حسین لشگری – 9731055 گزارش پروژه سوم

نحوه مدل سازی :

- در هر دو روش انتشار محدودیت از کلاس **Puzzle**، که در بردارنده مشخصات پازل اعم از ابعاد پازل، متغیرها، متغیرهای مقدارگرفته و نگرفته است، استفاده می شود.
- در روش **Forward Checking** ، متغیرها هر **Cell** در نظر گرفته شده است بنابراین در این الگوریتم از کلاس **Cell** که در بردارنده محل قرارگیری در پازل، دامنه، مقدار و همچنین همسایه های کناری است، استفاده می کنیم.
- در روش **MAC** ، متغیرها هر سطر و ستون در نظرگرفته شده اند تا محدودیت ها **Unary** و **Binary** شوند تا بتوان به روش **MAC** پیاده سازی کرد.

بنابراین از کلاس **Variable** که در بردارنده ابعاد پازل، سطر یا ستون بودن، شماره سطر یا ستون، دامنه، مقدار و همچنین **Cell** های قرارگرفته در آن است، استفاده می کنیم.

توضیح توابع مشترک در هر دو روش انتشار محدودیت :

- **mrv(Puzzle)** : این تابع بین متغیرهای مقدارنگرفته در پازل، متغیری که دامنه ی آن محدودترین است را برمی گرداند.
- **print_puzzle(Puzzle, x)** : این تابع برای چاپ لحظه ای پازل در ترمینال استفاده می شود. به این صورت که پازل و متغیری که جدیدا مقدار گرفته است را می گیرد و محتوای پازل و حرکت جدید را نشان می دهد.

الگوریتم Forward Checking :

بعد از تشکیل **Puzzle** و متغیرها چون تعدادی از متغیرها از همان ابتدا مقدار داشته اند یک بار تابع **forward_checking** صدا زده می شود تا دامنه ها اصلاح شوند.

نحوه کار این تابع مطابق کد زیر است:

این تابع برای هر متغیر مقدارنگرفته و هر مقدار در دامنه ی آن سه قانونی که باید برقرار باشند را بررسی میکند.

قانون اول : برای اینکه بیشتر از دو صفر و 1 پشت سرهم در هیچ سطر و ستونی وجود نداشته باشد اینکار با بررسی **Cell** های مجاور صورت می گیرد که اگر مقدارگرفته باشند و مقدار آن با **value** در دامنه یکی باشد آن مقدار از دامنه متغیر حذف می شود.

```
def forward_checking(puzzle: Puzzle):
    for cell in puzzle.not_assigned:
        for value in cell.domain:
            # more than 2 0s or 1s adjacency check
            if cell.left and cell.left.left and cell.left.left.value == cell.left.value == value:
                cell.domain.remove(value)
                continue
            if cell.up and cell.up.up and cell.up.up.value == cell.up.value == value:
                cell.domain.remove(value)
                continue
            if cell.down and cell.down.down and cell.down.down.value == cell.down.value == value:
                cell.domain.remove(value)
                continue
            if cell.right and cell.right.right and cell.right.right.value == cell.right.value == value:
                cell.domain.remove(value)
                continue
            if cell.left and cell.right and cell.left.value == value == cell.right.value:
                cell.domain.remove(value)
                continue
            if cell.down and cell.up and cell.down.value == value == cell.up.value:
                cell.domain.remove(value)
                continue
```

قانون دوم : اینکه تعداد صفر و یک ها در هر سطر و ستون برابر باشند توسط قطعه کد زیر بررسی می شود:

به این صورت که تعداد صفر و یک ها در هر سطر و ستون شمرده می شود و اگر مقدار **value** باعث شود این مقدار از نصف طول پازل تجاوز کند آن مقدار از دامنه حذف می شود.

این قطعه کد مربوط به بررسی سطرها است و برای بررسی ستون ها ادامه دارد.

```
        # equal number of 0s and 1s check
        zero_number = 0
        one_number = 0
        if value == 0:
            zero_number += 1
        elif value == 1:
            one_number += 1
        for i in range(puzzle.dimension * cell.index[0], puzzle.dimension * (cell.index[0] + 1)):
            if puzzle.cells[i].value == 0:
                zero_number += 1
            elif puzzle.cells[i].value == 1:
                one_number += 1
        if zero_number > puzzle.dimension / 2 or one_number > puzzle.dimension / 2:
            cell.domain.remove(value)
            continue
```

قانون سوم : برای بررسی یکتا بودن هر سطر و هر ستون از قطعه کد زیر استفاده می شود:

برای اینکار ابتدا بررسی می شود که کدام سطرها و ستون ها کامل هستند سپس بررسی می شود که آیا به ازای مقدار **value** اگر سطر/ستونی کامل شد با سطر/ستون دیگر یکسان نشود و اگر شد آن مقدار از دامنه حذف می شود.

کد برای بررسی ستون ها ادامه دارد.

```
        # every row and column uniqueness check
        cell.value = value
        row_complete = []
        for i in range(puzzle.dimension):
            row_complete.append(True)
            for j in range(puzzle.dimension * i, puzzle.dimension * (i + 1)):
                if puzzle.cells[j].value == -1:
                    row_complete[i] = False
                    break
        if row_complete[cell.index[0]]:
            for i in range(len(row_complete)):
                if i != cell.index[0] and row_complete[i] and cell.domain.count(value):
                    for j in range(len(row_complete)):
                        if puzzle.cells[puzzle.dimension * cell.index[0]+j].value != \
                                puzzle.cells[puzzle.dimension * i+j].value:
                            break
                        if j == len(row_complete) - 1:
                            cell.domain.remove(value)
        if cell.domain.count(value) == 0:
            cell.value = -1
            continue
```

درنهایت الگوریتم **backtracking** مطابق آنچه در درس مطرح شد، پیاده سازی شد.

برای اینکار ابتدا تعداد متغیرهای مقدارگرفته بررسی می شود که اگر همه مقدار گرفتند پازل کامل شده برگردانده می شود.

سپس یک متغیر برای مقداردهی انتخاب می شود. و به ازای هر مقدار در دامنه توسط **forward checking** بررسی می شود و اگر دامنه ی متغیری تهی شد مقدار بعدی در دامنه تست می شود و در غیراینصورت این تابع به صورت بازگشتی برای مقداردهی به متغیر بعدی صدا زده می شود.

هنگام مقداردهی گام ها به ترتیب در ترمینال چاپ می شوند.

```
def backtracking(puzzle: Puzzle):
    if len(puzzle.assigned) == len(puzzle.cells):
        return puzzle
    x = mrv(puzzle)
    for v in x.domain:
        prev_puzzle = copy.deepcopy(puzzle)
        x = mrv(puzzle)
        puzzle.not_assigned.remove(x)
        puzzle.assigned.append(x)
        x.value = v
        print_puzzle(puzzle, x)
        forward_checking(puzzle)
        flag = False
        for cell in puzzle.not_assigned:
            if len(cell.domain) == 0:
                flag = True
                break
        if flag:
            puzzle = prev_puzzle
            continue
        result = backtracking(puzzle)
        if result:
            return result
        puzzle = prev_puzzle
    x.value = -1
    print_puzzle(puzzle, x)
    return False
```

الگوریتم MAC :

در این الگوریتم متغیرها سطرها و ستون ها در نظر گرفته شده اند بنابراین اینکه تعداد صفر و یک های پشت هم بیشتر از 2 نشود و تعداد صفرها و یک ها برابر باشد، محدودیت های **Unary** هستند و با اصلاح دامنه متغیرها این محدودیت ها برقرار می شوند.

بنابراین هنگام تشکیل پازل و متغیرها این 2 قانون با اصلاح دامنه ها برقرار می شوند و تنها 2 محدودیت باقی می ماند.

یک آنکه هر سطر و هر ستون یکتا باشد. دو اینکه خانه تقاطع هر سطر و هر ستون یکی باشد.

این 2 قانون توسط **AC-3** بررسی و دامنه ها اصلاح می شود.

بنابراین درابتدای کار یکبار **AC-3** روی همه ی یال ها صدا زده می شود تا نسبت به هم سازگار شوند سپس پس از هر مقداردهی به یک متغیر دوباره الگوریتم **AC-3** روی یال های مجاور آن متغیر صدا زده می شود تا سازگاری یال همواره برقرار باشد.

هنگام ساخت هر سطر و ستون دو قانون **Unary** توسط کد زیر روی دامنه اجرا می شود.

برای اینکار ابتدا ضرب دکارتی دامنه تمام **Cell** ها در سطر/ستون را به عنوان دامنه موقت در نظر می گیریم سپس روی هر کدام از مقادیر دامنه دو قانون را بررسی می کنیم و در صورت برقراری به دامنه اصلی اضافه می کنیم.

```
        domain = []
        domains_list = []
        for cell in cells:
            domains_list.append(cell.domain)
        temp_domain = list(itertools.product(*domains_list))
        for d in temp_domain:
            flag = True
            for cell in range(len(d) - 2):
                if d[cell] == d[cell + 1] == d[cell + 2]:
                    flag = False
                    break
            if flag and d.count(0) == d.count(1):
                domain.append(d)
        self.value = None
        self.cells = cells
        self.domain = domain
```

تابع **ac3** پازل و صف شامل یال ها را می گیرد. و تا هنگامی که این صف خالی شود همه یال ها را سازگار می کند. نحوه کار این تابع مطابق آن چه در درس مطرح شد است که ابتدا هر یال در صف بررسی می شود و دامنه ی متغیر اصلاح می شود سپس اگر دامنه متغیر تغییر کرد یال های مجاور آن متغیر به صف اضافه می شوند تا هنگامی که این صف خالی شود. هنگام این عمل اگر دامنه متغیری تهی شد **False** برگردانده می شود. 2 قانون **Binary** توسط تابع **revise** بررسی می شود به این صورت که اگر دو متغیر هر دو سطر یا ستون بودند بررسی می کند به ازای هر مقدار در دامنه عضوی در دامنه دیگری وجود داشته باشد که یکی نشوند و درغیراین صورت بررسی می کند که به ازای هر مقدار دامنه بتواند خانه تقاطعشان یکی شود.

```python
def ac3(puzzle: Puzzle, queue: list):
    while len(queue):
        arc = queue.pop(0)
        if revise(arc):
            if len(arc[0].domain) == 0:
                return False
            for v in puzzle.variables:
                if v != arc[0] and v != arc[1]:
                    queue.append([v, arc[0]])
    return True
```

```python
def revise(arc: list):
    revised = False
    new_domain = []
    if arc[0].direction == arc[1].direction:
        for d in arc[0].domain:
            if len(arc[1].domain) == 1 and arc[1].domain[0] == d:
                revised = True
            else:
                new_domain.append(d)
    else:
        for d in arc[0].domain:
            flag = True
            for d2 in arc[1].domain:
                if d2[arc[0].index] == d[arc[1].index]:
                    new_domain.append(d)
                    flag = False
                    break
            if flag:
                revised = True
    arc[0].domain = new_domain
    return revised
```

الگوریتم **backtracking** مشابه الگوریتم **forward checking** عمل می کند با این تفاوت که اینجا **ac3** به جای **forward checking** صدا زده می شود.

```
def backtracking(puzzle: Puzzle):
    if len(puzzle.assigned) == len(puzzle.variables):
        return puzzle
    x = mrv(puzzle)
    for v in x.domain:
        prev_puzzle = copy.deepcopy(puzzle)
        x = mrv(puzzle)
        puzzle.not_assigned.remove(x)
        puzzle.assigned.append(x)
        x.value = v
        x.domain = [v]
        print_puzzle(puzzle, x)
        queue = []
        for v2 in puzzle.not_assigned:
            queue.append([v2, x])
        ac3(puzzle, queue)
        flag = False
        for variable in puzzle.variables:
            if len(variable.domain) == 0:
                flag = True
                break
        if flag:
            puzzle = prev_puzzle
            continue
        result = backtracking(puzzle)
        if result:
            return result
        puzzle = prev_puzzle
    x.value = None
    print_puzzle(puzzle, x)
    return False
```

مقایسه دو روش انتشار محدودیت :

با اجرای الگوریتم **MAC** و **Forward Checking** روی 8 تست کیس مجموع زمان پاسخ دهی برای هر الگوریتم به شکل زیر است:

```
Forward Checking => exec time: 0 minutes and 3.71 seconds
MAC => exec time: 0 minutes and 0.46 seconds
```

با مقایسه دو الگوریتم می بینیم **MAC** به طور قابل توجهی سرعتش بیشتر است. زیرا:

- این الگوریتم ناسازگاری های بیشتری را در هر مرحله تشخیص می دهد و دامنه ها را کوچکتر می کند بنابراین زودتر به جواب می رسد.
- در مواردی مانند این مسئله چون محدویت ها نهایتا باینری هستند و جواب یکتاست با همان اولین اجرای **AC-3** در هنگام ساخت پازل تمام دامنه ی متغیرها فقط دارای یک عضو می شوند و پاسخ پیدا می شود(اگر پاسخی وجود داشته باشد) و تمام ناسازگاری ها رفع می شود و دیگر نیازی به اجرای دوباره **AC-3** نیست.
  این دلیل دیگری بر اجرای سریع تر **MAC** است.

خروجی :

پس از اجرای الگوریتم اگر پاسخی وجود داشته باشد آن به شکل ماتریس خواسته شده در فایل خروجی نوشته می شود و اگر وجود نداشته باشد پیغام خطا نوشته می شود.

همچنین مراحل گام به گام مقداردهی به متغیرها تا رسیدن به جواب نیز در ترمینال چاپ می شود.