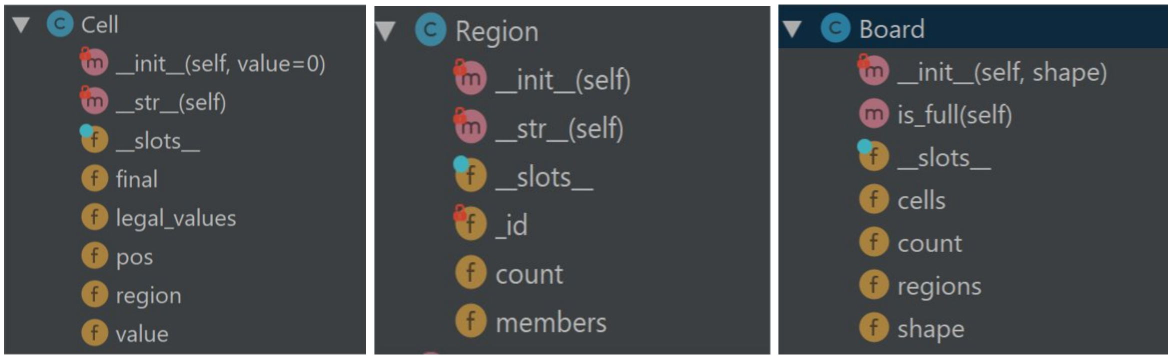
Intro to IS Lab 1 : Ripple Effect Report

In this lab, our task was to implement a brute force and an intelligent solver to solve the Ripple Effect puzzle. For intelligent solving, we had to use minimum remaining value heuristics, which essentially means to choose a successor state which has the least number of remaining legal values. This helps in reducing the branching factor, and thus, solving time.

We also had to incorporate forward checking utility. With forward checking, we can remove values from a cell's neighbourhood, once the value is assigned for a particular cell. This helps in essentially pruning illegal branches, which would have been explored in future, thus, reducing solving time.

Reading from input file

A puzzle is given in text format with "|" and "-" as region boundary. A cell can have a numerical value or ".", which means empty cell. To store the puzzle and search for a solution efficiently, I have used the following classes.



```
Cell
    __init__(self, value=0)
    __str__(self)
    __slots__
    final
    legal_values
    pos
    region
    value
```

```
Region
    __init__(self)
    __str__(self)
    __slots__
    _id
    count
    members
```

```
Board
    __init__(self, shape)
    is_full(self)
    __slots__
    cells
    count
    regions
    shape
```

Cell represents an individual cell of the puzzle. It can have a **value**, determined by the size of its **region**. It has a **position** in the Board, represented by (i, j), which is an array like representation. When the value of the cell is given as input, **final** is true. **Legal_values** is a list of all legal values possible for the cell.

Region represents a region on the board. It has an **_id,** which is assigned starting from 1. All cells in a region will have same region id. **Count** is a class variable and keeps track of total number of regions generated. **Members** is a dictionary of cells which are in the region, and provides fast access to all cell objects.

Board is the highest level abstraction of the puzzle. It has a **shape** (tuple), which is read from the input file. **Count** keeps track of the number of cells with value assigned. **Cells** and **regions** are dictionaries which store regions and cells of the board for efficient access. It would have been possible to exclude the cells dictionary from board because it's already there in the Region class, but in that case access to a particular cell would have been slower as we would have to check each region. The implemented solution uses a bit more space to provide quicker access. Finally, we have an **is_full** method in the Board class which returns true if all cells have assigned value, determined by the shape and count variables. This helps significantly while checking for goal state of the board, i.e., if is_full returns False, we can confidently say that goal state has not been reached without checking each cell's value for legality.

To make the board, first the text file is read as a 2-D list of characters (raw_board). Starting from the top left corner of the raw_board, it is checked if the cell is present in the Board object. If it is not present, then a new Region and a new Cell is created and then its surrounding is checked recursively with the created region as parameter. Thus, starting from the top left cell, all cells in that particular region are created first. After that the next cell, not present in the board is created, and so on. A newly created cell has either its input value or zero; zero is not a legal value in this puzzle.

Related functions are,

- **read_puzzle(file)**
- **make_board(raw_board, board_shape)**
- **_make_board(raw_board, board, shape, current, region=None)**

Brute force solver

The brute force solver has been implemented to start assigning all legal values (1 to region size) from top left cell. The successor states are always the next cell in the row (with all legal values). The algorithm is standard iterative Depth First Search, with the use of a stack to store all successors. If goal state is not reached but successor list is empty, it means a wrong value has been assigned, and so it backtracks to the previous state and tries the next value. Also, in every iteration, once the value has been assigned to that cell, it's checked if goal has been reached.

Related functions are,

- **brute_dfs_iter(board, calls=0, state=(0, 0))**
- **successors(board, state)**
- **is_num_placement_legal(board, num, pos)**
- **is_goal(board)**
- **clean_up(board, start, end)**

Intelligent Solving

In intelligent solving, MRV and forward checking are used to arrive at a solution faster. For both these techniques, Cell.legal_values is used to check the remaining legal values.

The algorithm runs as follows,

1. Cell.legal_values is updated with all legal values for each cell.
2. Recursive DFS-like function is called
    - 2.1. Goal state is checked. **If goal has been reached, return True**
    - 2.2. Call successor to get the next state
    - 2.3. If successor state has no legal values, **return False**
    - 2.4. For each legal value of the next state,
        - 2.4.1. Assign the value to the state
        - 2.4.2. Do **forward checking**. In forward checking, we go through the cell's region, row and column to remove values from neighbouring cells. Forward checking returns a record of the operations it performed, which is saved to a dictionary with the current state as key. This will help in backtracking to a previous state if we find that the goal is not reached and some cell has no legal value remaining.
        - 2.4.3. Do **constraint propagation**. In this, we go through every cell in the current cell's region and check if they have only 1 remaining value. If they do, the value is assigned and forward checking is done to remove values from neighbouring cells. Like forward checking, constraint propagation also maintains a record of all operations performed.
        - 2.4.4. Call the function at step 2 recursively. Expect a boolean return value
        - 2.4.5. If the value returned is false, wrong value has been assigned, thus, use the visited dictionary and backtrack. Also, remove entry from the visited dictionary
    - 2.5. Return result of 2.4.4 (True/False)

Step 2.2 in the algorithm uses minimum remaining value heuristic to return the next state by default, but a flag can be used to turn off this feature as suggested in the Lab requirements.

Related functions are,

- **dfs_mrv(board, calls=0)**
- **_dfs_mrv(board, calls, visited={}, found=False)**
- **initialize_legal_values(board)**
- **successors_mrv(board, visited, few=True)**

- **forward_checking(board, pos, num, visited)**
- **constraint_prop(board, pos, visited, ops)**
- **is_goal(board)**

Instructions to Run

The code is written in python uses standard libraries. It supports argument parser for easier control through console arguments. Usage description can be seen as follows.

```
$ python3 ripple.py -h
     usage: ripple.py [-h] [-b] [-nf] [-ncp] [-nv] file

     positional arguments:
       file                  path to puzzle file

     optional arguments:
       -h, --help            show this help message and exit
       -b, --brute           Run brute force solver
       -nf, --not_fewest     Don't select successor with min
     remaining values
       -ncp, --no_constraint_prop
                             Do not Use constraint propagation
       -nv, --no_visualization
                             Don't visualize final result
```

Only required argument is the path to the puzzle file. The program will run with MRV and constraint propagation by default, if no flag is used to turn these features off.

- -nf flag turns off the MRV feature; in this case, any one of the not visited states can be chosen as a successor state.
- -ncp flag turns off constraint propagation. Thus, after each forward checking, it won't be checked if there are cells with only one remaining legal value.
- -nv flag turns off turtle visualization.

Results

The following table has time in seconds and number of calls it took to solve each board with each method. Note that, forward checking cannot be turned off in intelligent solving, thus, every method except brute force does forward checking.

Board5 is a 9x9 puzzle that I converted manually from a puzzle and Board7 is same as Board6 with every given value removed. Board 6 and 7, which are both 18x18, takes 2 to 3 seconds to solve.

Machine config: Quad-Core Intel® Core™ i7-8550U CPU @ 1.80GHz, 16GB Ram.
OS: elementary OS 5.0 Juno, built on Ubuntu 18.04 LTS, Linux 4.15.0-45-generic.

Time(seconds) and calls to solver for different boards and methods

(time, calls)	**Board1 (4x4)**	**Board2 (10x10)**	**Board3 (10x10)**	**Board4 (7x6)**	**Board5 (9x9)**	**Board6 (18x18)**	**Board7 (18x18)**
Brute force solver (<file> -b)	(0.0004, 52)	(0.00092, 101)	(0.0111, 1032)	(0.00539, 655)	(2.0542, 226135)	NA	NA
No MRV, No CP (<file> -nf -ncp)	(0.0008, 67)	(0.00515, 200)	(0.808, 45503)	(0.00614, 309)	(1.83339, 122271)	NA	NA
No MRV (<file> -nf)	(0.00131, 34)	(0.00389, 117)	(0.6507, 27774)	(0.00435, 132)	(1.3973, 62556)	NA	NA
No CP (<file> -ncp)	(0.0004, 19)	(0.00356, 100)	(0.00384, 114)	(0.0015, 53)	(0.00624, 183)	(3.157, 50476)	(14.692, 313419)
With MRV and CP. Most efficient. (<file>)	(0.0003, 9)	(0.00251, 60)	(0.0039, 82)	(0.0011, 26)	(0.00456, 89)	**(1.909, 26647)**	**(2.948, 40091)**

Conclusions

From the results, it can be seen how efficiently minimum remaining value, forward checking and constraint propagation solve the puzzles. These utilities, solve the puzzle intelligently, by reducing legal values for neighbouring cells whenever possible. This method is similar to how a human might try to solve the puzzle. Brute force, on the other hand, just keeps trying values until the solution is found. It can be observed how significantly number of calls to the solver is reduced from row 1 to row 5 of the result table. Also, it is interesting to note that brute force sometimes takes less time to solve smaller puzzles, however, for larger puzzles, my brute force algorithm didn't finish in 15 minutes. Intelligent solver reduces the state space by pruning branches with the help of forward checking and constraint propagation.

While a larger puzzle takes longer time to solve, it is not the only factor that contributes to difficulty of the puzzle. Size of a region is a very important factor as can be seen from board 2 and 3. Both boards are 10x10 but board 2 has regions of size at most 3, while board 3 has larger regions. Larger region implies that the algorithm has to check more values for every cell, thus increasing difficulty. Also, complexity increases when number of regions are sufficiently large.

We can build easier or more difficult puzzle by changing the puzzle size, region size and region number. Increasing these parameters will make the puzzle more difficult to solve, and vice-versa.