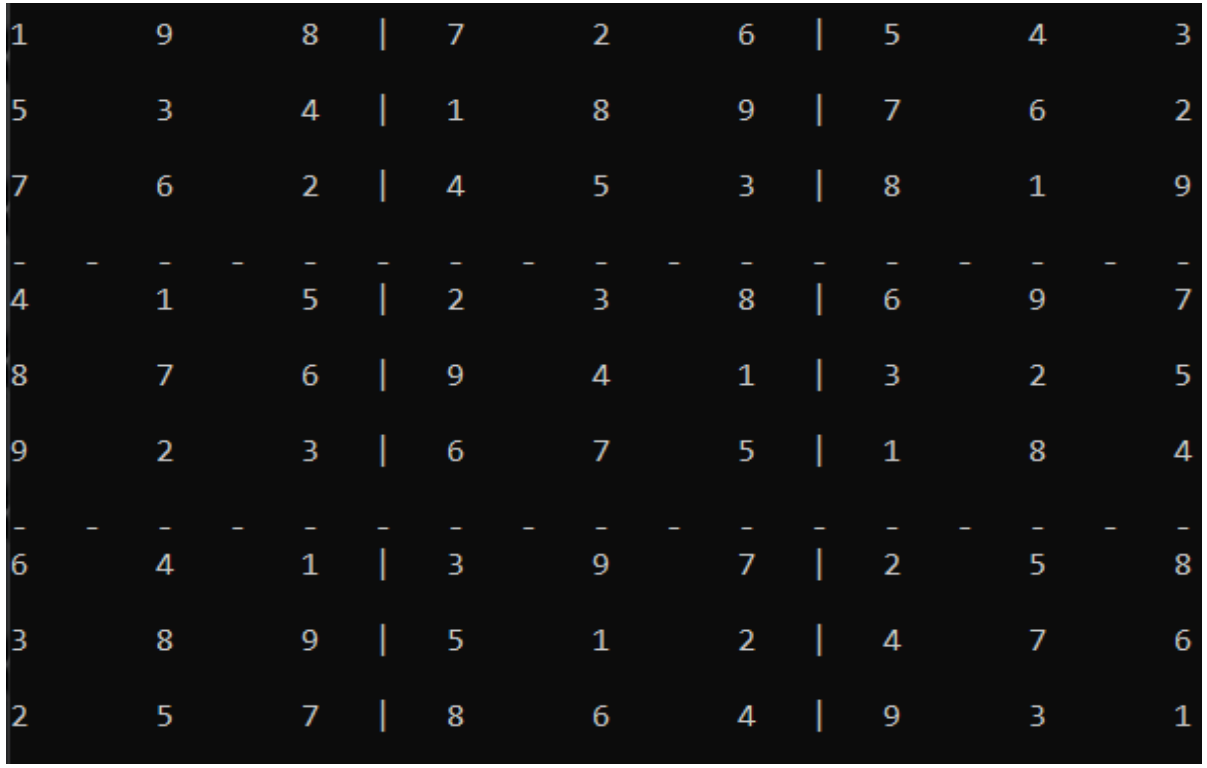
CSP BackTracking

تابع solveCSPBackTracking :

در لوپ اول یک متغیری که هنوز مقدار دهی نشده است را از self.rv انتخاب میکند.

در صورتی که تمام خانه ها مقداردهی شده باشند تابع True برمیگرداند.

در صورتی که خانه ای وجود داشته باشد که هنوز مقداردهی نشده باشد؛ ابتدا یک واحد به expandedNodes اضافه میکنیم. سپس مشابه تابع solveSimpleBackTracking عمل میکنیم ولی با این تفاوت که به جای اینکه تمام مقادیر ۱ تا ۹ بررسی کنیم، فقط مقادیر مجاز داخل دامنه ی آن خانه را چک میکنیم.



تفاوت CSP و simpleBackTracking در حالاتی می باشد که بررسی میکند.

در simpleBackTracking تمام اعداد را بررسی میکنیم ولی در CSP ابتدا دامنه خانه ها را بررسی میکنیم و سپس مقادیر دامنه را بررسی میکنیم.

برای بهبود زمانی و حافظه میتوان از روش های filtering مانند forward checking یا arc consistency و ordering استفاده کرد.

CSP BackTracking

تابع بهبود یافته solveCSPBackTracking :

```
def solveCSPBackTracking(self):
        location = self.getNextLocation()
        x = location[0]
        y = location[1]
        domain = location[2]
        if x == -1:
            return True
        elif len(domain) == 0:
            return False
        else:
            occurrence = {x : 0 for x in domain}
            for d in self.board:
                for number in d:
                    if number in domain:
                        occurrence[number] += 1
            domain.sort(key=lambda x : occurrence[x], reverse=True)
            self.expandedNodes += 1
            for choice in domain:
                self.board[x][y] = choice
                self.rv = self.getRemainingValues()
                if self.solveCSPBackTracking():
                    return True
                self.board[x][y] = '0'
```

CSP BackTracking

تابع بهبود یافته solveCSPBackTracking :

این تابع ابتدا لوکیشن بعدی را توسط تابع getNextLocation پیدا میکند. تابع getNextLocation مختصات و دامنه خانه ای را برمیگرداند که دامنه ی آن کمترین تعداد مقدار را برای بررسی دارد. (ordering)

```
occurrence = {x : 0 for x in domain}
            for d in self.board:
                for number in d:
                    if number in domain:
                        occurrence[number] += 1
            domain.sort(key=lambda x : occurrence[x], reverse=True)
```

این بخش از تابع دامنه خانه را به گونه ای میچیند که اعداد بر اساس تعداد تکرار داخل صفحه سودوکو مرتب شوند و از اعدادی برای آزمایش شروع کند که بیشترین تکرار را داخل صفحه سودوکو دارند؛ چون با این کار زمانی که consistency نقض میشود زودتر پیدا میشود و جلوی پیشروی بیشتر رو میگیرد. (filtering)

با روش های بالا و بهبود تابع، تعداد expandedNodes به طور چشمگیری کاهش پیدا میکند.