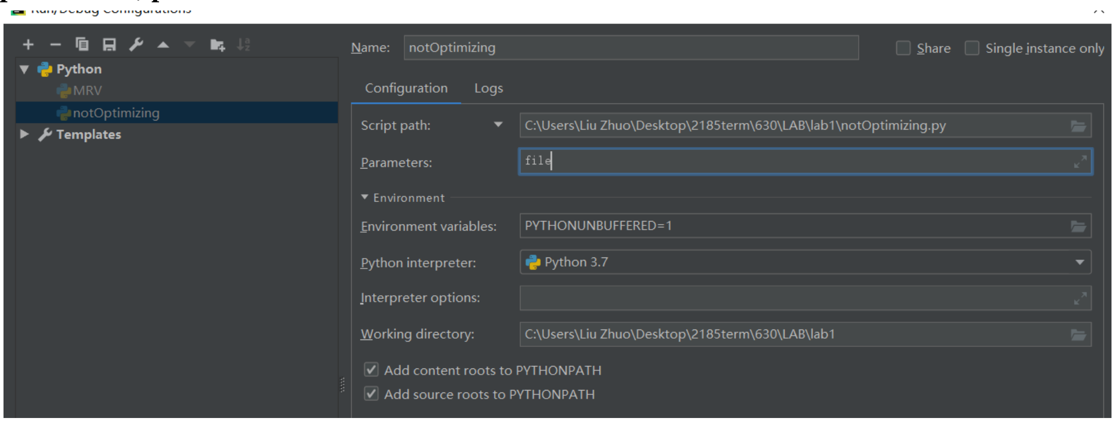
There are two python files, one is called BruteForce.python and the other one is called MRV.python.
These two python files are designed to compare the different forms of solving for efficiency.

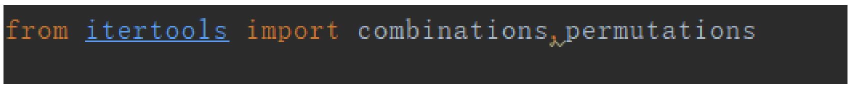
You have to name your file like xxx.txt first.
I use the name of txt file in the command line without **.txt**
Please look at the parameters shown below, if we use document file.txt, just use file as the parameters, please do not use file.txt instead.



Name: notOptimizing
Share
Single instance only
Script path: C:\Users\Liu Zhuo\Desktop\2185term\630\LAB\lab1\notOptimizing.py
Parameters: file
Environment variables: PYTHONUNBUFFERED=1
Python interpreter: Python 3.7
Interpreter options:
Working directory: C:\Users\Liu Zhuo\Desktop\2185term\630\LAB\lab1
✓ Add content roots to PYTHONPATH
✓ Add source roots to PYTHONPATH

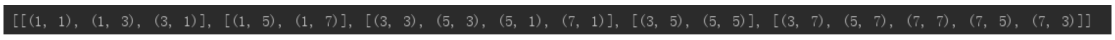
Please do not include any other symbols in the txt file, such as space or enter symbol, just copy the puzzle to txt file, save it and exit txt file.

At the beginning of this program, I need to import permutations from itools for my programming purpose. If you don't have this package, please make sure to download one.



```
from itertools import combinations,permutations
```

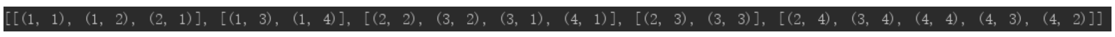
Using 4*4 puzzle for instance
The first step is to process the txt file given by professor, I use processFile() function to do this.
And then in my produceGraph() and processRegion() functions, I put the coordinates of each region in an array, the size of matrix is 7*7 we can see



```
[[(1, 1), (1, 3), (3, 1)], [(1, 5), (1, 7)], [(3, 3), (5, 3), (5, 1), (7, 1)], [(3, 5), (5, 5)], [(3, 7), (5, 7), (7, 7), (7, 5), (7, 3)]]
```

But we need to transformation operation which is shown as followed.

The purpose of processStack() is to convert the matrix to a smaller one, the size of matrix is 4*4 we can see, so we can get rid of the redundant space between coordinates



```
[[(1, 1), (1, 2), (2, 1)], [(1, 3), (1, 4)], [(2, 2), (3, 2), (3, 1), (4, 1)], [(2, 3), (3, 3)], [(2, 4), (3, 4), (4, 4), (4, 3), (4, 2)]]
```

searchDFS() function is to do the DFSsearch, which is brute-force solver solution. At first select region randomly, then do DFSsearch, if next region's numbers which are filled through DFS don't meet the constraint, then we go back to the previous region and allocate the next sequence of numbers. It's a recursive process.

postProcess() function is to output the correct result as a two-dimensional matrix

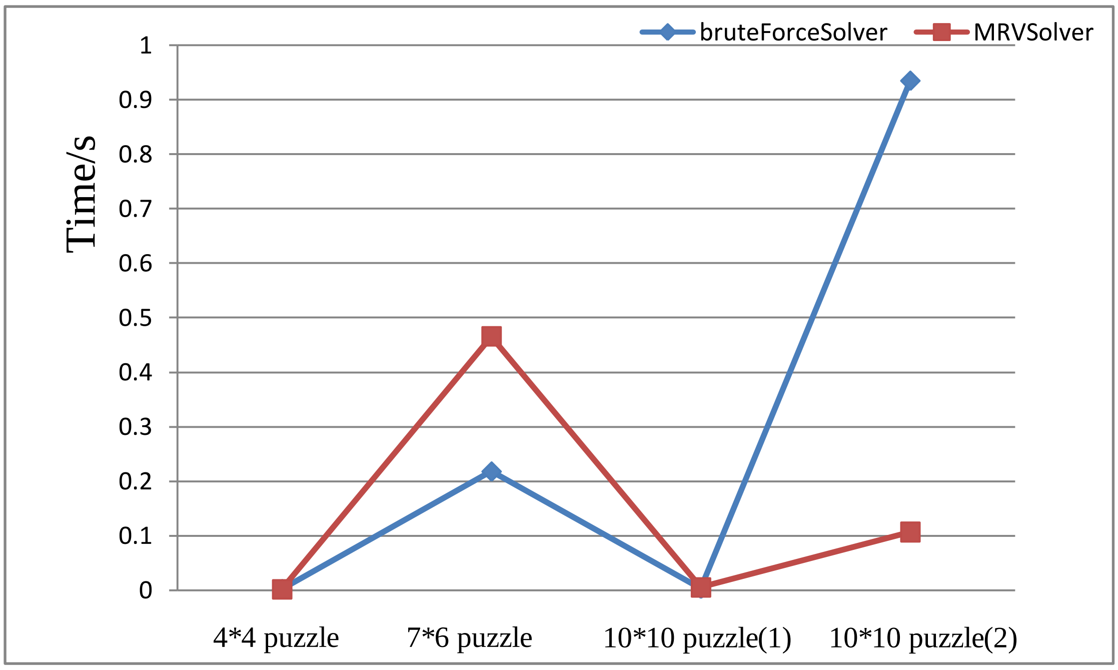


Fig 1. Empirical data for solving efficiency of two solvers(time parameter)

As the figure shown above, we can see with the increasing size of the puzzle, time of MRVSolver is much faster than the bruteForceSolver. As for the 7*6 puzzle, in my opinion, since the puzzle is really small and easy to deal with, our MRVSolver needs to select a cell with fewest possible values and eliminate all possible values in other cells that would be in conflict, so MRVSolver needs to take more time than bruteForceSolver. But as the complexity of puzzle increases, we can see the advantage of MRVSolver.

Please notice, in my report, I use two versions of 10*10 puzzle.

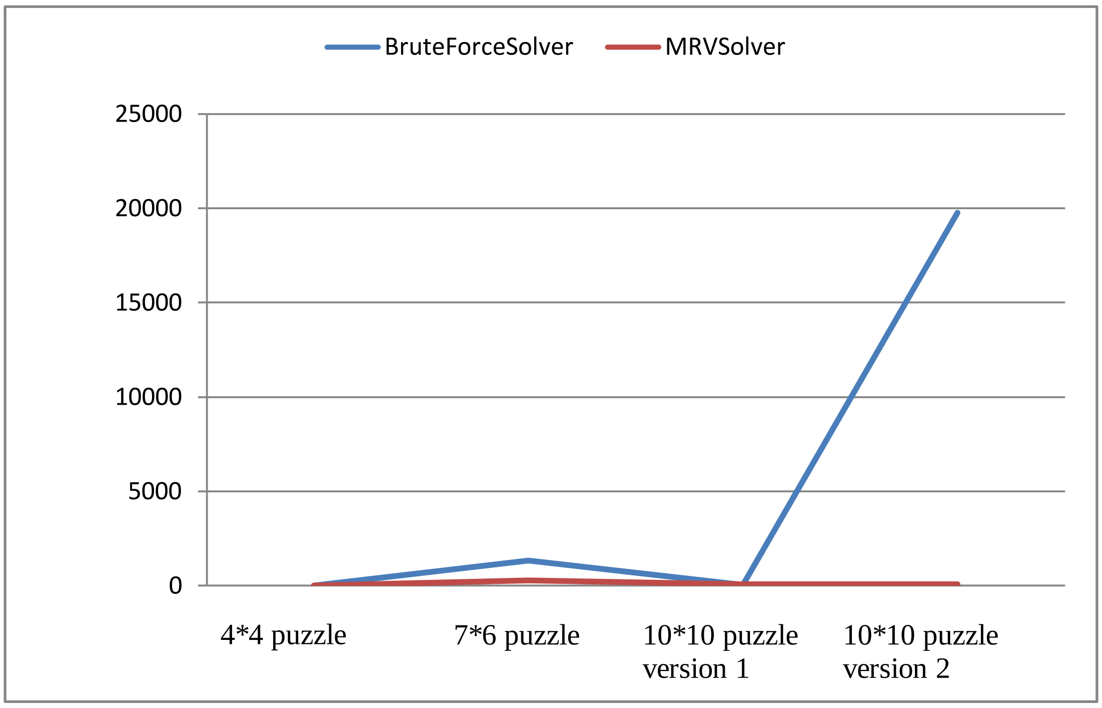


Fig 2. Empirical data for solving efficiency of two solvers(number of function calls)

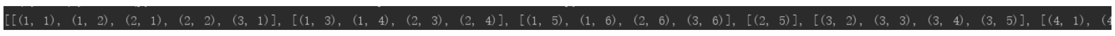
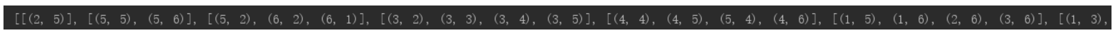
We can learn from the statistic that, BruteForceSolver's function calls is apparently more than MRVSolver's. It is obvious clear when the complexity of puzzle increases. This is because I use MRV forwarding checking, so the number of function calls reduced significantly.

Answer:

The complexity of puzzle is based on the size of puzzle and the amount of large regions which the size are over 5, the more complex the solution will be. If I design the puzzle, I will increase its size and contain much more large regions which size are over 5, so solving the puzzle will take much time.

Bonus solution:
At any point, there might be only one cell in a region where a particular number can be placed
In MRV.python, I use quickSort to sort each region, as we can see, after quickSort, I always fill one cell region first, and then use MRV and forward checking to immediately eliminate all possible values in other cells that would be in conflict. So in my program, I always detect one cell region first.

Improvement is based on the size of the puzzle and the MRV/forward checking function in my program. And because of I do quickSort first, the single cell can be visited and placed at the beginning of the solution of each puzzle.

Take 7*6 puzzle for instance:
Before quickSort:

```
[[(1, 1), (1, 2), (2, 1), (2, 2), (3, 1)], [(1, 3), (1, 4), (2, 3), (2, 4)], [(1, 5), (1, 6), (2, 6), (3, 6)], [(2, 5)], [(3, 2), (3, 3), (3, 4), (3, 5)], [(4, 1), (4
```
After quickSort:

```
 [[(2, 5)], [(5, 5), (5, 6)], [(5, 2), (6, 2), (6, 1)], [(3, 2), (3, 3), (3, 4), (3, 5)], [(4, 4), (4, 5), (5, 4), (4, 6)], [(1, 5), (1, 6), (2, 6), (3, 6)], [(1, 3),
```
That’s why I can make sure the single cell can be visited in the first place..