گزارش پروژه سوم:
پس از انتخاب یک متغیر، الگوریتم باید نسبت به انتساب مقدار به آن تصمیم گیری نماید. هیوریستیک مقدارِ با حداقل محدودیت مقداری برای یک متغیر انتخاب می کند که در گراف محدودیت، باعث ایجاد کمترین محدودیت در متغیرهای مجاور شود.
در الگوریتم ها(MRV)، محدودیت ها فقط در زمان انتخاب متغیر بعدی در نظر گرفته شد. ولی می توان با در نظر گرفتن زودهنگام برخی از محدودیت ها در جستجو یا حتی پیش از آغاز جستجو، تا حد زیادی فضای جستجو را کاهش داد . برای این منظور، از دو تکنیک زیر استفاده می شود:
Forward Checking
Constraint Propagation

وقتی که متغیر X انتساب داده می شود، الگوریتم بررسی پیشرو، تمامی مقادیری که با مقدار اختصاص داده شده به X ناسازگارند را از دامنه Y حذف می کند و این کار ار از طریق بررسی دامنه تمام متغیرهای انتساب نیافته Y انجام می دهد که به واسطه یک محدودیت با X در ارتباط هستند.

پس در کل سرعت forward checking و MAC بیشتر از MRV است چون با forward checking جلوتر دامنه ها را کاهش میدهد و این باعث افزایش سرعت در این پیاده سازی نسبت به MRV میشود.

```
"""
function for getting the input
"""


def scan():
    f = open('puzzle4.txt', "r")
    coordination = f.readline()
    x, y = map(int, coordination.split())
    table = [[0 for i in range(y)] for j in range(x)]
    heuristic = [[0 for i in range(y)] for j in range(x)]
    for i in range(x):
        data = f.readline().split()
        for j in range(y):
            table[i][j] = data[j]
            heuristic[i][j] = [0, 1]
    return table, heuristic


'''
print the table each time
'''


def terminal(table):
    strings_row = []
    for i in range(len(table)):  # all rows are copied in strings_row
        table_copy = table[i]
        table_copy = ''.join(table_copy)
        strings_row.append(table_copy)
    for j in range(len(table)):
        print(strings_row[j])
    print('\n\n\n')
```

```
'''
a class for rules , heuristic and filling the table
'''


class Game:
    def __init__(self, table, heuristic):
        self.table = table
        self.heuristic = heuristic
        self.dimension = len(table)

    '''
    a function to check rules and change the domains of blocks
    if there two ones or zeros in row it change the domain for next block
    if a row or column is one to full it check the equality between ones and zeros and update the domain
    '''

    def h(self):
        if not self.error():
            exit(0)
        strings_row = []
        strings_column = []
        for i in range(self.dimension):
            for j in range(self.dimension):
                if self.table[i][j] == '1':
                    self.heuristic[i][j] = [1]
                    if j < self.dimension - 1 and self.table[i][j + 1] == '1':
                        if j > 0 and 1 in self.heuristic[i][j - 1]:
                            self.heuristic[i][j - 1].remove(1)
                        if j < self.dimension - 2 and 1 in self.heuristic[i][j + 2]:
                            self.heuristic[i][j + 2].remove(1)
                    if i < self.dimension - 1 and self.table[i + 1][j] == '1':
                        if i > 0 and 1 in self.heuristic[i - 1][j]:
                            self.heuristic[i - 1][j].remove(1)
                        if i < self.dimension - 2 and 1 in self.heuristic[i + 2][j]:
                            self.heuristic[i + 2][j].remove(1)
                    if i < self.dimension - 2 and self.table[i + 2][j] == '1' and 1 in self.heuristic[i + 1][j]:
                        self.heuristic[i + 1][j].remove(1)
                    if i > 1 and table[i - 2][j] == '1' and 1 in heuristic[i - 1][j]:
                        self.heuristic[i - 1][j].remove(1)
                    if j < self.dimension - 2 and self.table[i][j + 2] == '1' and 1 in self.heuristic[i][j + 1]:
                        self.heuristic[i][j + 1].remove(1)
                    if j > 1 and self.table[i][j - 2] == '1' and 1 in self.heuristic[i][j - 1]:
                        self.heuristic[i][j - 1].remove(1)
                elif self.table[i][j] == '0':
                    self.heuristic[i][j] = [0]
                    if j < self.dimension - 1 and self.table[i][j + 1] == '0':
                        if j > 0 and 0 in heuristic[i][j - 1]:
                            self.heuristic[i][j - 1].remove(0)
                        if j < self.dimension - 2 and 0 in self.heuristic[i][j + 2]:
                            self.heuristic[i][j + 2].remove(0)
                    if i < self.dimension - 1 and self.table[i + 1][j] == '0':
                        if i > 0 and 0 in self.heuristic[i - 1][j]:
                            self.heuristic[i - 1][j].remove(0)
                        if i < self.dimension - 2 and 0 in self.heuristic[i + 2][j]:
                            self.heuristic[i + 2][j].remove(0)
                    if i < self.dimension - 2 and self.table[i + 2][j] == '0' and 0 in self.heuristic[i + 1][j]:
                        self.heuristic[i + 1][j].remove(0)
                    if i > 1 and self.table[i - 2][j] == '0' and 0 in self.heuristic[i - 1][j]:
                        self.heuristic[i - 1][j].remove(0)
                    if j < self.dimension - 2 and self.table[i][j + 2] == '0' and 0 in self.heuristic[i][j + 1]:
                        self.heuristic[i][j + 1].remove(0)
                    if j > 1 and self.table[i][j - 2] == '0' and 0 in self.heuristic[i][j - 1]:
```

```
    '''
    for having same count for zeros and ones in each row and column
    '''
    for i in range(self.dimension):
        counter = 0
        count_one = 0
        count_zero = 0
        x = y = -1
        for j in range(self.dimension):
            if self.table[i][j] == '-':
                counter += 1
                x = i
                y = j
            if self.table[i][j] == '1':
                count_one += 1
            if self.table[i][j] == '0':
                count_zero += 1
        if counter == 1:
            if count_one - count_zero == 1 and 1 in self.heuristic[x][y]:
                self.heuristic[x][y].remove(1)
            if count_zero - count_one == 1 and 0 in self.heuristic[x][y]:
                self.heuristic[x][y].remove(0)
    for i in range(self.dimension):
        counter = 0
        count_one = 0
        count_zero = 0
        x = y = -1
        for j in range(self.dimension):
            if self.table[j][i] == '-':
                counter += 1
                x = i
                y = j
            if self.table[j][i] == '1':
                count_one += 1
            if self.table[j][i] == '0':
                count_zero += 1
```

```
    for i in range(self.dimension):
        counter = 0
        count_one = 0
        count_zero = 0
        x = y = -1
        for j in range(self.dimension):
            if self.table[j][i] == '-':
                counter += 1
                x = i
                y = j
            if self.table[j][i] == '1':
                count_one += 1
            if self.table[j][i] == '0':
                count_zero += 1
        if counter == 1:
            if count_one - count_zero == 1 and 1 in self.heuristic[y][x]:
                self.heuristic[y][x].remove(1)
            if count_zero - count_one == 1 and 0 in self.heuristic[y][x]:
                self.heuristic[y][x].remove(0)
```

```
    '''
    backtrack algorithm with MRV checking
    each time update the heuristic of each block
    '''

    def MRV_backTrack(self):
        for i in range(self.dimension):
            for j in range(self.dimension):
                if len(self.heuristic[i][j]) == 1 and self.table[i][j] == '-':
                    self.table[i][j] = str(self.heuristic[i][j][0])
                    self.h()
                    terminal(self.table)
                    if not game.rules():
                        print("❌ ERROR ❌")
                        exit(0)
                    self.MRV_backTrack()
        for i in range(self.dimension):
            for j in range(self.dimension):
                if self.table[i][j] == '-':
                    if not self.error():
                        terminal(self.table)
                        exit(0)
                    self.table[i][j] = str(self.heuristic[i][j][0])
                    terminal(self.table)
                    if not game.rules():
                        self.heuristic[i][j].remove(self.heuristic[i][j][0])
                        self.table[i][j] = '-'
                        self.h()
                        self.MRV_backTrack()
                    self.h()
                    self.MRV_backTrack()
        return self.table
```

```
'''
check all the rules
if any of them is broken returns false esle returns true
'''

def rules(self):
    strings_row = []
    strings_column = []
    for i in range(self.dimension):  # all rows are copied in strings_row
        table_copy = self.table[i]
        table_copy = ''.join(table_copy)
        strings_row.append(table_copy)

    for i in range(self.dimension):  # each row should have equal zeros and ones
        count_one = 0
        count_zero = 0
        if '-' not in strings_row[i]:
            for j in range(self.dimension):
                if self.table[i][j] == '1':
                    count_one += 1
                if self.table[i][j] == '0':
                    count_zero += 1
            if count_one != count_zero:
                return False

    while len(strings_row) > 0:  # check if there is a repeated string in rows
        element = strings_row.pop(0)
        # print(strings_row, "before")
        if '-' not in element and element in strings_row:
            strings_row = []
            return False

    for j in range(self.dimension):  # all columns are copied in strings_column
        table_copy = [row[j] for row in self.table]
        table_copy = ''.join(table_copy)
        strings_column.append(table_copy)
```

یک کلاس برای کل عملیات و هیوریستیک و قوانین و backtracking وجود دارد. تابع هیوریستیک هردفعه جدول جدید را میگیرد و باتوجه به قوانین دامنه را تغییر میدهد.
تابع rules تمامی قوانین اعم از
هر سطر و هر ستون باید تعداد برابری صفر و یک داشته باشد

اعداد قرار گرفته در هر سطر و هر ستون باید یک رشتهی یکتا تولید کند

در هر سطر و ستون نباید بیش از 2 عدد تکراری پشت سر هم قرار بگیرد

```
    for i in range(self.dimension):  # each column should have equal zeros and ones
        count_one = 0
        count_zero = 0
        if '-' not in strings_column[i]:
            for j in range(self.dimension):
                if self.table[j][i] == '1':
                    count_one += 1
                if self.table[j][i] == '0':
                    count_zero += 1
            if count_one != count_zero:
                return False

    while len(strings_column) > 0:  # check if there is a repeated string in columns
        element = strings_column.pop(0)
        # print(strings_column, "before")
        if '-' not in element and element in strings_column:
            strings_column = []
            return False
    '''
    for the not three same number in a row rule
    '''
    for i in range(self.dimension):
        for j in range(self.dimension):
            if i + 2 < self.dimension:
                if self.table[i][j] == self.table[i + 1][j] == self.table[i + 2][j] != '-':
                    return False
            if i - 2 >= 0:
                if self.table[i][j] == self.table[i - 1][j] == self.table[i - 2][j] != '-':
                    return False
            if j + 2 < self.dimension:
                if self.table[i][j] == self.table[i][j + 1] == self.table[i][j + 2] != '-':
                    return False
            if j - 2 >= 0:
                if self.table[i][j] == self.table[i][j - 1] == self.table[i][j - 2] != '-':
                    return False
    return True
```

```
'''
use the class and functions to get the answer
'''
if __name__ == '__main__':
    table, heuristic = scan()
    game = Game(table, heuristic)
    game.h()
    game.error()
    table = game.MRV_backTrack()
    terminal(table)
```

را بررسی میکند و در صورت رعایت شدن همگی true برمیگرداند.

تابع complete کامل شدن جدول را بررسی میکند.

تابع error در صورت نداشتن جواب ارور چاپ میکند.

تابع backtracking هم الگوریتم را اجرا میکند و جدول را خانه به خانه پر میکند و این را با کمک تابع terminal نمایش میدهد.