



دانشگاه صنعتی امیرکبیر  
(پلی تکنیک تهران)  
دانشکده ریاضی و علوم کامپیوتر

## گزارش ۵: پیاده‌سازی مسئله‌ی سودوکو با روش ارضای محدودیت

نگارش

نیما حسینی دشت بیاض

استاد

دکتر مهدی قطعی

فروردین ۱۴۰۰

## مقدمه

در برخی از مسائل، با تعریف مجموعه‌ای از متغیرها، دامنه‌ها و قیود، هدف به دست آوردن مقادیری مجاز برای متغیرهاست که در قیود صدق کنند. این نوع مسئله‌ها را مسائل ارضای محدودیت یا CSP<sup>۱</sup> می‌نامند. این مسائل را مانند مسائل بهینه‌سازی می‌توان با روش‌های جستجو حل کرد؛ اما این کار باعث به‌وجود آمدن یک درخت جستجوی بسیار بزرگ می‌شود. در واقع اگر  $n$  متغیر داشته باشیم و اندازه‌ی دامنه‌ی هر متغیر برابر  $d$  باشد، با این روش‌ها درختی به اندازه‌ی  $d^n \cdot n!$  تشکیل می‌شود؛ درحالی‌که تعداد کل حالات ممکن برابر  $d^n$  است. ویژگی خاص مسائل ارضای محدودیت، خاصیت جابه‌جایی<sup>۲</sup> است. این خاصیت به ما اجازه می‌دهد که در هر گره از درخت، تنها یک متغیر را بررسی و مقداردهی کنیم. با این روش، اندازه‌ی درخت به‌دست آمده برابر  $d^n$  می‌شود که برابر تعداد حالات ممکن است.[1]

در این گزارش، پیاده کردن مسئله‌ی سودوکو به صورت یک مسئله‌ی ارضای محدودیت و حل آن با استفاده از روش Backtracking به‌همراه MRV<sup>۳</sup> و Forward Checking را بررسی می‌کنیم.

## مسئله‌ی سودوکو

این مسئله، از یک جدول  $9 \times 9$  تشکیل شده است که خود شامل سه قطعه‌ی  $3 \times 3$  است. در هر خانه‌ی جدول عددی بین ۱ تا ۹ می‌تواند قرار بگیرد. شرط معتبر بودن چینش اعداد این است که اعداد داخل هر سطر، هر ستون و هر قطعه‌ی جدول تکراری نباشند.۲

## مدل مسئله

برای مدل کردن سودوکو به صورت CSP، لازم است متغیرها، دامنه‌ی آن‌ها و قیود مشخص شوند. در یک جدول سودوکو، معمولاً برخی خانه‌های جدول اولیه پر شده‌اند و مقدار بقیه‌ی خانه‌های جدول باید محاسبه شود. هر خانه‌ی جدول را با نماد  $X_{i,j}$  نشان می‌دهیم که  $1 \leq i, j \leq 9$ . در صورتی که خانه‌ای از ابتدا پر نشده باشد، در مجموعه‌ی متغیرها قرار می‌گیرد.

$$Variables = \{X_{i,j} | 1 \leq i, j \leq 9 \text{ \& } X_{i,j} \text{ is not filled}\}$$

دامنه‌ی هر متغیر  $X_{i,j}$  را با نماد  $D_{i,j}$  نشان می‌دهیم که مجموعه‌ی اعداد بین ۱ تا ۹ است.

$$D_{i,j} = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

قیود مسئله با توجه به شرط ذکر شده در شرح مسئله طراحی می‌شوند. برای هر متغیر، همسایه‌های آن

---

۱ Constraint Satisfaction Problem

۲ Commutativity

۳ Minimum Remaining Value

خانه‌هایی هستند که با آن‌ها در یک سطر، یک ستون یا یک قطعه قرار گرفته است. قیدها را با استفاده از همسایه‌های هر متغیر تعریف می‌کنیم. برای هر متغیر، باید مقدار آن با تمام همسایه‌هایش متفاوت باشد. اگر مجموعه‌ی همسایه‌ها را به صورت  $Neighbors(X_{i,j})$  نشان دهیم، قیدها به صورت زیر نوشته می‌شوند.

$$\forall Y_{\alpha,\beta} \in Neighbors(X_{i,j}) \quad X_{i,j} \neq Y_{\alpha,\beta} \text{ for all } X_{i,j} \text{ in Variables}$$

با داشتن این مجموعه‌ها و روابط، اکنون می‌توانیم مسئله را به صورت CSP حل کنیم.

## حل و تحلیل

برای حل، از الگوریتم Backtracking به همراه روش‌های MRV و Forward Checking استفاده می‌کنیم. الگوریتم Backtrack مانند یک جستجوی DFS می‌کند، با این تفاوت که هرگاه تشخیص دهد که در شاخه‌ی فعلی جوابی یافت نمی‌شود، به گره‌ی قبلی باز می‌گردد و شاخه‌های دیگر را ادامه می‌دهد. برای تشخیص عدم وجود جواب در یک شاخه، می‌توان از روش‌های ذکر شده استفاده کرد تا نیازی به پیمودن تمام شاخه نباشد.

## هیوریستیک‌ها

در ادامه روش‌هایی که در کنار الگوریتم Backtrack استفاده شده‌اند را بررسی می‌کنیم. [1]

۱. **اصلاح دامنه‌ی اولیه:** در آغاز حل جدول سودوکو، مقدار برخی خانه‌های جدول مشخص و ثابت شده است که باعث می‌شود در همان آغاز، دامنه‌ی متغیرهای همسایه‌ی آن‌ها شامل مقادیری باشد که در قیدها صدق نمی‌کنند. به همین دلیل، در آغاز تشکیل جدول، ابتدا تمام دامنه‌ها را اصلاح می‌کنیم تا با مقادیر ثابت اولیه سازگار باشند. برای این کار، خانه‌های ثابت را پیدا می‌کنیم و مقدار آن‌ها را از دامنه‌ی همسایه‌هایشان حذف می‌کنیم. با این کار، دامنه‌ها کوچکتر می‌شوند و درخت جستجو کوچک می‌شود.

۲. **انتخاب متغیر با MRV:** در هر مرحله از الگوریتم Backtracking، باید متغیری را برای مقداردهی انتخاب کنیم. در روش MRV، متغیری انتخاب می‌شود که تعداد مقادیر مجاز آن از همه کمتر باشد.

۳. **به‌روز رسانی دامنه‌ها با Forward Checking:** هر بار که به یک متغیر مقداری نسبت داده می‌شود، متغیرهای همسایه‌ی آن دیگر نمی‌توانند آن مقدار را بگیرند؛ بنابراین این مقدار را از دامنه‌ی تمام آن‌ها می‌توان حذف کرد. در صورتی که در این مرحله دامنه‌ی یکی از متغیرها تهی شود، مقدار تخصیص داده شده به متغیر برگردانده می‌شود و یک مقدار جدید به آن داده می‌شود یا به گره قبلی برمی‌گردیم. به این کار Forward Checking می‌گویند.

## اجرا و تحلیل

در این قسمت جزئیات حل چهار سری مسئله‌ی سودوکو با روش ذکر شده را بررسی می‌کنیم. این جدول‌ها از وبسایت websudoku برداشته شده‌اند.<sup>۴</sup>

جدول ۱: تحلیل اجرای ۴ سری جدول سودوکو. دو مسئله‌ی اول Hard و دو جدول آخر از سطح Evil.

تعداد خانه‌های ثابت	زمان اجرا به میلی ثانیه	تعداد گره کشف شده	تعداد مقادیر حذف شده از دامنه‌ها با FC	تعداد دفعات برگشت در Backtrack	تعداد تشخیص شکست توسط FC	تعداد مقادیر حذف شده در اصلاح اولیه دامنه
۲۶	۲.۴۴	۷۴	۲۰۹	۱۸	۱	۲۹۷
۲۷	۳.۷۴	۶۸	۱۸۷	۱۳	۲	۲۹۸
۲۴	۳۶.۵	۱۷۴۶	۷۱۲۳	۱۶۸۸	۲۲۹	۲۸۶
۲۶	۷.۷۴	۳۰۸	۱۰۶۲	۲۵۲	۲۷	۲۹۵

همانطور که مشاهده می‌شود، با کاهش تعداد مقادیر ثابت اولیه، تعداد متغیرها افزایش می‌یابد و اندازه‌ی درخت مسئله بزرگتر می‌شود. با بزرگتر شدن درخت، احتمال شکست خوردن هر شاخه‌ی Backtrack افزایش یافته و تعداد گره‌های بیشتری باید کشف شود.

```
[4, 2, 8, 6, 1, 7, 3, 5, 9]
[9, 6, 7, 2, 5, 3, 4, 8, 1]
[1, 3, 5, 4, 9, 8, 2, 7, 6]
[6, 1, 2, 5, 4, 9, 8, 3, 7]
[7, 4, 9, 3, 8, 1, 6, 2, 5]
[5, 8, 3, 7, 2, 6, 9, 1, 4]
[2, 9, 6, 1, 3, 5, 7, 4, 8]
[8, 5, 4, 9, 7, 2, 1, 6, 3]
[3, 7, 1, 8, 6, 4, 5, 9, 2]
```

عکس ۲: خروجی برنامه برای نمونه‌ی قبلی

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

عکس ۱: نمونه ورودی برنامه

## پیاده‌سازی در پایتون

پیاده‌سازی کامل این برنامه در آدرس گیت‌هاب زیر قرار دارد.

<https://github.com/nimahsn/AI-Course-Projects/tree/main/report05-CSP>

پروژه دارای دو فایل csp\_solver.py و sudoku.py است که به ترتیب پیاده‌سازی الگوریتم‌ها و مدل مسئله در

<sup>۴</sup> <https://www.websudoku.com>

آن‌ها قرار دارد.

## مدل مسئله

کلاس Sudoku در فایل sudoku.py برای نمایش مسئله‌ی سودوکو نوشته شده است. این کلاس از یک ماتریس که به عنوان ورودی دریافت می‌کند برای نمایش جدول سودوکو استفاده می‌کند. در این ماتریس، برای خانه‌هایی که مقدار نداشته باشند عدد 0 ذخیره می‌شود. در تابع سازنده‌ی کلاس (\_\_init\_\_)، چند فیلد دیگر برای ذخیره‌ی متغیرها، دامنه‌ها و همسایه‌ها نیز ساخته می‌شود. دیکشنری neighbors به ازای هر خانه‌ی جدول، همسایه‌های آن را نگهداری می‌کند. فیلد variables، مجموعه‌ی متغیرها را نشان می‌دهد و در صورتی که یک متغیر مقدار دهی شود، از این مجموعه حذف می‌شود. در نهایت فیلد domain هم دامنه‌ی هر متغیر را ذخیره می‌کند. این کلاس با دریافت آرگومان reduce\_domain می‌تواند هیوریستیک اصلاح اولیه دامنه را اجرا کند. متغیر omits صرفاً برای تهیه این Log و نگارش این گزارش اضافه شده است. تابع constraint\_check وظیفه دارد که با دریافت یک متغیر و مقدار آن، بررسی کند که آیا این مقدار در قیود مسئله صدق می‌کند یا خیر. برای این کار کافی است که برابر نبودن این مقدار با تمام همسایه‌های متغیر ورودی بررسی شود.

تابع get\_neighbors برای ساخت دیکشنری همسایه‌ها نوشته شده است و به ازای هر خانه‌ی جدول، همسایه‌های آن را برمی‌گرداند.

```
3 class Sudoku():
4     def __init__(self, init_board: List[List[int]], reduce_domain: bool = True) -> None:
5         self.board = init_board
6         self.neighbors = {}
7         self.omits = 0
8         for i in range(9):
9             for j in range(9):
10                self.neighbors[(i,j)] = list(self.get_neighbors((i,j)))
11        self.variables = set()
12        for i in range(9):
13            for j in range(9):
14                if self.board[i][j] == 0:
15                    self.variables.add((i,j))
16        self.domain: Dict[Tuple[int, int], Set[int]] = {}
17        for var in self.variables:
18            self.domain[var] = set(range(1, 10))
19        if reduce_domain:
20            for i in range(9):
21                for j in range(9):
22                    if self.board[i][j] != 0:
23                        for var in self.neighbors[(i,j)]:
24                            if var in self.variables and self.board[i][j] in self.domain[var]:
25                                self.domain[var].remove(self.board[i][j])
26                                self.omits += 1
27
28    > def constraint_check(self, index, value) -> bool:--
29
30
31
32
33
34    @staticmethod
35    > def get_neighbors(index: Tuple[int, int]):--
```

عکس ۳: کلاس Sudoku

## پیاده‌سازی الگوریتم

کلاس BacktrackBase در فایل csp\_solver.py یک کلاس abstract برای پیاده‌سازی الگوریتم است. این کلاس تابع backtrack را که قسمت اصلی الگوریتم است را پیاده‌سازی می‌کند.<sup>۵</sup> سایر توابعی که برای اجرای این الگوریتم نیاز است باید در کلاس‌های فرزند این کلاس پیاده شوند.<sup>۳</sup>

```
45 def backtrack(self, csp):
46     self.discovered_nodes += 1
47     if self.is_complete(csp): return csp
48     var = self.select_unassigned_variable(csp)
49     for value in self.order_domain_values(csp, var):
50         inferences = None
51         if self.is_consistent(csp, var, value):
52             self.add_assignment(csp, var, value)
53             inferences = self.inference(csp, var, value)
54             if inferences != False:
55                 self.apply_inferences(csp, inferences)
56                 result = self.backtrack(csp)
57                 if result:
58                     return result
59             self.remove_assignment(csp, var)
60             if inferences:
61                 self.revert_inference(csp, inferences)
62     self.failed_branches += 1
63     return False
64
```

عکس ۴: پیاده‌سازی الگوریتم Backtrack در کلاس BacktrackBase

برای اضافه کردن هیوریستیک‌ها و تمرکز بر مسئله‌ی سودوکو، کلاس SudokuBacktrackForwardMRV نوشته شده است که کلاس قبلی را به ارث می‌برد. این کلاس توابع لازم برای اجرای الگوریتم را با استفاده از هیوریستیک‌ها و با توجه به مدل سودوکو پیاده می‌کند.

```
65 class SudokuBacktrackForwardMRV(BacktrackBase):
66 > def __init__(self) -> None: ...
70
71 > def is_complete(self, csp: "Sudoku"): ...
75
76 > def select_unassigned_variable(self, csp: "Sudoku"): ...
78
79 > def order_domain_values(self, csp: "Sudoku", var): ...
81
82 > def is_consistent(self, csp: "Sudoku", var, value): ...
84
85 > def inference(self, csp: "Sudoku", var, value): ...
99
100 > def remove_assignment(self, csp: "Sudoku", var): ...
103
104 > def add_assignment(self, csp: "Sudoku", var, value): ...
107
108 > def apply_inferences(self, csp: "Sudoku", inferences: dict): ...
111
112 > def revert_inference(self, csp: "Sudoku", inferences: dict): ...
```

عکس ۵: خلاصه‌ی کلاس SudokuBacktrackForwardMRV

<sup>۵</sup> پیاده‌سازی الگوریتم Backtrack براساس شبیه کد داخل کتاب است ولی متغیر assignment حذف شده و در کلاس Sudoku مقاداردهی‌ها انجام می‌شود.

در این کلاس، تابع `is_complete` بررسی می کند که مسئله حل شده است یا خیر. تابع `select_unassigned_variable` با استفاده از روش MRV، متغیر با کوچکترین دامنه را بر می گرداند.

```
76 def select_unassigned_variable(self, csp: "Sudoku"):
77     return min(csp.variables, key=lambda var: len(csp.domain[var]))
```

عکس ۶: پیاده سازی روش MRV

تابع `order_domain_values` تنها وظیفه دارد دامنه ی هر متغیر را از کلاس سودوکو بگیرد و برگرداند. اگر برای انتخاب مقادیر از یک هیوریستیک مانند LCV استفاده می شد، پیاده سازی آن در این تابع انجام می شد. تابع `is_consistent` بعد از انتخاب یک مقدار برای یک متغیر صدا زده می شود و بررسی می کند که آیا این مقدار در قیود مسئله صدق می کند یا خیر. تابع `inference` برای پیاده سازی روش Forward Checking نوشته شده است. این تابع بعد از مقداردهی یک متغیر صدا زده می شود و مقادیری که باید از دامنه ی همسایه های آن متغیر حذف شود را بر می گرداند.

```
85 def inference(self, csp: "Sudoku", var, value):
86     forward_check_domain = {}
87     for neighbor in csp.neighbors[var]:
88         if neighbor not in csp.variables:
89             continue
90         if value not in csp.domain[neighbor]:
91             continue
92         elif len(csp.domain[neighbor]) == 1:
93             self.forward_check_cut += 1
94             return False
95         else:
96             forward_check_domain[neighbor] = value
97             self.forward_check_omits += len(forward_check_domain)
98     return forward_check_domain
```

عکس ۷: پیاده سازی Forward Checking

سایر توابع وظیفه دارند که مقداردهی متغیرها و همچنین مقادیری که توسط تابع `inference` برگردانده شده اند را اعمال کنند. همچنین دو تابع هم برای حذف کردن مقدار متغیرها و بازگرداندن مقادیر حذف شده از دامنه ها (`revert` کردن تاثیر Forward Checking) نوشته شده است.

## اجرا و دریافت خروجی

نوت بوک `runner.ipynb` دارای یک تابع `solve` می باشد که با دریافت ماتریس ورودی، آبجکت هایی از کلاس های `Sudoku` و `SudokuBacktrackForwardMRV` می سازد و با فراخوانی تابع `backtrack` مسئله را حل می کند و جدول نهایی و برخی جزئیات اجرا را چاپ می کند. داده ها و مسائل موجود در این گزارش در همین نوت بوک به دست آمده است.

## منابع

- Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010 ١
- <https://en.wikipedia.org/wiki/Sudoku> ٢
- <https://github.com/speix/sudoku-solver> ٣