

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارش ه

پیااده‌سازی حل یک مساله تصمیم گیری با استفاده از ارضای محدودیت
(بازی Sudoku با الگوریتم Constraint Satisfaction Problem)

نام و نام خانوادگی : محمدحسین ملک‌پور

رشته تحصیلی : علوم کامپیوتر

شماره دانشجویی : ۹۶۱۱۳۴۲۵

استاد : دکتر مهدی قطعی

پست الکترونیکی : mohammadhossein.malekpour@gmail.com

فهرست

۳	الگوریتم ارضای محدودیت CSP
۴	پیاده سازی بازی Sudoku با الگوریتم CSP
۷	اجرای برنامه
۸	نتیجه گیری
۸	منابع

الگوریتم ارضای محدودیت CSP

همانطور که میدانید مسئله ها میتوانند از طریق جست وجو در فضایی از حالت ها حل شوند. این حالت ها را میتوان با ابتکارهای خاص دامنه ارزیابی، و تست کرد که آیا آنها حالت های هدف هستند یا خیر. اما از نظر الگوریتم جست وجو هر حالت یک جعبه سیاه است که ساختار داخلی آن قابل تمیز دادن نیست.

در اینجا روشی داریم که تعداد زیادی از مسئله ها را با کارایی بیشتری حل میکند. برای هر حالت، از مجموعه ای از متغیرها که هر کدام دارای یک مقدار است، استفاده میکنیم. مسئله وقتی حل میشود که هر متغیر دارای مقداری باشد که تمام محدودیت های روی آن متغیر را برآورده کند (آن محدودیت را ارضا کند). مسئله ای که به این روش توصیف میشود، مسئله ارضای محدودیت یا CSP نام دارد.

الگوریتم جست وجوی CSP از ساختار حالت استفاده میکند و به جای ابتکارهای خاص مسئله، از ابتکارهای همه منظوره برای حل مسئله های پیچیده بهره میگیرند. ایده ی اصلی این است که با مشخص کردن ترکیب هایی از متغیر/ مقدار که محدودیت ها را نقض میکنند، بخش های بزرگی از فضای جست وجو بطور همزمان حذف شوند. مسئله های ارضای محدودیت شامل سه مولفه ی X ، D ، و C است:

- X مجموعه ای از متغیرها است، $\{X_1, X_2, \dots, X_n\}$.
- D مجموعه ای از دامنه هاست، $\{D_1, D_2, \dots, D_n\}$ که برای هر متغیر یک دامنه وجود دارد.
- C مجموعه ای از محدودیت هاست که ترکیب هایی از مقادیر را مشخص میکند.

برای حل CSP لازم است یک فضای حالت و مفهوم جواب را تعریف کنیم. هر حالت در CSP توسط انتساب مقادیر به تمام یا بعضی از متغیرها بصورت $\{X_1 = v_1, X_2 = v_2, \dots\}$ تعریف میشود. انتسابی که هیچ محدودیتی را نقض نمیکند، انتساب سازگار یا معتبر نام دارد. انتساب کامل انتسابی است که در آن به هر متغیر مقداری نسبت داده شده است، و جواب CSP یک انتساب سازگار و کامل است. انتساب جزئی، انتسابی است که در آن مقادیر به بعضی از متغیرها نسبت داده میشود.

پیاده سازی بازی Sudoku با الگوریتم CSP

برای مسئله سودکوسه مولفه را به شکل زیر تعریف میکنیم:

X متغیرها: هر سلول خالی در صفحه یک متغیر است

D دامنه‌ها: برای هر سلول از صفحه یک دامنه به عنوان مجموعه ای از اعداد ۱ تا ۹ داریم

C محدودیت‌ها: در هر سطر، ستون و مربع های ۳*۳ اعداد تکراری نداریم

حال که مسئله‌ی CSP را تعریف کردیم میخواهیم الگوریتم backtracking را بهینه کنیم.

در ابتدا ما به یک آرایه از تمام دامنه های همه متغیرها نیاز داریم. به عبارت دیگر یک فضا برای نگهداری مقادیر باقیمانده برای هر متغیر مورد نیاز است. بنابراین یک اتریبیوت به نام `rv` به کلاس اضافه میکنیم. در صورتی که یک سلول شامل یک عدد ثابت باشد دامنه مقادیر ثابت در صفحه را با ['X'] جایگزین میکنم. در موارد دیگر معیارهای سودوکو را بررسی می کنم

تا مقادیر مناسب سلول را پیدا

کرده و آن را به لیست `self.rv`

اضافه کنیم.

```
class SudokuSolver:
    def __init__(self, dim, fileDir):
        self.dim = dim
        self.expandedNodes = 0
        with open(fileDir) as f:
            content = f.readlines()
            self.board = [list(x.strip()) for x in content]
            self.rv = self.getRemainingValues()

    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 i in range(self.dim):
            if self.board[i][col] != '0':
                if self.board[i][col] in RVCell:
                    RVCell.remove(self.board[i][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

    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
```

اکنون که اطلاعات برای استفاده آماده است ما باید یک سلول خالی یا به عبارت دیگر یک متغیر را به عنوان حرکت دوم انتخاب کنیم. برای این مشکل ساده ترین روش این است که ابتدا سلول ها را با دامنه های کوچک تر را پر کنیم. به عنوان مثال، اگر دامنه سلول [۳] و دامنه سلول دیگری [۱، ۲، ۹] باشد بدیهی است که پر کردن سلول با اندازه دامنه‌ی ۱ بهتر است چون این تنها انتخاب است پس قطعاً درست است.

اگر این ایده را گسترش دهیم و مقداری را از مجموعه دامنه کوچک انتخاب کنیم احتمال انتخاب مقدار مناسب زیاد است. عملکرد خود را با تابع `getNextMRVRowCol()` که MRV مخفف Minimum Remaining Value است فراخوانی کنیم. همانطور که گفتیم ما مقادیر ثابت را به عنوان 'x' علامت گذاری کردیم بنابراین ابتدا بررسی می کنیم که آیا یک سلول شامل عدد ثابت است و همچنین اگر دامنه خالی باشد ۱۰ را به عنوان تعداد زیادی از فضای مشکل برمی گردانیم تا از انتخاب دامنه خالی به عنوان mrv توسط agent جلوگیری کنیم.

```
def getDomainLength(self,lst):
    if 'x' in lst or lst == []:
        return 10
    else:
        return len(lst)

def getNextMRVRowCol(self):
    rvMap = list(map(self.getDomainLength,self.rv))
    minimum = min(rvMap)
    if minimum == 10:
        return (-1,-1)
    index = rvMap.index(minimum)
    return(index // 9, index % 9)
```

اکنون تابعی داریم که بهترین نقطه را برای پر کردن انتخاب می کند. از آنجا که دامنه ها را به صورت خطی ذخیره می کنیم، باید سطر و ستون ها را با یک عمل تقسیم و باقی مانده بر روی هر `self.rv` محاسبه کنیم. تا اینجا مسئله `backtracking` را به یک مسئله CSP تبدیل کرده ایم و تنها بهینه سازی انجام شده تغییر مکان یاب (یک تابع هیورستیک) است. این کارها نتایج ما را کمی بهبود می بخشد. حال از تکنیک خوبی به نام "Forward Checking" استفاده کنیم تا به نتایج مورد نیاز خود برسیم.

Forward Checking به سادگی چشم انداز برنامه را برای افزایش احتمال سودآوری از انتخاب خود در سطوح بالاتر درخت فراهم می کند. به عنوان مثال تصور کنید که دو سلول V_1 و V_2 در یک ردیف با دامنه های $d_1 = [1, 2]$ و $d_2 = [1]$ وجود دارند. تابع location ما V_1 را انتخاب می کند. در حالت عادی برنامه ابتدا ۱ را انتخاب می کند در نتیجه مقدار جدید $d_2 = []$ خواهد بود و در لایه عمیق تر متوجه می شود که هیچ کدام از مقادیر ممکن برای سلول با دامنه d_2 وجود ندارد و backtrack اجرا می شود.

هدف ما کاهش تعداد انبساط گره ها بود که شامل حرکت backtracking است. به عنوان یک راه حل می توانیم ابتدا با انتخاب ۱ بررسی کنیم آیا گزینه های احتمالی دیگری را از بین می برد؟ خواهیم دید که تمام مقادیر باقی مانده برای d_2 را از بین می برد. در این مرحله به ما اطلاع داده می شود که انتخاب ۱ قطعاً یک نتیجه backtracking دارد و بنابراین ۲ را به عنوان پاسخ احتمالی انتخاب می کنیم.

امکان پیاده سازی forward checking بیش از یک مرحله وجود دارد. با این حال ممکن است باعث ایجاد time overhead شود. در این حالت من ترجیح دادم روشی را اجرا کنم که با انتخاب مقدار مشخصی برای سلول، بررسی کند که آیا فرصت های دیگر سلول های موجود در

صفحه را از بین می برد. با جمع بندی کد نهایی بدین شکل است:

```
def isEmptyDomainProduced(self, row, col, choice):
    element = self.rv.pop(row*9 + col)
    if [] in self.rv:
        self.rv.insert(row*9+col, element)
        return True
    else:
        self.rv.insert(row*9+col, element)
        return False

def solveCSPFH(self):
    location = self.getNextMRVRowCol()
    if location[0] == -1:
        return True
    else:
        self.expandedNodes+=1
        # rv = self.getRemainingValues()
        row = location[0]
        col = location[1]
        for choice in self.rv[row*9+col]:
            choice_str = str(choice)
            self.board[row][col] = choice_str
            cpy = copy.deepcopy(self.rv)
            self.rv = self.getRemainingValues()

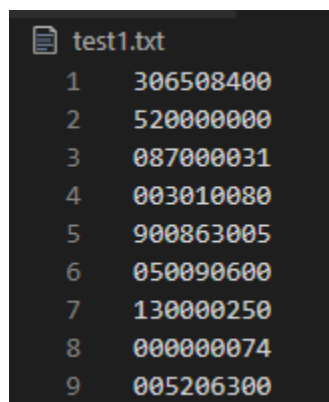
            if not self.isEmptyDomainProduced(row, col, choice_str):
                if self.solveCSPFH():
                    return True
            self.board[row][col] = '0'
            self.rv = cpy

        return False
```

اجرای برنامه

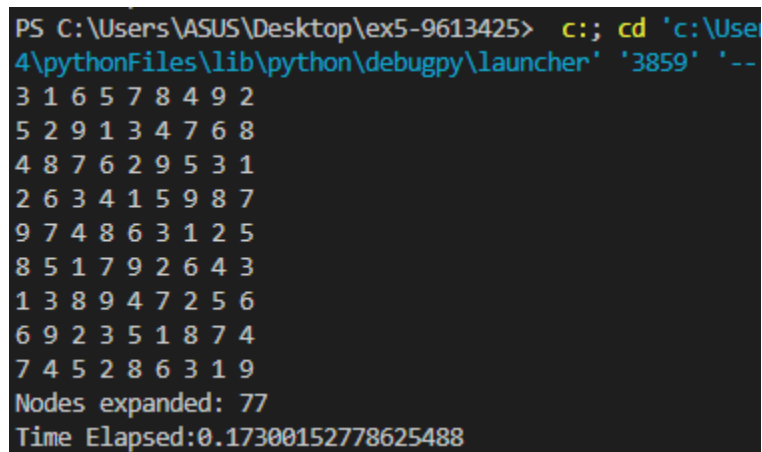
برای اجرای بازی یک تست کیس سودکو با نام test1.txt کنار برنامه داریم و سلل های خالی را با صفر مشخص کردیم. می توانید جدول سودکو دلخواهتان را در این فایل با فورمت گفته شده قرار دهید. سپس برنامه را اجرا کنید و پاسخ در ترمینال چاپ می شود.

input text file:



```
test1.txt
1 306508400
2 520000000
3 087000031
4 003010080
5 900863005
6 050090600
7 130000250
8 000000074
9 005206300
```

Output:



```
PS C:\Users\ASUS\Desktop\ex5-9613425> c:; cd 'c:\Use
4\pythonFiles\lib\python\debugpy\launcher' '3859' '--
3 1 6 5 7 8 4 9 2
5 2 9 1 3 4 7 6 8
4 8 7 6 2 9 5 3 1
2 6 3 4 1 5 9 8 7
9 7 4 8 6 3 1 2 5
8 5 1 7 9 2 6 4 3
1 3 8 9 4 7 2 5 6
6 9 2 3 5 1 8 7 4
7 4 5 2 8 6 3 1 9
Nodes expanded: 77
Time Elapsed:0.17300152778625488
```

نتیجه گیری

اگرچه الگوریتم های بازگشت به عقب یا backtracking از نظر تئوری توانایی حل هر مسئله ای را دارند، اما به فضای حافظه زیادی احتیاج دارد و زمان زیادی را مصرف می کند. الگوریتم های CSP به منظور کوچک سازی فضای زیاد و تقویت الگوریتم ها معرفی شدند. با الگوریتم های Forward Check و تابع های Heuristic حل مسئله با سرعت بالا و حافظه کم امکان پذیر است.

منابع

https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

<https://towardsdatascience.com/solving-sudoku-with-ai-d6008993c7de?gi=9b1c69dd125>

<https://levelup.gitconnected.com/csp-algorithm-vs-backtracking-sudoku-304a242f96d0>