نام و نام خانوادگی: **امیرحسین علی‌بخشی**
شماره‌ی دانشجویی: **۹۷۳۱۰۹۶**

گزارش پروژه دوم (SUDOKU+)

استاد: **جناب آقای روشنفکر**
درس: **مبانی هوش مصنوعی**

فهرست مطالب

فرموله سازی مسئله

متغیر ها

میتوان متغیر ها را ترکیب دو تایی رنگ و عدد در نظر گرفت

دامنه ها

هر یک از خانه های موجود در جدول دو نوع دامنه دارند.

دامنه رنگ ها

میتواند زیر مجموعه ای از رنگ هایی باشد که کاربر وارد کرده است.

دامنه عدد ها

میتواند یک زیر مجموعه از اعداد ۱ تا n باشد.

محدودیت ها

تمایز اعداد در سطر و ستون

نباید در یک سطر یا یک ستون از جدولی که در اختیار داریم، اعداد تکراری وجود داشته باشد. همین موضوع باعث میشود در کل سطر ها و ستون ها، تمامی اعداد از ۱ تا n وجود داشته باشند.

تمایز رنگ در خانه های مجاور

رنگ یک خانه باید با رنگ خانه های بالا، پایین، چپ و راست خود تفاوت داشته باشد.

عدد بزرگتر، رنگ با اولویت بیشتر

اگر عدد یک خانه از خانه مجاورش بیشتر باشد، باید رنگی که برای آن انتخاب میشود دارای اولویت بیشتری نسبت به رنگ خانه کنارش باشد.

توضیح در مورد کد

در این قسمت به توضیح روند اجرای کد میپردازیم و در میان توضیحات، توابع استفاده شده را بررسی میکنیم.

قسمت main

در قسمت main یک while بینهایت وجود دارد که در آن ورودی ها وارد میشوند و تا زمانی که کاربر ورودی ها را درست وارد نماید، برنامه میتواند اجرا شود.

ماتریس های داده و دامنه

برای هر مسئله، هم برای رنگ ها و هم برای اعداد، دو ماتریس دامنه و داده اختصاص میدهیم. در ماتریس داده ها که مقدار اولیه ۰ دارد، مقادیر نسبت داده شده را قرار میدهیم؛ برای اعداد همان عدد را میگذاریم و برای رنگ ها، شماره اولویت هر یک از رنگ ها را قرار میدهیم که از ۱ تا m میباشد. برای دامنه ها هم هر یک از درایه های ماتریس، یک لیست میباشد که شامل مقادیر مجاز برای هر یک از خانه ها را مشخص میکند. در ابتدا هر خانه که مقداری نداشت را با دامنه کامل قرار میدهیم که جلو تر دامنه ها را به کمک سازگاری کمان به روز رسانی کنیم.

سازگاری کمان برای دامنه ها

حال که دامنه ها تولید شدند، باید از سازگاری کمان استفاده کنیم تا دامنه ها مقدار های درستی داشته باشند.

سازگاری کمان برای عدد ها

برای سازگاری کمان برای اعداد از کد زیر استفاده میکنیم که برای هر عدد، دامنه های آن سطر و ستون را آپدیت میکند:

```
for i in range(n):
                for j in range(n):
                    if numbers_table[i][j] != 0:  # there is a number in the cell
                        for k in range(n):
                            tmp = numbers_table[i][j]
                            while True:
                                try:
                                    numbers_domain[i][k].remove(tmp)
                                except ValueError:
                                    break
                            while True:
                                try:
                                    numbers_domain[k][j].remove(tmp)
                                except ValueError:
                                    break
```

سازگاری کمان برای رنگ ها

همچنین برای سازگاری کمان برای رنگ ها نیز از کد زیر استفاده میکنیم:

```
for i in range(n):
                for j in range(n):
                    if colors_table[i][j] != 0:  # there is a color in the cell!
                        colors_domain = arc_consistency(i, j)
```

که تابع arc_consistency به شرح زیر است:

```
def arc_consistency(row, col):
    temp = colors_table[row][col]
    if col > 0:
        try:
            colors_domain[row][col - 1].remove(temp)
        except ValueError:
            pass
    if col < n - 1:
        try:
            colors_domain[row][col + 1].remove(temp)
        except ValueError:
            pass
    if row > 0:
        try:
            colors_domain[row - 1][col].remove(temp)
        except ValueError:
            pass
    if row < n - 1:
        try:
            colors_domain[row + 1][col].remove(temp)
        except ValueError:
            pass
    return colors_domain
```

یعنی چک میکند که اگر رنگ خانه فعلی در ۴ خانه مجاور وجود داشت، آن رنگ را از دامنه آن ها حذف میکند.

Backtracking

Backtracking برای عدد ها

شیوه عملکرد ما برای حل این مسئله به این صورت میباشد که ابتدا سعی میکنم اعداد را مقدار دهی کنیم و سپس به سراغ رنگ ها برویم. از این رو با فراخوانی solve_sudoku_by_numbers اجرای الگوریتم شروع میشود.

```
def solve_sudoku_by_numbers(...):
    row, col = mrv_heuristic_for_numbers(numbers_domain, numbers_table)
    # table is complete from numbers
    # now we need to solve the problem by colors
    if row == -1:
      if solve_sudoku_by_colors(numbers_table, colors_table, colors_domain) != 0:
            return 1
        return 0
    for n in numbers_domain[row][col]:
        numbers_table[row][col] = n
        res = forward_checking_for_numbers(row, col, n, numbers_domain, numbers_t
able)
        if res != 0:
```

```
            if solve_sudoku_by_numbers(numbers_table, res, colors_table, colors_d
omain) != 0:
                return 1
    numbers_table[row][col] = 0
    return 0
```

این تابع یک تابع بازگشتی است که اگر تمام اعداد مقدار دهی شده باشند به سراغ solve_sudoku_by_colors میرود وگرنه داخل بدنه تابع میشویم.

Backtracking برای رنگ ها

تابع solve_sudoku_by_colors به صورت زیر است:

```
def solve_sudoku_by_colors(table, colors_table, colors_domain):
    row, col = mrv_heuristic_for_colors(colors_domain, colors_table)
    # table is complete
    if row == -1:
        return 1
    for assinNum in colors_domain[row][col]:
        colors_table[row][col] = assinNum
        res = forward_checking_for_colors(row, col, colors_table, colors_domain)
        if res != 0:
            if solve_sudoku_by_colors(table, colors_table, res) != 0:
                return 1
    colors_table[row][col] = 0
    return 0
```

هیوریستیک ها

اولِ دو تابعی که تعریف کردیم، باید در اصل قادر باشیم یکی از خانه های جدول که نسبت به سایر خانه ها صلاحیت بیشتری دارند را انتخاب کنیم. از همین رو همانطور که در صورت پروژه نیز از ما خواسته شد از دو هیوریستیک MRV و درجه استفاده میکنیم.

هیورسیتیک برای اعداد

هیوریستیک MRV برای اعداد

هیوریستیک MRV برای اعداد به صورت زیر است:

```
def mrv_heuristic_for_numbers(numbers_domain_copy, table):
    mincond = n + 1
    candidates = []
    for i in range(n):
        for j in range(n):
            # the cell is empty
            if table[i][j] == 0:
                if len(numbers_domain_copy[i][j]) < mincond:
                    candidates.append([i, j])
```

```
                    mincond = len(numbers_domain_copy[i][j])
                elif len(numbers_domain_copy[i][j]) == mincond:
                    candidates.append([i, j])
    # no candidates found because all cells has a value
    if len(candidates) == 0:
        return [-1, -1]
    return degree_heuristic_for_numbers(table, candidates)
```

هیوریستیک درجه برای اعداد

هیوریستیک درجه برای اعداد به صورت زیر است:

```
def degree_heuristic_for_numbers(table, candidates):
    maximum_degree = -1
    maxcell = []
    for cell in candidates:
        counts = 0
        for i in range(n):
            if table[i][cell[1]] == 0 or table[cell[0]][i] == 0:
                counts += 1
        if counts > maximum_degree:
            maxcell = cell
            maximum_degree = counts
    return maxcell
```

هیوریستیک برای رنگ ها

هیوریستیک MRV برای رنگ ها

هیوریستیک MRV برای رنگ ها به صورت زیر است:

```
def mrv_heuristic_for_colors(numbers_domain_copy, table):
    mincond = m + 1
    candidates = []
    for i in range(n):
        for j in range(n):
            # the cell is empty
            if table[i][j] == 0:
                if len(numbers_domain_copy[i][j]) < mincond:
                    candidates.append([i, j])
                    mincond = len(numbers_domain_copy[i][j])
                elif len(numbers_domain_copy[i][j]) == mincond:
                    candidates.append([i, j])
    # no candidates found because all cells has a value
    if len(candidates) == 0:
        return [-1, -1]
    return degree_heuristic_for_colors(table, candidates)
```

هیوریستیک درجه برای رنگ ها

هیوریستیک درجه برای رنگ ها به صورت زیر است:

```
def degree_heuristic_for_colors(table, candidates):
    maximum_degree = -1
    maxcell = []
    for cell in candidates:
        counts = 0
        row = cell[0]
        col = cell[1]
        if col > 0:
            if table[row][col - 1] == 0:
                counts += 1
        if col < n - 1:
            if table[row][col + 1] == 0:
                counts += 1
        if row < n - 1:
            if numbers_table[row + 1][col] == 0:
                counts += 1
        if row > 0:
            if numbers_table[row - 1][col] == 0:
                counts += 1
        if counts > maximum_degree:
            maxcell = cell
            maximum_degree = counts
    return maxcell
```

حال همانطور که گفته شد، در صورت وجود خانه‌ی انتخابی، به سراغ بدنه توابع بازگشتی گفته شده میرویم. بعنی برای خانه گفته شده تابع forward checking تعریف شده را فراخوانی میکنیم.

Forward checking

Forward checking برای اعداد

تابع forward_checking_for_numbers به صورت زیر تعریف میشود:

```
def forward_checking_for_numbers(row, col, num, numbers_domain, numbers_table):
    numbers_domain_copy = deepcopy(numbers_domain)
    for i in range(n):
        # remove the number from the cells in the same row or column
        while True:
            try:
                numbers_domain_copy[i][col].remove(num)
            except ValueError:
                break
        while True:
            try:
                numbers_domain_copy[row][i].remove(num)
            except ValueError:
                break
        # no value and empty domain
```

```
        if numbers_table[i][col] == 0 and len(numbers_domain_copy[i][col]) == 0 o
r\
                numbers_table[row][i] == 0 and len(numbers_domain_copy[row][i]) =
= 0:
            return 0
    # when a cell has color and number
    if numbers_table[row][col] > 0 and colors_table[row][col] > 0:
        # check the left cell
        if col > 0:
            if check_number(row, col, row, col - 1) == 0:
                return 0
        # check the right cell
        if col < n - 1:
            if check_number(row, col, row, col + 1) == 0:
                return 0
        # check the top cell
        if row > 0:
            if check_number(row, col, row - 1, col) == 0:
                return 0
        # check the bottom cell
        if row < n - 1:
            if check_number(row, col, row + 1, col) == 0:
                return 0
    return numbers_domain_copy
```

که check_number را به صورت زیر تعریف میکنیم که در اصل در صورتی که خانه مجاور دارای رنگ بود، اولویت رنگی را بررسی میکند:

```
def check_number(row, col, row_, col_):
    # if the neighbor cell has number AND color...
    if numbers_table[row_][col_] > 0 and colors_table[row_][col_] > 0:
        # if the number of current cell was greater than other cell but
        # the color priority was not, return 0.
        if numbers_table[row][col] > numbers_table[row_][col_] and \
                colors_table[row][col] > colors_table[row_][col_]:
            return 0
        # if the number of current cell was smaller than other cell but
        # the color priority was not, return 0.
        elif numbers_table[row][col] < numbers_table[row_][col_] and \
                colors_table[row][col] < colors_table[row_][col_]:
            return 0
```

Forward checking برای رنگ ها

حال به سراغ forward_checking_for_colors میرویم:

```
def forward_checking_for_colors(row, col, colors_table, colors_domain):
    colors_domain_copy = deepcopy(colors_domain)
```

```
    temp = colors_table[row][col]

    # check the left cell
    if col > 0:
        if check_color(row, col, row, col - 1, temp, colors_domain_copy) == 0:
            return 0

    # check the right cell
    if col < n - 1:
        if check_color(row, col, row, col + 1, temp, colors_domain_copy) == 0:
            return 0

    # check the top cell
    if row > 0:
        if check_color(row, col, row - 1, col, temp, colors_domain_copy) == 0:
            return 0

    # check the bottom cell
    if row < n - 1:
        if check_color(row, col, row + 1, col, temp, colors_domain_copy) == 0:
            return 0

    return colors_domain_copy
```

و check_color را به این صورت تعریف میکنیم که دامنه های دو خانه مجاور را بررسی میکند:

```
def check_color(row, col, row_, col_, temp, colors_domain_copy):
    # if the number of the neighbor was smaller than current cell
    # then remove colors with higher priority from domain
    if numbers_table[row][col] > numbers_table[row_][col_]:
        for color in range(1, temp + 1):
            while True:
                try:
                    colors_domain_copy[row_][col_].remove(color)
                except ValueError:
                    break
    # if the number of the neighbor was greater than current cell
    # then remove colors with lower priority from domain
    else:
        for color in range(temp, m + 1):
            while True:
                try:
                    colors_domain_copy[row_][col_].remove(color)
                except ValueError:
                    break
    # if domains saw empty return 0
```

```
    if colors_table[row_][col_] == 0 and len(colors_domain_copy[row_][col_]) == 0:
        return 0
```

همانطور که در توابع solve_sudoku دیدیم، مقداری که برمیگردانند یا ۰ است یا ۱. ۱ به معنی اتمام کار است و ۰ به معنی شکست میباشد. حال زمانی که این مسئله به کمک موارد گفته شده حل شد(backtracking, forward-checking)، علاوه بر ماتریس هایی که به دست آوردیم، یک عدد در متغیر result ذخیره میشود که بیانگر جواب داشتن/نداشتن مسئله میباشد. اگر result برابر با ۱ بود، یعنی مسئله جواب دارد و جواب آن را نمایش میدهیم؛ اما در صورت ۰ بودن آن، پیامی به کاربر نشان داده میشود که بیانگر جواب نداشتن مسئله میباشد.

خروجی کد

در هر بار اجرا وضعیت فعلی کد برای کاربر در ترمینال نمایش داده میشود. علاوه بر این ها، زمان اجرای الگوریتم نیز در ترمینال قابل مشاهده است. سه نمونه خروجی در قسمت زیر آورده شده است:

```
>>> SUDOKU+
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   >>> ENTER THE INPUTS: (PRINT Q TO EXIT)
5 5
r g b y p
*# 2b *r *# *#
*# *# *# *# 4y
*# *# *# *# *p
*# *# *# *# *#
1# *# 5r *# 2p
   >>> STARTING THE ALGORITHM...
   >>> TIMER IS ACTIVATED
   >>> SET THE DOMAINS (NUMBERS): DONE
   >>> SET THE DOMAINS (COLORS): DONE
   >>> ARC CONSISTENCY (NUMBERS): DONE
   >>> ARC CONSISTENCY (COLORS): DONE
   >>> ALGORITHM HAS BEEN FINISHED
   >>> TIMER IS STOPPED
      >>> CALCULATION TIME: 0.008944034576416016s
   >>> SHOWING THE RESULT

      3r 2b 4r 1g 5r
      2b 1y 3g 5r 4y
      4g 5r 2b 3g 1p
      5r 4g 1y 2b 3r
      1y 3b 5r 4g 2p
```

```
>>> SUDOKU+
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   >>> ENTER THE INPUTS: (PRINT Q TO EXIT)
5 3
r g b y p
1# *r *#
*# 3g *#
*g 1# *#
   >>> STARTING THE ALGORITHM...
   >>> TIMER IS ACTIVATED
   >>> SET THE DOMAINS (NUMBERS): DONE
   >>> SET THE DOMAINS (COLORS): DONE
```

```
   >>> ARC CONSISTENCY (NUMBERS): DONE
   >>> ARC CONSISTENCY (COLORS): DONE
   >>> ALGORITHM HAS BEEN FINISHED
   >>> TIMER IS STOPPED
      >>> CALCULATION TIME: 0.0s
   >>> THE PROBLEM HAS NO ANSWERS
```

```
>>> SUDOKU+
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
   >>> ENTER THE INPUTS: (PRINT Q TO EXIT)
q
   >>> END OF THE PROGRAM
```