دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)
دانشکده ریاضی و علوم‌کامپیوتر

گزارش چهارم

حل سودوکو با استفاده از مسئله‌ی رضایت محدود (CSP)

نگارش
امید امیر آتشانی

شماره دانشجویی:
۹۸۳۳۰۰۵

استاد
دکتر مهدی قطعی

دکتر بهنام یوسفی‌مهر

اردیبهشت ۱۴۰۲

چکیده

سودوکو، مخفف یک عبارت ژاپنی 数字は独身に限る به معنی «ارقام باید تنها باشند» است.

نوع متداول پازل سودوکو یک جدول ۹x۹ است که کل جدول هم به ۹ جدول کوچک‌تر ۳x۳ تقسیم شده‌است. در این جدول چند عدد به طور پیش فرض قرار داده شده که باید باقی اعداد را با رعایت سه قانون زیر یافت:

- قانون اول: در هر سطر جدول اعداد ۱ الی ۹ بدون تکرار قرار گیرد.
- قانون دوم: در هر ستون جدول اعداد ۱ الی ۹ بدون تکرار قرار گیرد.
- قانون سوم: در هر ناحیه ۳x۳ جدول اعداد ۱ الی ۹ بدون تکرار قرار گیرد.

الگوریتم‌های نسبتاً ساده‌ای برای حالت ۹x۹ سودوکو وجود دارند که در کمتر از یک ثانیه سخت‌ترین آن‌ها را هم حل می‌کنند اما با بزرگ شدن اندازه جدول سختی این مسئله به‌طور قابل توجهی افزایش می‌یابد.

یکی از این الگوریتم‌ها که در ادامه بررسی خواهد شد مسئله رضایت محدودیت (CSP) می‌باشد.

در این گزارش حل حالت ۹x۹ و چند ابعاد دیگر مانند ۴x۴، ۱۶x۱۶ و ۲۵x۲۵ نیز بررسی خواهند شد.

واژه‌های کلیدی:

سودکو، Constraint Satisfaction Problem (CSP)، Backtracking، Arc Consistency 3،
Minimum Remaining Value (MRV)، Least Constraint Value (LCV)

فهرست مطالب

فصل اول

مقدمه

مسائل ارضای محدودیت Constraint satisfaction problem یا به اختصار CSP تعداد زیادی از مسائل هوش مصنوعی را در بر می‌گیرند. یک مثال ساده سودوکو می‌باشد که می‌توان آن را به عنوان یک مسئله ارضای محدودیت در نظر گرفت. بسیاری از مسائل مطرح در زمینه هوش مصنوعی را می‌توان به صورت مسائل ارضای محدودیت توصیف کرد. این مسائل با استفاده از مجموعه‌ای از متغیرها و تعدادی محدودیت برای مقادیری که این متغیرها می‌توانند اختیار کنند، تعریف می‌شوند. حل این مسائل مجموعه‌ای از مقادیر منحصر به فرد برای متغیرهاست، به طوری که تمامی محدودیت‌های موردنظر مسئله ارضا شده باشد. الگوریتم Arc Consistencyیکی از معروفترین الگوریتم‌ها برای حل این مسائل می‌باشد.

برای حل مسایل CSP نیاز داریم که سه چیز را تعیین کنیم:

۱. متغییرها: موجودیت‌هایی هستند که می‌خواهیم مقادیری را به آن‌ها نسبت دهیم. در اینجا می‌شود سلول‌های سودوکو.

۲. دامنه‌ها: مجموعه ای از مقادیر ممکن که هر متغیر می تواند بگیرد، یعنی ارقام ۱ تا n برای سودوکو.

۳. محدودیت‌ها: قوانینی هستند که تخصیص مقادیر را به متغیرها محدود می‌کنند، مانند شرط منحصر به فرد بودن سودوکو.

سه قانون داریم:

قانون اول: در هر سطر جدول اعداد ۱ الی n بدون تکرار قرار گیرد.

قانون دوم: در هر ستون جدول اعداد ۱ الی n بدون تکرار قرار گیرد.

قانون سوم: در هر ناحیه $\sqrt{n}x\sqrt{n}$ جدول اعداد ۱ الی n بدون تکرار قرار گیرد.

روش پَس‌گرد (به انگلیسی: Backtracking) یکی از شیوه‌های کلی جستجوی فضای حالت برای حل مسائل ترکیبیاتی است. این شیوه، تمام ترکیب‌های ممکن را بررسی می‌کند تا یک جواب پیدا کند یا تمام جواب‌های ممکن را شمارش کند. تنها مزیت روش پسگرد در این است که می‌توان حالت‌هایی را بدون آنکه صریحاً بررسی شوند، با در نظر گرفتن ویژگی‌های مسئله، کنار گذاشت. واژه BackTrack به وسیله یک ریاضی‌دان آمریکایی به نام D.H. lehmer در سال ۱۹۵۰ ابداع شد.

در بسیاری از مسائل می‌توان بدون نیاز به بررسی تمام ترکیبات، از اینکه جواب مسئله توسط چه ترکیبی از مقادیر قابل تحقق نیست اطمینان بدست آورد و بدین ترتیب بخش‌های بزرگی از فضای جستجو را کنار گذاشت.

پسگرد همه حالت‌های ممکن را برای جواب بررسی می‌کند تا حالت درست را بیابد. این یک جستجوی عمق اول بین جواب‌های ممکن است. هنگام جستجو اگر راهی که طی می‌شود نتیجه نداد (به جواب نرسید) به نقطه قبلی بازمی‌گردد و راه بعدی را امتحان می‌کند. اگر همه راه‌ها را امتحان کرد و به جواب نرسید، جستجو نا موفق بوده‌است. این الگوریتم معمولاً در قالب توابع بازگشتی پیاده‌سازی می‌شود. به این صورت که در هر بار فراخوانی تابع، با اضافه شدن یک متغیر به‌طور متناوب همه مقادیر ممکن را به آن نسبت می‌دهد و آن مقداری که با فراخوانی‌های بازگشتی بعدی سازگار است را ذخیره می‌کند. روش پسگرد را می‌توان یک پیاده‌سازی بازگشتی از جستجوی عمق-اول دانست.

شبه کد:

```
procedure backtrack(P, c) is
    if reject(P, c) then return
    if accept(P, c) then output(P, c)
    s ← first(P, c)
    while s ≠ NULL do
        backtrack(P, s)
        s ← next(P, s)
```

برای حل سودوکو با این تکنیک، خانه‌های خالی را با حالات مختلفی پر می‌کنیم به صورتی که قوانین نقض نشود و تا جایی اینکار را ادامه می‌دهیم که جدول کامل پر شود. به‌طور مثال از اولین خانه خالی شروع می‌کنیم، اگر در سطر و ستون و مربع ۳×۳ مربوط به این خانه عدد ۱ ظاهر نشده بود، این خانه را با ۱ پر می‌کنیم و ادامه می‌دهیم؛ وگرنه اعداد ۲ تا ۹ را امتحان می‌کنیم. هر بار به خانه‌ای رسیدیم که با هیچ عددی پر نمی‌شد باید اولین خانه قبل از آن که می‌توانیم عددش را زیاد کنیم را زیاد کنیم.

شبه‌کد:

```
backtrack(cell):
    if cell is empty:
        for i from 1 to 9:
            if i not in row and column and box cell:
                fill cell with i
                boolean result = backtrack(next cell)
                if result is True:
                    return True
                unfill cell
        return False
    if i is last cell:
        return True
    backtrack(next cell)
```

مزیت‌های این روش عبارتند از:

- اگر جدول درست باشد حتماً جواب پیدا می‌شود.
- زمان حل به سختی جدول ورودی وابسته نیست.
- پیاده‌سازی آن بسیار ساده است.

اما در صورت بزرگ‌تر شدن سودوکو خیلی طول می‌کشد و مناسب ما نیست.

برای حل این مشکل در بخش بعدی به تعدادی الگوریتم و پیاده سازی آن‌ها می‌پردازیم.

فصل دوم

الگوریتم‌ها و پیاده‌سازی

تابع read_sudoku : یک نام فایل را به عنوان ورودی می گیرد و یک لیست دوبعدی نشان دهنده بورد سودوکو برمی‌گرداند. خط اول فایل شامل اندازه سودوکو و به دنبال آن تعداد سلول های پر و سپس در سطر های بعد مختصات و مقدار خانه های پر را دریافت می‌کند.

سطر و ستون های صفحه سودوکو با اعداد ۰تا n-۱ شماره گذاری شده اند. سلول‌های خالی با ۰ نشان داده می‌شوند. تابع فایل را می‌خواند و یک لیست دوبعدی با اندازه پازل ایجاد می‌کند، جایی که هر سلول به ۰ مقداردهی اولیه می‌شود. سپس مقادیر فایل را در سلول‌های مربوط به بورد پر می‌کند.

```
def read_sudoku(file):
    size = int(file.readline())
    n_full_cells = int(file.readline())
    grid = [[0] * size for _ in range(size)]
    for _ in range(n_full_cells):
        i, j, value = map(int, file.readline().split())
        grid[i][j] = value
    return grid
```

تابع is_valid : یک بورد سودوکو، اندازه اش، ردیف و ستون یک سلول و مقدار را به عنوان ورودی می‌گیرد و اگر بتوان مقدار را بدون نقض قوانین سودوکو در سلول قرار داد، True را برمی‌گرداند. تابع بررسی می کند که آیا مقدار قبلاً در همان ردیف، ستون یا زیرمربع سلول وجود دارد یا خیر. اگر مقدار هر یک از این قوانین را نقض کند، تابع False را برمی‌گرداند.

```
def is_valid(grid, size, row, col, num):
    for i in range(size):
        if grid[row][i] == num or grid[i][col] == num:
            return False
    box_size = int(size ** 0.5)
    box_row, box_col = row // box_size * box_size, col // box_size * box_size
    for i in range(box_size):
        for j in range(box_size):
            if grid[box_row + i][box_col + j] == num:
                return False
    return True
```

تابع find_unassigned : یک بورد سودوکو و اندازه آن را به عنوان ورودی می گیرد و سطر و ستون یک سلول اختصاص نیافته را برمی‌گرداند. یک سلول اختصاص نیافته سلولی با مقدار ۰ است. تابع از طریق هر سلول بورد حلقه می زند و شاخص های اولین سلول اختصاص نیافته را که پیدا می‌کند برمی‌گرداند.

```
def find_unassigned(grid, size):
    for row in range(size):
        for col in range(size):
            if grid[row][col] == 0:
                return row, col
    return None
```

تابع mrv : یک بورد سودوکو و اندازه اش را به عنوان ورودی می گیرد و سطر و ستون سلول را با حداقل مقادیر باقی مانده (MRV) برمی گرداند. هیوریستیک MRV سلولی را با کمترین تعداد مقادیر معتبری که می توان به آن اختصاص داد انتخاب می کند. این تابع از طریق هر سلول اختصاص نیافته بورد حلقه می زند و تعداد مقادیر معتبری را که می توان به آن اختصاص داد محاسبه می کند. ایندکس های سلول را با حداقل تعداد مقادیر معتبر برمی گرداند. Min_pos با بزرگ‌ترین مقدار ۶۴ بیتی مقداردهی شده است. در صورتی که از سیستم ۳۲ بیتی استفاده می کنید با import کردن کتابخانه sys و فراخوانی maxsize() می توانید آن را تغییر دهید.

```
def mrv(grid, size):
    min_remaining, min_pos = 9223372036854775807, None
    for row in range(size):
        for col in range(size):
            if grid[row][col] == 0:
                count = sum(is_valid(grid, size, row, col, num) for num in range(1,
size + 1))
                if count < min_remaining:
                    min_remaining, min_pos = count, (row, col)
    return min_pos
```

تابع lcv : یک بورد سودوکو، اندازه اش و ردیف و ستون یک سلول را به عنوان ورودی می‌گیرد و فهرستی از مقادیری را که می‌توان به سلول نسبت داد، مرتب‌سازی شده بر اساس هیوریستیک حداقل مقدار محدود (LCV) برمی‌گرداند. هیوریستیک LCV مقداری را انتخاب می کند که کمترین تعداد مقادیر را در سلول های مجاور رد می کند. تابع از طریق هر مقدار در دامنه سلول حلقه می زند و تعداد مقادیر معتبری را که می توان به هر سلول همسایه نسبت داد محاسبه می کند. سپس مقادیر سلول را با مجموع تعداد مقادیر معتبر سلول های مجاور مرتب می کند.

```
def lcv(grid, size, row, col):
    return sorted((num for num in range(1, size + 1) if is_valid(grid, size, row, col,
num)),key=lambda num: sum(is_valid(grid, size, row, col, num) for row in range(size)
for col in range(size)))
```

تابع revise : یک بورد سودوکو، اندازه اش و ردیف و ستون یک سلول را به عنوان ورودی می گیرد و مقادیری را که قوانین سودوکو را نقض می کند از دامنه سلول حذف می کند. تابع از طریق هر مقدار در دامنه سلول حلقه می زند و بررسی می کند که آیا قوانین را نقض می کند. اگر این کار را کرد، تابع مقدار را از دامنه حذف می کند و True را برمی گرداند. اگر هیچ مقداری حذف نشود، تابع False را بر می گرداند.

```
def revise(grid, size, row, col):
    revised = False
    for num in range(1, size + 1):
        if grid[row][col] == num:
            continue
        if not is_valid(grid, size, row, col, num):
            grid[row][col] = 0
            revised = True
            break
    return revised
```

تابع ac۳ : یک بورد سودوکو، اندازه اش و یک صف از سلول ها را برای پردازش به عنوان ورودی می گیرد و الگوریتم AC-۳ را پیاده سازی می کند. الگوریتم AC-۳ مقادیری را از دامنه سلول هایی که قوانین سودوکو را نقض می کنند حذف می کند. این تابع از طریق هر سلول در صف حلقه می زند و مقادیری را که قوانین را نقض می کند از دامنه خود حذف می کند. اگر مقداری حذف شود، تابع همسایه های سلول را به صف اضافه می کند تا بعدا پردازش شود. درواقع با کاهش مکرر دامنه هر متغیر بر اساس محدودیت‌ها تا زمانی که راه حلی پیدا شود یا مشخص شود که هیچ راه حلی وجود ندارد کار می کند. در نهایت اگر به انتهای حلقه برسیم و صف خالی مانده باشد، الگوریتم True را برمی‌گرداند که نشان می‌دهد راه‌حلی پیدا شده است.

```
def ac3(grid, size, queue):
    while queue:
        row, col = queue.pop(0)
        if revise(grid, size, row, col):
            if grid[row][col] == 0:
                return False
            for i in range(size):
                if i != col and grid[row][i] == 0:
                    queue.append((row, i))
                if i != row and grid[i][col] == 0:
                    queue.append((i, col))
            box_size = int(size ** 0.5)
            box_row, box_col = row // box_size * box_size, col // box_size * box_size
            for i in range(box_size):
                for j in range(box_size):
                    r, c = box_row + i, box_col + j
                    if r != row and c != col and grid[r][c] == 0:
                        queue.append((r, c))
    return True
```

تابع Solve_sudoku : یک بورد سودوکو و اندازه اش را به عنوان ورودی می گیرد و با استفاده از الگوریتم عقبگرد با AC-۳ و هیوریستیک MRV و LCV پازل را حل می کند. این تابع ابتدا الگوریتم AC-۳ را برای کاهش دامنه سلول ها در شبکه اعمال می کند. سپس سلولی را با حداقل مقادیر باقیمانده (MRV) پیدا می‌کند و سعی می‌کند با استفاده از حداقل مقدار محدود (LCV) مقداری را به آن اختصاص دهد. اگر بتوان مقداری را تخصیص داد، تابع به صورت بازگشتی خود را فراخوانی می کند تا سلول های باقی مانده را حل کند. اگر نمی توان مقداری را تخصیص داد، تابع به سلول قبلی برمی گردد و مقدار دیگری را امتحان می کند. اگر همه مقادیر امتحان شده باشند و هیچ یک از آنها کار نکند، عملکرد بیشتر به عقب برمی گردد تا زمانی که بتوان مقدار دیگری را تخصیص داد.

الگوریتم Backtracking با تلاش برای تخصیص یک مقدار به یک سلول اختصاص نیافته و حل بازگشتی سلول های باقی مانده کار می کند. اگر بدون نقض قوانین سودوکو، مقداری را نتوان به یک سلول اختصاص داد، الگوریتم به سلول قبلی برگشته و مقدار دیگری را امتحان می کند. اگر همه مقادیر امتحان شده باشند و هیچ یک از آنها کار نکند، الگوریتم بیشتر به عقب برمی‌گردد تا زمانی که بتوان مقدار دیگری را تخصیص داد.
الگوریتم زمانی خاتمه می یابد که به تمام سلول ها یک مقدار اختصاص داده شود.

```
def solve_sudoku(grid, size):
    queue = [(row, col) for row in range(size) for col in range(size) if
grid[row][col] == 0]
    ac3(grid, size, queue)
    unassigned = mrv(grid, size)
    if unassigned is None:
        return True
    row, col = unassigned
    for num in lcv(grid, size, row, col):
        if is_valid(grid, size, row, col, num):
            grid[row][col] = num
            if solve_sudoku(grid, size):
                return True
            grid[row][col] = 0
    return False
```

تابع print_sudoku : یک لیست دوبعدی نشان دهنده بورد سودوکو به عنوان ورودی می گیرد و آن را در خروجی چاپ می کند. تمامی اعداد تک رقمی و دو رقمی در ۴ خانه چاپ شده تا تک رقمی و دو رقمی بودن آن‌ها باعث جابجایی فاصله‌ها نشود و اعداد زیر یکدیگر بیافتند.

```
Def print_sudoku(grid):
    for row in grid:
        out = “{: >4}” * len(grid)
        print(out.format(*row))
```

در نهایت فایل ورودی را خوانده و بورد سودوکو را می سازیم و اگر قابل حل بود بورد حل شده را چاپ و در غیر این صورت Unsolvable CSP! چاپ می شود. در نهایت نیز زمانی که برای حل سودوکو گذشت چاپ می شود.

```
if __name__ == '__main__':
    with open('/Users/omid/Downloads/input16.txt', 'r') as file:
        grid = read_sudoku(file)
    size = len(grid)
    start_time = time.time()
    if solve_sudoku(grid, size):
        print_sudoku(grid)
    else:
        print('Unsolvable CSP!')
    end_time = time.time()
    print("Time spent to solve:", end_time - start_time)
```

فصل سوم

تست هوش‌مصنوعی بازی

ابتدا از کوچک ترین سایز شروع می کنیم، ۴x۴ و با ۵ خانه پر تست می کنیم.

ورودی:

```
4
5
0 0 2
1 1 1
1 3 2
2 2 3
3 3 4
```

خروجی:

```
  2   4   1   3
  3   1   4   2
  4   2   3   1
  1   3   2   4
Time spent to solve: 0.00072908401489257812 s
```

مشاهده می شود که هر ۳ شرط رعایت شده و در زمان بسیار کوتاهی سودوکو حل شده است.

حال یک سایز بزرگتر را تست می کنیم یعنی ۹x۹

به دلیل طولانی تر بودن ورودی آن را در گزارش قرار ندادم اما در گیتهاب موجود است. ورودی تست شده ۳۰ خانه پر دارد.

خروجی:

```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
Time spent to solve: 0.0187110900878906 25 s
```

مشاهده می شود که هر ۳ شرط رعایت شده و باز هم در زمان کوتاهی سودوکو حل شده است.

حال یک سایز بزرگتر را تست می کنیم یعنی ۱۶x۱۶

ورودی تست شده ۱۰۰ خانه پر دارد.

```
 4 10  9 15  1  7 13  8  6 14  2 12 16  5  3 11
 2  5  3  1 15  4 11 16 13  9  8  7  6 10 12 14
14  6 13 12  3 10  5  2 16 11  1  4  8 15  9  7
11  7 16  8  6 14  9 12  5  3 10 15  1  2 13  4
 8 16 11  4 13 15 14  9  2  5  7  3 12  1 10  6
 1 14  6 13 12  8  4  5 10 16  9 11  2  3  7 15
10 15  5  3  2  1  7  6 12  4 13  8 11 16 14  9
 7 12  2  9 10 16  3 11  1  6 15 14  4  8  5 13
 3  1  7  6  5  9 12  4  8 10 16 13 14 11 15  2
15 11 14 10  7  6  8 13  9  1  3  2  5 12  4 16
16  8 12  2 11  3 10 15 14  7  4  5  9 13  6  1
 9 13  4  5 14  2 16  1 15 12 11  6  3  7  8 10
 5  2  1  7  4 11  6 10  3 15 12  9 13 14 16  8
 6  4 10 14  8 12  1  7 11 13  5 16 15  9  2  3
12  3  8 16  9 13 15 14  7  2  6  1 10  4 11  5
13  9 15 11 16  5  2  3  4  8 14 10  7  6  1 12
Time spent to solve: 0.329782962799072 27 s
```

مشاهده می شود که هر ۳ شرط رعایت شده و باز هم در زمان کوتاهی سودوکو حل شده است.

در تست های اولیه که صرفا با backtracking بود چند دقیقه برای حل سودوکوی مشابه زمان می برد.

حال بزرگترین سایز را تست می کنیم یعنی ۲۵x۲۵

ورودی تست شده ۱ خانه پر دارد برای آن که طولانی ترین حالت ممکن را بررسی کنیم.

ورودی:

25

1

0 0 24

خروجی:

24 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 25

5 6 7 8 9 16 17 18 19 25 1 2 3 4 15 20 21 22 23 24 10 11 12 13 14

10 11 12 13 14 1 2 3 4 15 20 21 22 23 25 5 6 7 8 9 16 17 18 19 24

15 16 17 18 19 20 21 22 23 14 5 6 7 8 24 25 10 11 12 13 1 2 3 4 9

20 21 22 23 25 10 11 12 13 24 9 16 17 18 19 1 2 3 4 14 5 6 7 8 15

1 9 3 10 2 14 12 15 5 4 23 13 16 17 6 11 18 21 19 7 24 25 20 22 8

16 22 20 21 23 24 1 6 3 17 15 18 19 14 2 4 25 8 9 5 13 7 11 10 12

12 18 11 14 7 25 13 20 10 22 3 1 5 21 8 6 24 2 15 16 19 4 23 9 17

17 19 25 15 8 18 7 2 21 16 4 9 10 24 11 12 22 20 13 23 3 5 14 1 6

4 13 6 5 24 11 8 19 9 23 7 22 20 25 12 14 3 1 17 10 2 18 15 21 16

2 3 13 16 15 4 23 11 20 6 24 19 14 7 9 21 12 18 10 1 25 22 8 17 5

25 12 5 17 1 8 19 13 7 3 16 10 15 6 22 9 14 23 24 4 11 20 2 18 21

19 7 23 11 6 9 22 14 25 1 17 8 21 2 18 13 15 5 16 20 4 24 10 12 3

8 4 21 20 22 15 10 17 24 18 13 12 23 5 3 2 7 6 11 25 9 16 19 14 1

14 10 24 9 18 12 5 16 2 21 11 25 1 20 4 17 8 19 3 22 6 23 13 15 7

3 24 8 7 5 19 9 1 14 2 18 4 6 10 17 23 11 13 25 15 22 12 21 16 20

21 2 16 1 11 17 25 10 15 5 12 23 8 19 20 24 4 9 22 3 14 13 6 7 18

9 23 14 6 12 3 18 4 11 7 22 5 13 15 21 16 19 10 20 17 8 1 24 25 2

22 20 4 19 17 13 16 21 6 8 14 24 25 3 1 7 5 12 2 18 15 10 9 11 23

13 15 18 25 10 22 24 23 12 20 2 7 9 11 16 8 1 14 6 21 17 3 4 5 19

6 14 1 4 13 2 3 5 16 10 19 15 11 12 23 18 17 24 7 8 21 9 25 20 22
7 5 9 12 16 6 14 24 22 13 8 3 18 1 10 19 20 25 21 11 23 15 17 2 4
11 25 10 22 20 21 15 8 17 12 6 14 4 9 7 3 23 16 1 2 18 19 5 24 13
18 8 15 2 3 23 20 25 1 19 21 17 24 22 13 10 9 4 5 12 7 14 16 6 11
23 17 19 24 21 7 4 9 18 11 25 20 2 16 5 22 13 15 14 6 12 8 1 3 10
Time spent to solve: 16.21527075767517 s

مشاهده می شود که هر ۳ شرط رعایت شده و در بدترین حالت ممکن نیز زمان مناسبی (زیر ۲۰ ثانیه) سودوکو حل شده است.

فرمت خروجی در گزارش کمی بهم ریخت، با ران کردن کد می‌توان خروجی را تمیز تر دید.

فایل های ورودی دیگری برای اندازه ۱۶x۱۶ و ۲۵x۲۵ تمامی اعداد جدول را دارند و تعداد خانه های پر در سطر دوم ورودی کمتر از کل خانه ها می باشد تا سودوکو حل شود. این فایل ها در گیتهاب قرار دارند و می توانید تعدادی سطر آن ها را پاک کرده و دوباره ورودی دهید.

در نهایت یک مثال غیر قابل حل تست می شود.

ورودی:

9
20
0 0 5
0 1 3
0 4 7
1 0 6
1 3 1
1 4 9
1 5 5
2 1 9
2 2 8

2 7 6

3 0 8

3 4 6

3 8 3

4 0 4

4 3 8

4 5 3

4 8 4

5 0 7

5 4 2

5 8 6

خروجی:

Unsolvable CSP!

Time spent to solve: 7.890884876251221 s

فصل چهارم

نتیجه‌گیری

مشاهده شد که الگوریتم های اضافه شده می توانند کمک شایانی در زمان حل سودوکو به خصوص در سایز های بالاتر بکنند. این الگوریتم ها توانستند زمان حل بورد ۱۶X۱۶ را که چند دقیقه طول می کشید به زیر ۱ ثانیه بکشانند و حتی بورد بزرگی مانند ۲۵X۲۵ را فقط با یک خانه پر زیر ۲۰ ثانیه حل کنند.

این الگوریتم ها عبارت بودند از:

Arc Consistency ۳

Minimum Remaining Value (MRV)

Least Constraint Value (LCV)

منابع

https://en.wikipedia.org/wiki/Sudoku
https://en.wikipedia.org/wiki/Backtracking
https://en.wikipedia.org/wiki/Sudoku_solving_algorithms
https://en.wikipedia.org/wiki/AC-3_algorithm

لینک کد

https://github.com/omidatashani/sudoku-solver

همفکری با دوستان: امیرعلی معتقدی، محمد امیر خانی و پارسا پورسیستانی