CS 404 ASSIGNMENT 2
FALL 2020
Buse Ak
25469

CSP Representation of Aquarium Puzzle

The puzzle cells are represented by their row and column number such as the cell in the first column and first row is represented by number 11 and the cell in the sixth row and sixth column is represent by number 66.

Variables: numbers end with digits 1 to 6, between 11 and 66

Domain: 0,1 (0:cell is empty, 1: cell is filled with water)

11	**12**	**13**	**14**	**15**	**16**
21	**22**	**23**	**24**	**25**	**26**
31	**32**	**33**	**34**	**35**	**36**
41	**42**	**43**	**44**	**45**	**46**
51	**52**	**53**	**54**	**55**	**56**
61	**62**	**63**	**64**	**65**	**66**

```
for i in range(1, 7):
    problem.addVariables(range(i * 10 + 1, i * 10 + 7), range(0, 2))
```

Constraints:

The total number of filled cells in a row and column is limited by a given number. ExactSumConstraint is used in order to satisfy the total row and column constraint.

```
iterator_col = 1
for col in cols:
    # print(index_col)
    problem.addConstraint(ExactSumConstraint(col), range(10 + iterator_col, 70 + iterator_col, 10))
    iterator_col += 1

iterator_row = 1
for row in rows:
    # print(index_row)
    problem.addConstraint(ExactSumConstraint(row), range(10 * iterator_row + 1, 10 * iterator_row + 7))
    iterator_row += 1
```

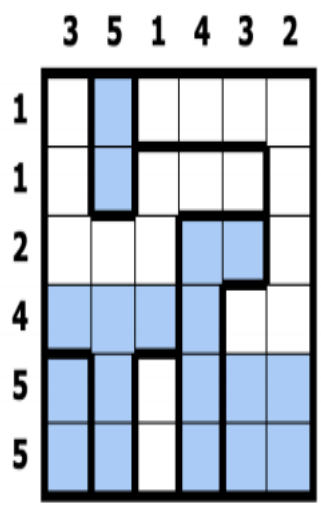
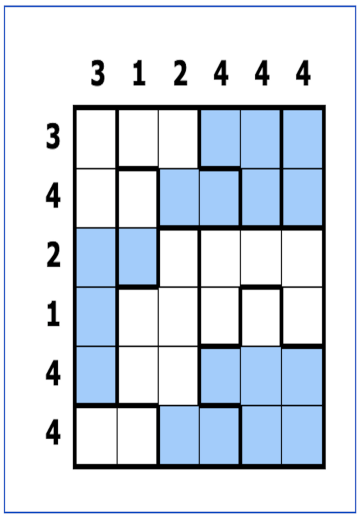
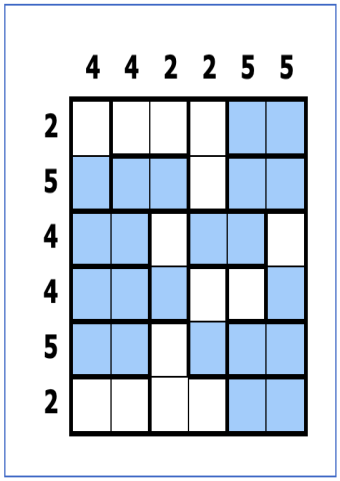
Puzzles are modeled considering their vertical and horizontal edges since the locations of the edges play important role in constraint creation. Edge contraints are added by proccessing the text files for the corresponding puzzle.

If there is no vertical edge between two cells ,which means vertical edge value is 0, then these two cells must have the same value 1 or 0. By processing vertical edges, AllEqualConstraint is added to two adjacent cells in case there is no vertical edge. The matrix containing the vertical edge information has 7x7 dimension since it includes the outer border of the puzzle. The left neighbor is not checked for the cells in the first column since left neighbor for them does not exist.

```
for i in range(len(vertical_edges)):
    for x in range(len(vertical_edges[i])):
        # always checking the left neighbor of the current cell
        # when x is 0, we cant check the left neighbor since it doesnt exist
        if x != 0:
            if vertical_edges[i][x] == 0:
                problem.addConstraint(AllEqualConstraint(), [10 * (i + 1) + (x + 1), 10 * (i + 1) + (x + 1) - 1])
```

Moreover, if there exists no horizontal edge between a cell and its lower neighbor, then if the lower neighbor is not filled with water, the upper cell can not be full due to gravity. FunctionConstraint , which specifies the value of upper cell by checking the value of lower cell: the lower cell's value has to be greater than or equal to the value of upper cell, is used. The lower neighbor for the cells in the last column is not checked since it does not exist.

```
for p in range(len(horizontal_edges)):
    for q in range(len(horizontal_edges)):
        # always checking the down neighbor of the current cell
        # when x is 6, we cant check the down neighbor since it doesnt exist
        if q != 6:
            if horizontal_edges[p][q] == 0:
                # a is the lower column of b
                # if a is not 1, then b can not be 1
                # if a is 0, then b is 0
                # if a is 1, be can be either 0 or 1
                problem.addConstraint(lambda a, b: a >= b, [10 * (q + 1) + (p + 1), 10 * q + (p + 1)])
```

Difficulty level: Easy	Normal	Hard
	 6x6 Normal Puzzle ID: 5,602,973	 6x6 Hard Puzzle ID: 6,886,797
easy.txt vertical edges 1,1,1,0,0,0,1 1,1,1,0,0,1,1 1,0,0,1,0,1,1 1,0,0,1,1,0,1 1,1,1,1,1,0,1 1,1,1,1,1,0,1 horizontal edges 1,0,0,0,1,0,1 1,0,1,0,0,0,1 1,1,0,0,1,0,1 1,1,1,0,0,0,1 1,1,1,1,0,0,1 1,0,0,0,0,0,1 columns 3,5,1,4,3,2 rows 1,1,2,4,5,5	normal.txt vertical edges 1,1,0,1,0,1,1 1,1,1,0,1,1,1 1,1,1,0,1,1,1 1,1,0,1,1,1,1 1,1,0,1,0,0,1 1,0,1,0,1,0,1 horizontal edges 1,0,0,0,0,1,1 1,1,0,1,0,1,1 1,0,1,0,0,0,1 1,1,1,0,1,1,1 1,0,1,1,0,0,1 1,0,1,0,1,0,1 columns 3,1,2,4,4,4 rows 3,4,2,1,4,4	hard.txt vertical edges 1,1,0,1,1,0,1 1,1,0,1,1,0,1 1,0,1,1,0,1,1 1,0,1,1,1,1,1 1,0,1,1,1,0,1 1,0,1,0,1,0,1 horizontal edges 1,0,1,1,1,1,1 1,1,1,1,1,1,1 1,1,1,0,1,0,1 1,0,1,1,0,1,1 1,1,1,1,1,1,1 1,1,1,0,1,1,1 columns 4,4,2,2,5,5 rows 2,5,4,4,5,2

Discussion on A* Search and CSP

For Aquarium puzzle constraint satisfaction problem, backtracking solver with forward checking is used. If the selected heuristic function used in A* search is admissible and monotone, than the solution is reached by following an optimal path. The heuristics might stand for number of conflicts to solve in order to reach the goal state. However, selecting a heuristic function and determining which state to follow is not easy and it may mislead the algorithm while it causes the creation of a large search space. For instance, the heuristic function value for the current assignment may be large but it may be because of wrong assignment of the previous step. Hence backtracking is needed. Backtracking search reduces the search space drastically by constraint propagation and the problem is simplified to solve since the domain for each state is rearranged and reduced by the algorithm.