HW4 – Wenyue Wang, Shushu Zhao

- Our program includes three files, main.c to include the general processor of filling the puzzle, search.c to include all helper functions and search.h to include all header files.
- The general flow to is create a n*n puzzle matrix and a n*n*2 domain matrix. Then modify the domain matrix based on the pre-filled grids. After that, we will enter the main while loop, which will iterate the whole puzzle matrix. For each grid, we will test if the filled matrix agree with three constraints. If it passed all three constraints test, we will go to next grid; if now, we should backtrack to the one with available domains. This procedure would continue until we found the final puzzle solution.

Approach:

- Step1: read in the file (include puzzle size and initial filled matrix)
- Step2: get all O's and X's coordinates
- Step3: create a n*n puzzle matrix A and initialize with all zeros
- Step4: create a n*n*2 domain matrix D, initialize each grid's domain as 1 and 100, so that D[i][j][0] = 1, D[i][j][1] = 100, where we assume X as 1 and O as 100.
- Step5: based on O's and X's location, modify the domain matrix. If one grid is pre-defined to be X, then its domain is [1][1]; similarly, if one grid is pre-defined to be O, then its domain will be modified as [100][100]
- Step6: make a copy of modified domain
- Step7: enter the main while loop, Initialized r = 0, c = 0, d_index = 0;
    1) Get the domain number based on d_index, pop it out from domain matrix and insert it in puzzle matrix.
    2) Get the current grid's current row and current column
    3) Perform **three constraint tests (next page)** based on whole matrix, current row and column.
    4) If all tests passed: move to next grid
    5) If one of the tests not passed: check current grid's domain, if no domain left, then backtrack to previous grid and recover current grid's domain. Until we found a proper grid.
    6) Repeat these process until we iterating over the whole matrix
- Step8: print out results, perform final test and free up memory.

Constraint tests:

<u>Constraint 1 - each row and each col must have same number of O's and X's:</u>

Here we let X be 1 and O be 100, then if puzzle size is 6, then each row or column should have 3 O's and 3 X's. Thus, we summarize the current row or current column as SUM. Then number of O is sum/100, number of X is sum%100, we would test if it equals to 3 to see if it passes this test.

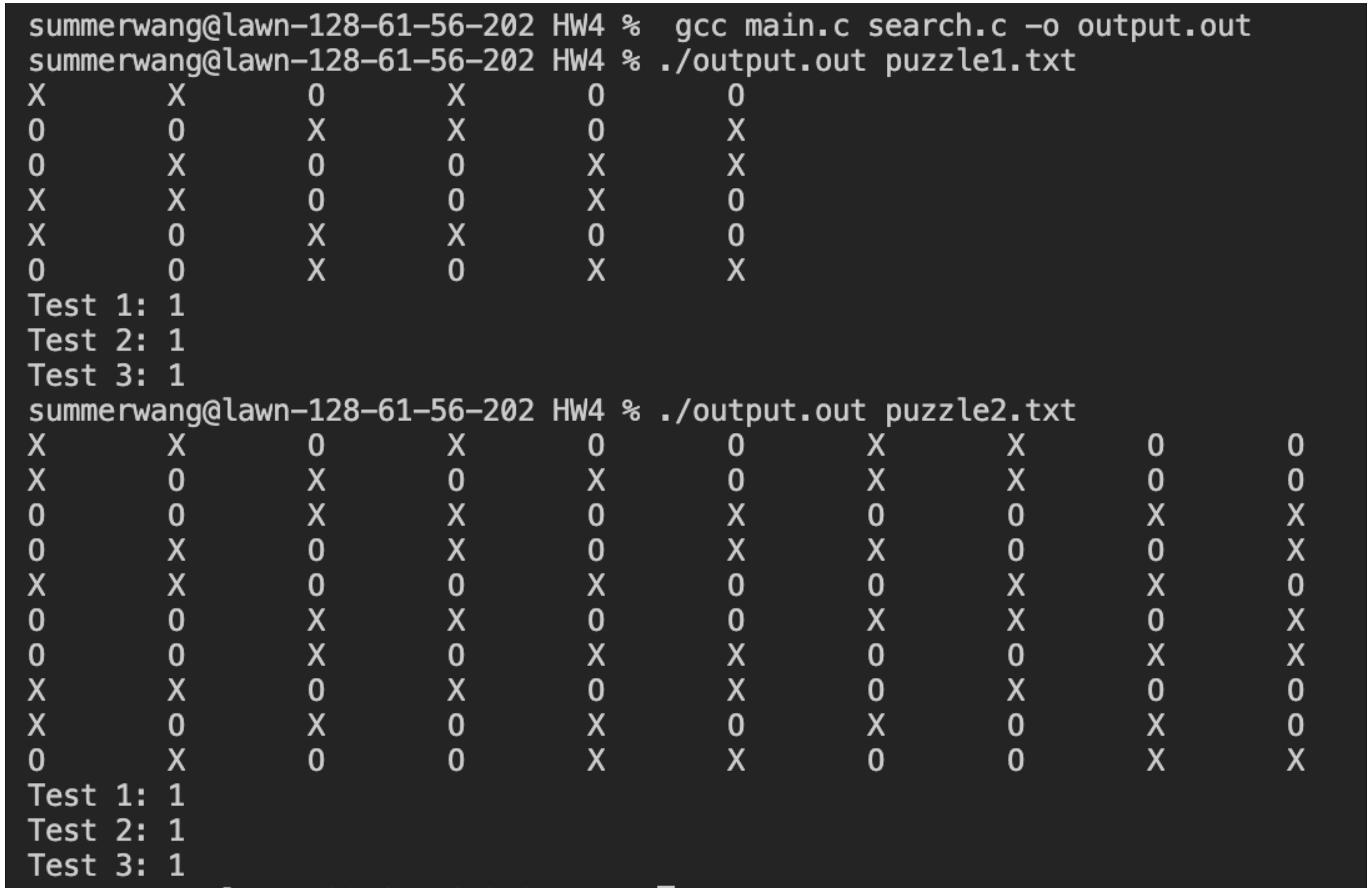
<u>Constraint 2 – no more than 2 X's or O's can appear in successions in any row or column:</u>

For each row/col, if number of filled elements is less than 3, we would pass the test. If not, we would check if the current number equal to the previous one and if it equals to the one before the previous one.

<u>Constraint 3 – each row/col is different from each other row/col:</u>

Build a vector to store current row/col and a matrix to store all previous rows/cols. Then test current row/col with all previous rows/cols one by one and get the result. Here we ignore the cases where current row/col has not been fully filled.

- Correct operations:



```
summerwang@lawn-128-61-56-202 HW4 %  gcc main.c search.c -o output.out
summerwang@lawn-128-61-56-202 HW4 % ./output.out puzzle1.txt
X       X       O       X       O       O
O       O       X       X       O       X
O       X       O       O       X       X
X       X       O       O       X       O
X       O       X       X       O       O
O       O       X       O       X       X
Test 1: 1
Test 2: 1
Test 3: 1
summerwang@lawn-128-61-56-202 HW4 % ./output.out puzzle2.txt
X       X       O       X       O       O       X       X       O       O
X       O       X       O       X       O       X       X       O       O
O       O       X       X       O       X       O       O       X       X
O       X       O       X       O       X       X       O       O       X
X       X       O       O       X       O       O       X       X       O
O       O       X       X       O       O       X       X       O       X
O       O       X       O       X       X       O       O       X       X
X       X       O       X       O       X       O       X       O       O
X       O       X       O       X       O       X       O       X       O
O       X       O       O       X       X       O       O       X       X
Test 1: 1
Test 2: 1
Test 3: 1
```

Here we did not limit the puzzle size, but use the state space search associate with CSP. So if we can represent the question with available domains, the puzzle problem would be solved correctly.

Division of labors:

- Wenyue Wang: read in data, write helper functions, main while loop
- Shushu Zhao: main while loop, final result test functions