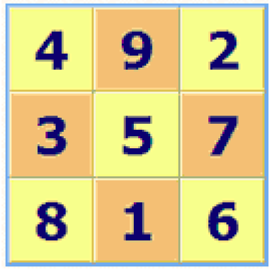
Fun Sudoku Solver

DEAN HUANG

113DB DESIGN PROJECT

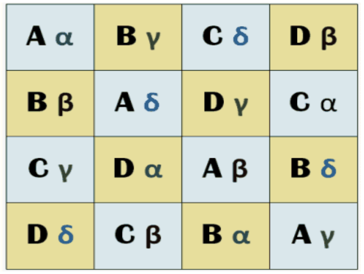
PROFESSOR BRIGGS

Brief History of Sudoku



4	9	2
3	5	7
8	1	6

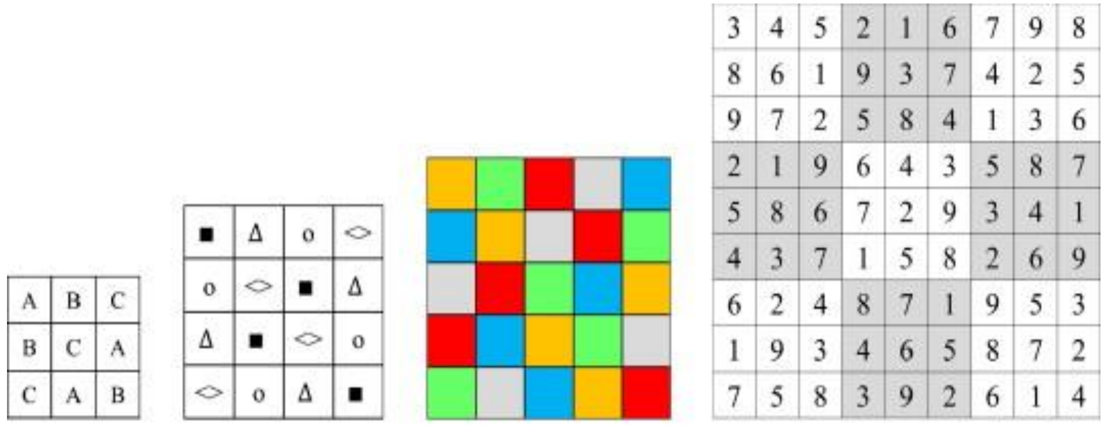
Magic Squares [1]



A α	B γ	C δ	D β
B β	A δ	D γ	C α
C γ	D α	A β	B δ
D δ	C β	B α	A γ

Greek-Latin Squares (Euler) [1]

Howard Garns in 1979 [1]



A	B	C
B	C	A
C	A	B

(a) 3×3

■	Δ	o	◇
o	◇	■	Δ
Δ	■	◇	o
◇	o	Δ	■

(b) 4×4

🟧	🟩	🟥	⬜	🟦
🟦	🟧	⬜	🟥	🟩
⬜	🟥	🟩	🟦	🟧
🟥	🟦	🟧	🟩	⬜
🟩	⬜	🟦	🟧	🟥

(c) 5×5

3	4	5	2	1	6	7	9	8
8	6	1	9	3	7	4	2	5
9	7	2	5	8	4	1	3	6
2	1	9	6	4	3	5	8	7
5	8	6	7	2	9	3	4	1
4	3	7	1	5	8	2	6	9
6	2	4	8	7	1	9	5	3
1	9	3	4	6	5	8	7	2
7	5	8	3	9	2	6	1	4

(d) 9×9

Varying Latin Squares [1]

Two Solvers

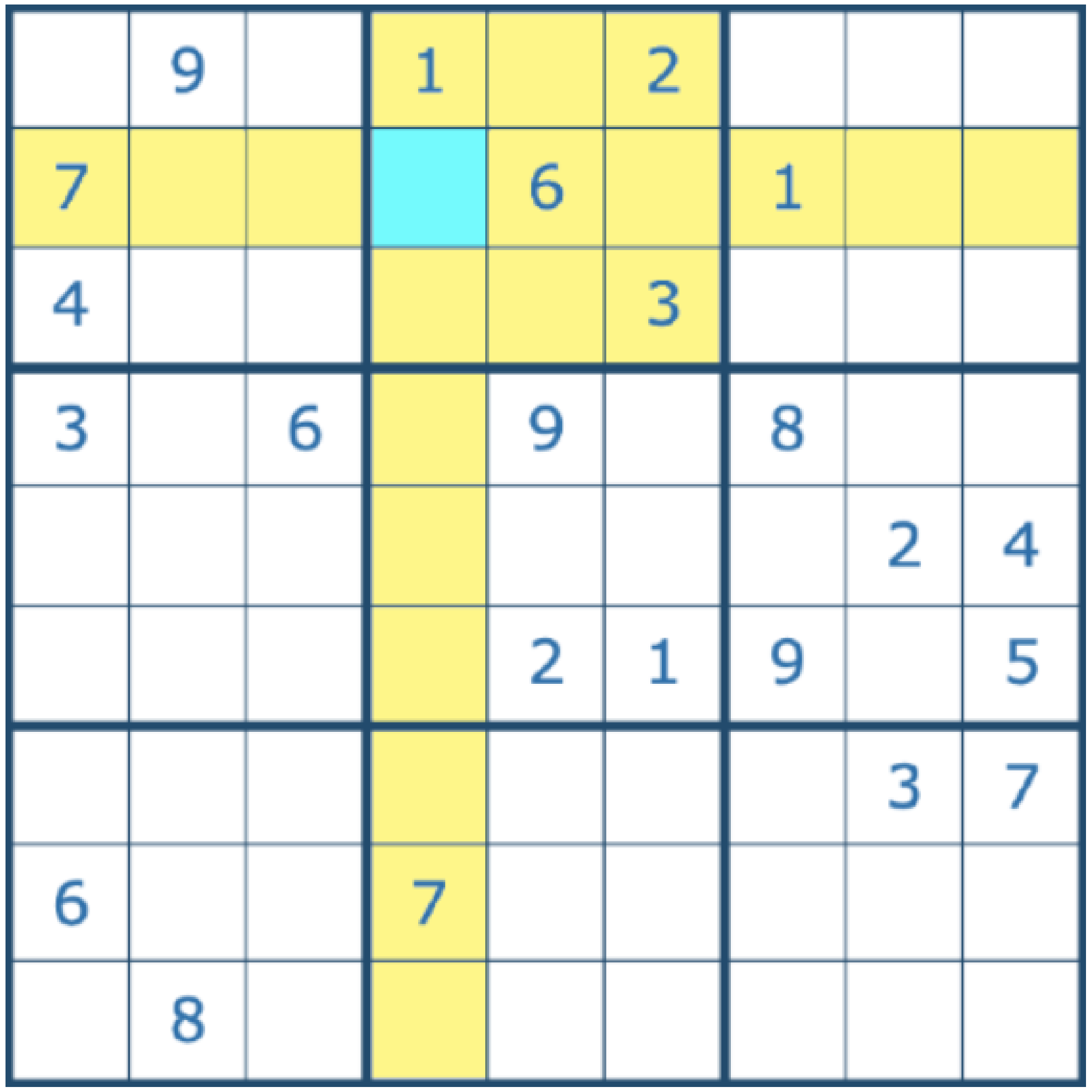
- Contract Solving (Easy – Medium)
  - Single Suspect Filling
  - Unique Filling
    - Row, Column, and Box

- Guess Solving (Hard – Expert)
  - Binary Search Tree to find valid solution

Contracts:

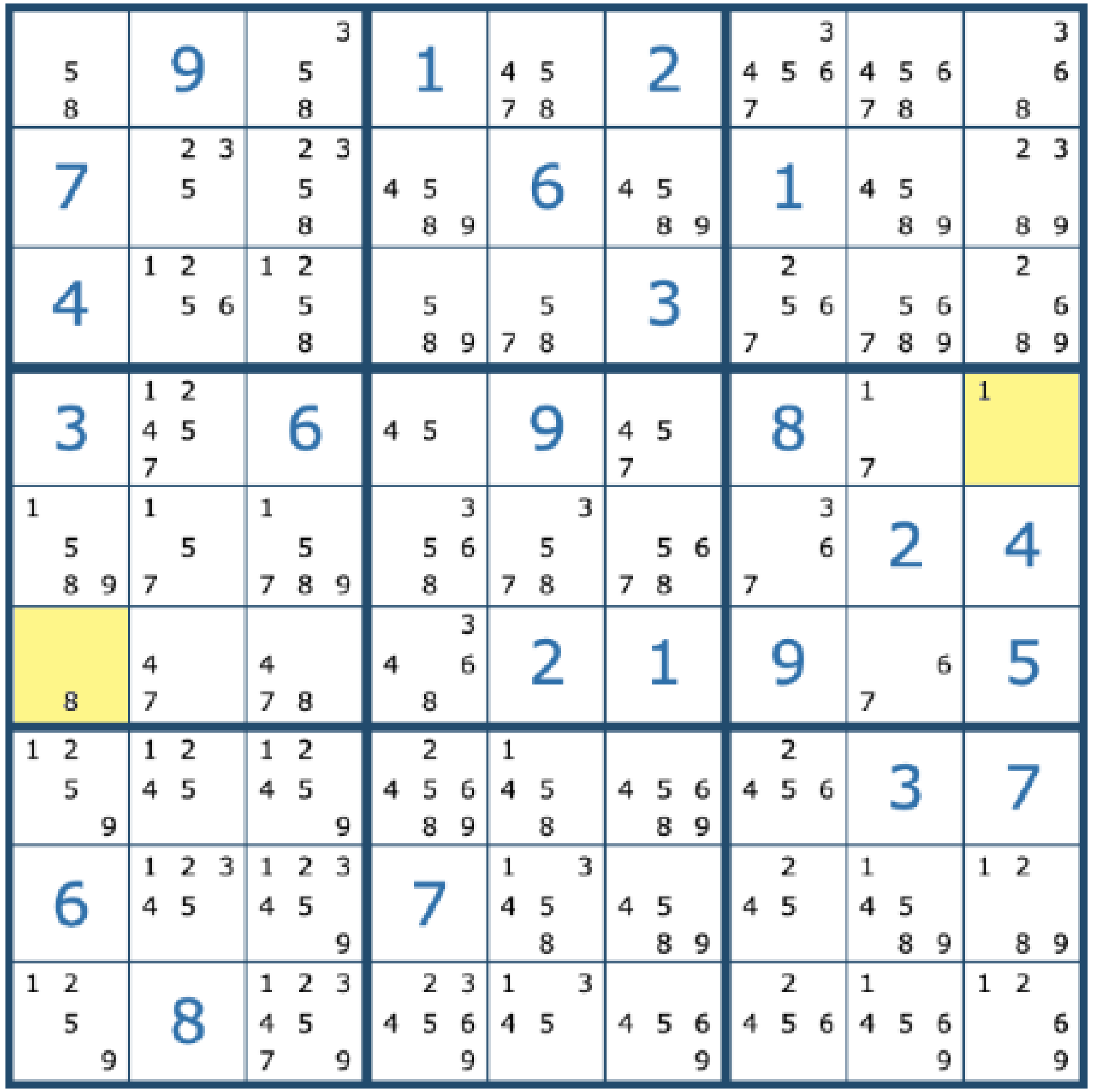
The sphere of influence due to the blue unit.

If blue is “n”, no other yellow unit could be “n”.



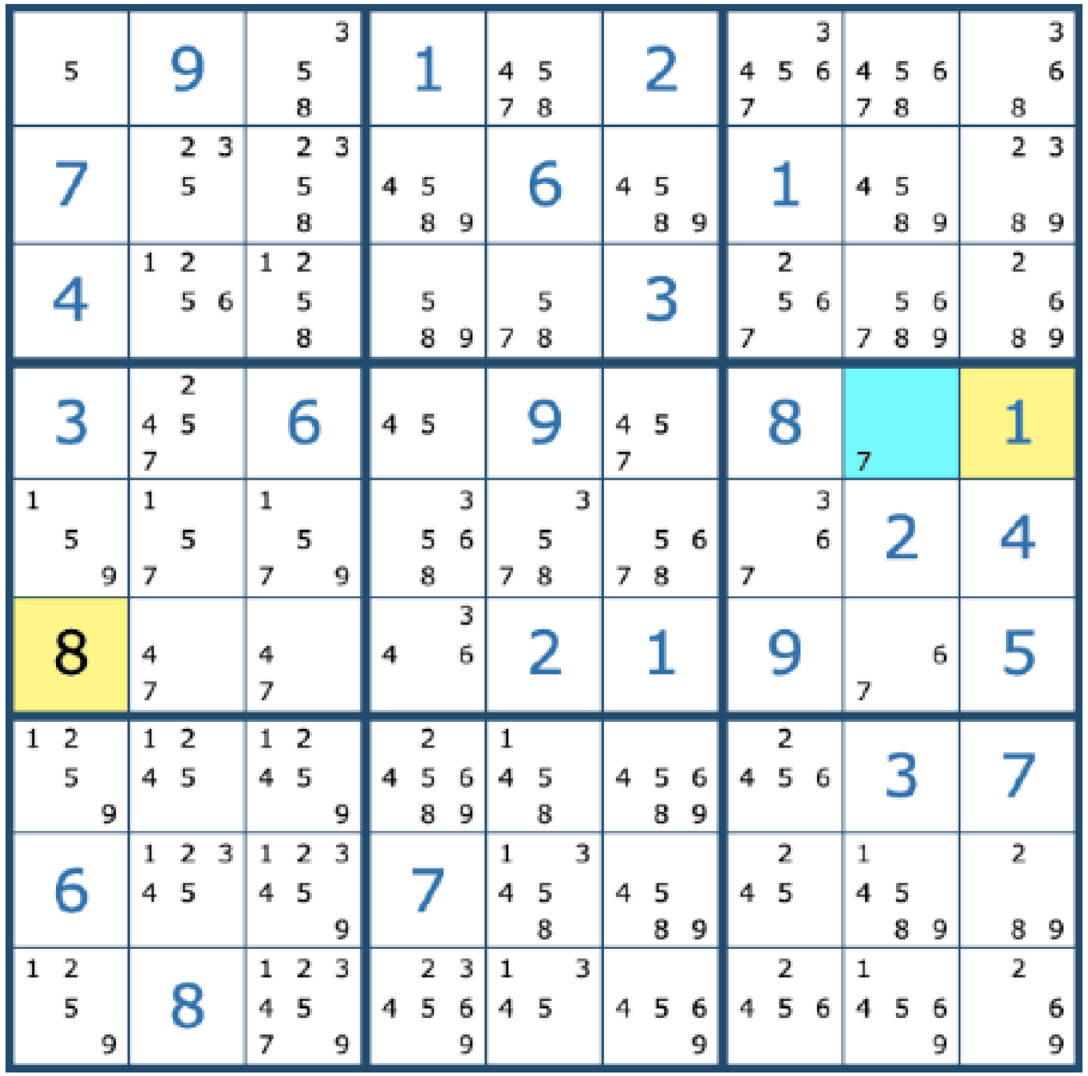
	9		1		2			
7				6		1		
4					3			
3		6		9		8		
							2	4
				2	1	9		5
							3	7
6			7					
	8							

Contract Solving: Single Suspect Before



5 8	9	3 5 8	1	4 5 7 8	2	3 4 5 6 7	4 5 6 7 8	3 6 8
7	2 3 5	2 3 5 8	4 5 8 9	6	4 5 8 9	1	4 5 8 9	2 3 8 9
4	1 2 5 6	1 2 5 8	5 8 9	5 7 8	3	2 5 6 7	5 6 7 8 9	2 6 8 9
3	1 2 4 5 7	6	4 5	9	4 5 7	8	1 7	1
1 5 8 9	1 5 7	1 5 7 8 9	3 5 6 8	3 5 7 8	5 6 7 8	3 6 7	2	4
8	4 7	4 7 8	3 4 6 8	2	1	9	6 7	5
1 2 5 9	1 2 4 5	1 2 4 5 9	2 4 5 6 8 9	1 4 5 8	4 5 6 8 9	2 4 5 6	3	7
6	1 2 3 4 5	1 2 3 4 5 9	7	1 3 4 5 8	4 5 8 9	2 4 5	1 4 5 8 9	1 2 8 9
1 2 5 9	8	1 2 3 4 5 7 9	2 3 4 5 6 9	1 3 4 5	4 5 6 9	2 4 5 6	1 4 5 6 9	1 2 6 9

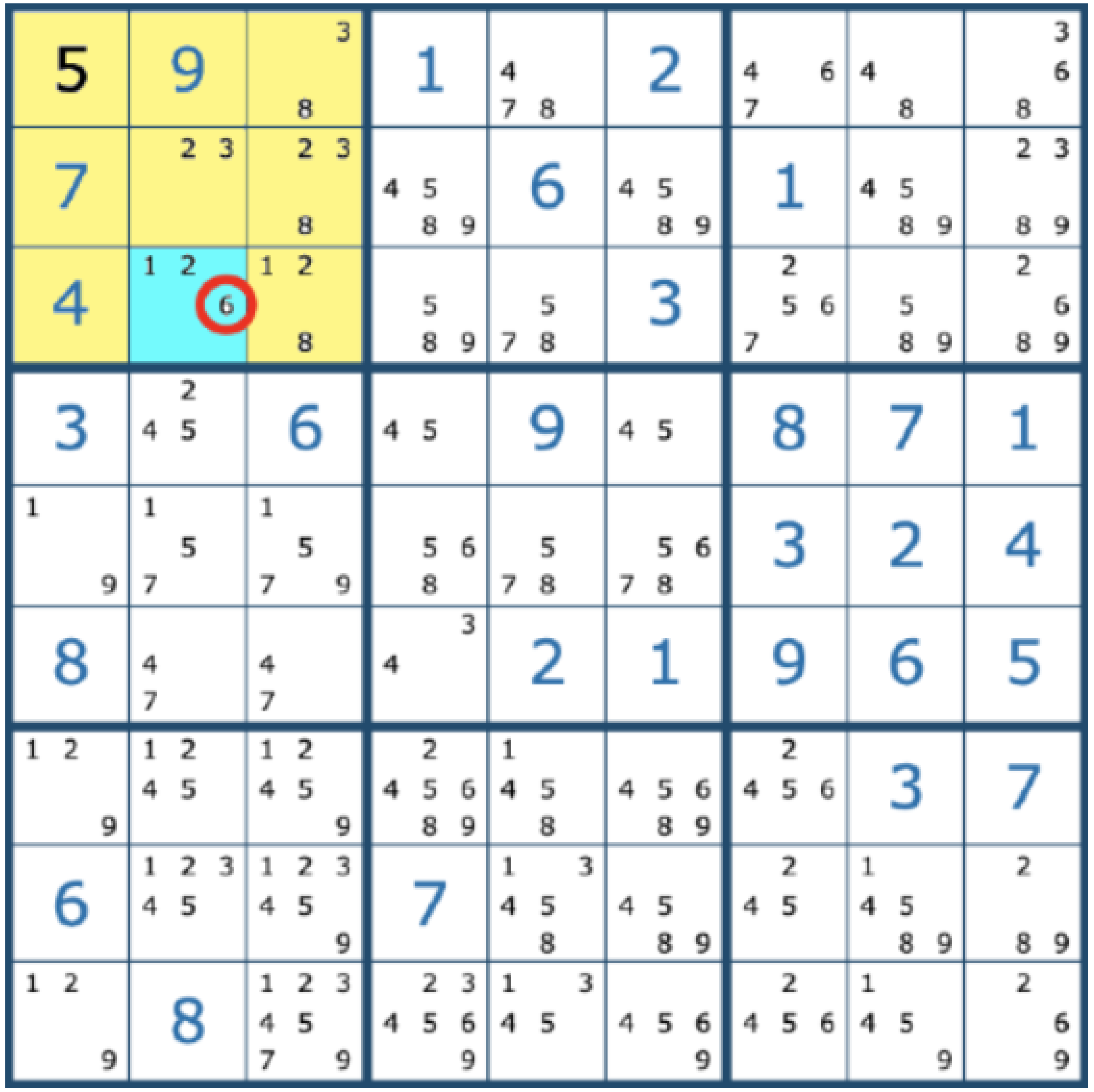
Contract Solving: Single Suspect After



5	**9**	3 5 8	**1**	4 5 7 8	**2**	3 4 5 6 7	4 5 6 7 8	3 6 8
7	2 3 5	2 3 5 8	4 5 8 9	**6**	4 5 8 9	**1**	4 5 8 9	2 3 8 9
4	1 2 5 6	1 2 5 8	5 8 9	5 7 8	**3**	2 5 6 7	5 6 7 8 9	2 6 8 9
3	2 4 5 7	**6**	4 5	**9**	4 5 7	**8**	7	**1**
1 5 9	1 5 7	1 5 7 9	3 5 6 8	3 5 7 8	5 6 7 8	3 6 7	**2**	**4**
8	4 7	4 7	3 4 6	**2**	**1**	**9**	6 7	**5**
1 2 5 9	1 2 4 5	1 2 4 5 9	2 4 5 6 8 9	1 4 5 8	4 5 6 8 9	2 4 5 6	**3**	**7**
6	1 2 3 4 5	1 2 3 4 5 9	**7**	1 3 4 5 8	4 5 8 9	2 4 5	1 4 5 8 9	2 8 9
1 2 5 9	**8**	1 2 3 4 5 7 9	2 3 4 5 6 9	1 3 4 5	4 5 6 9	2 4 5 6	1 4 5 6 9	2 6 9

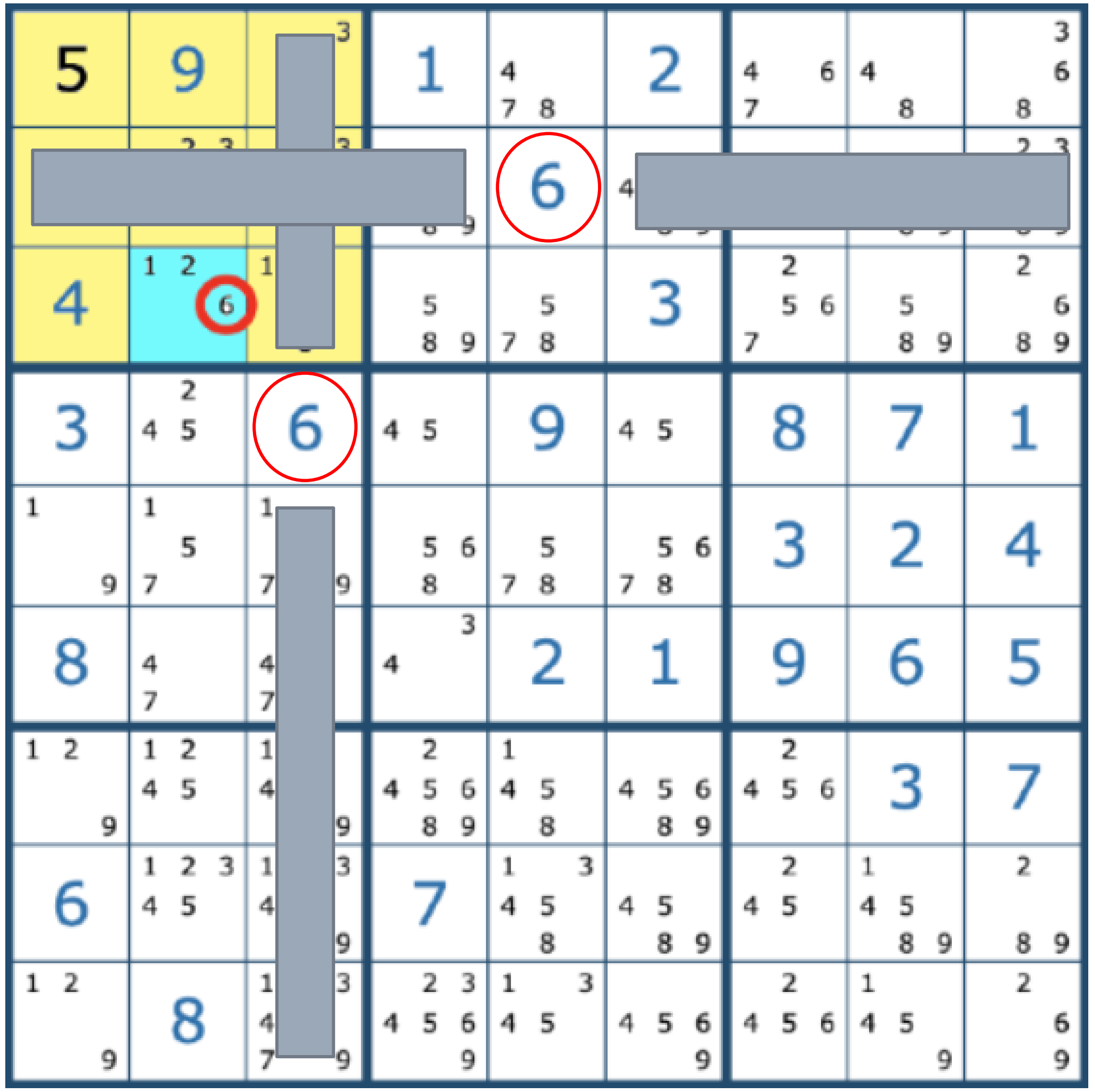
Single Suspect Fill [2]

Contract Solving: Unique Box Before


Unique Box Fill [2]

Contract Solving: Unique Box After

Dean Logic: “6” has nowhere else to run!

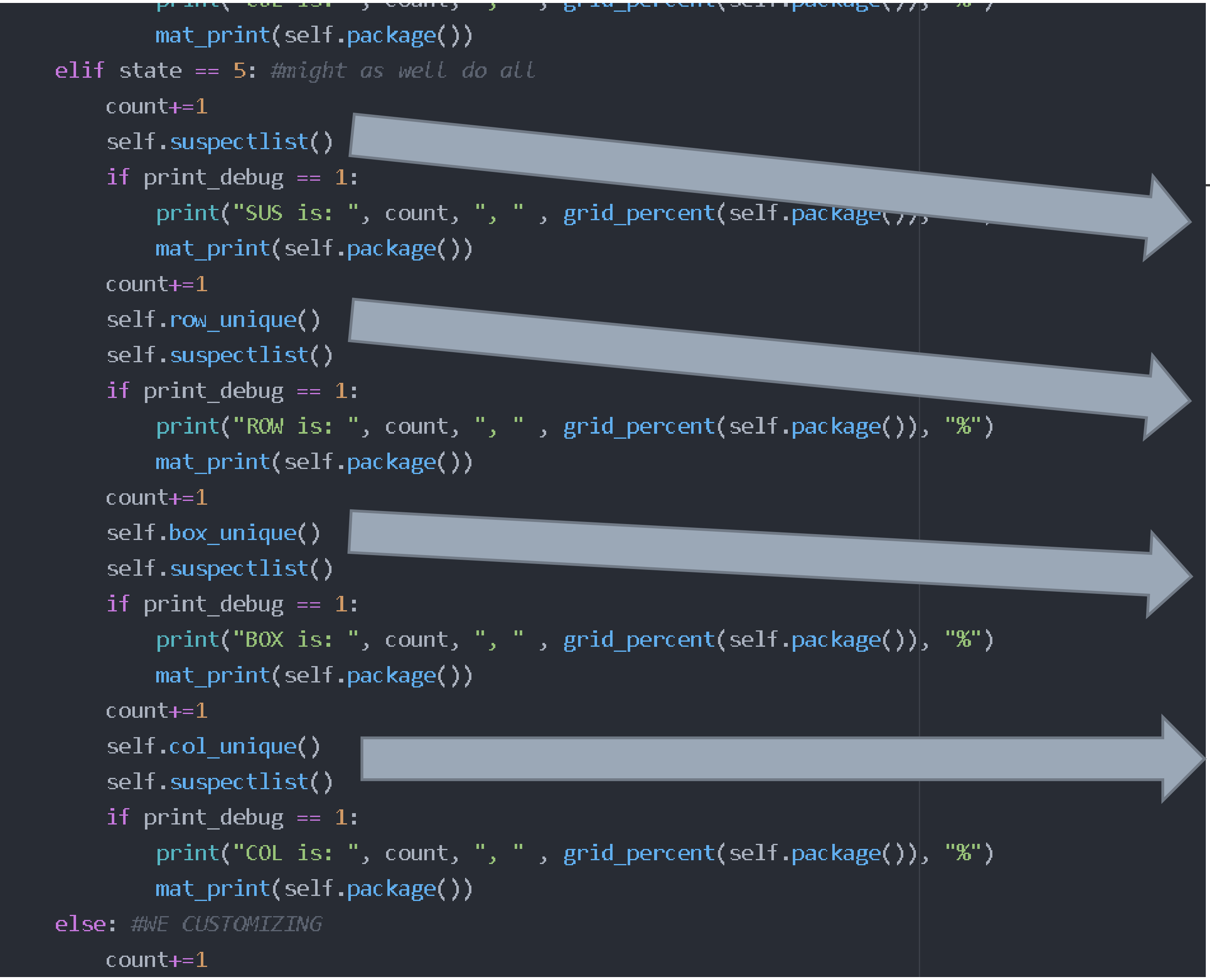


Unique Box Fill [2]

Contract Solver: Unique cont.

- ➢Due to exclusionary logic, if one unique contract solver (row, column, or box) finds an unique suspect, the unit HAS to be the unique suspect. Will never be wrong…
- ➢Create separate row, column, and box unique finders since they could uncover each other’s blind spots.

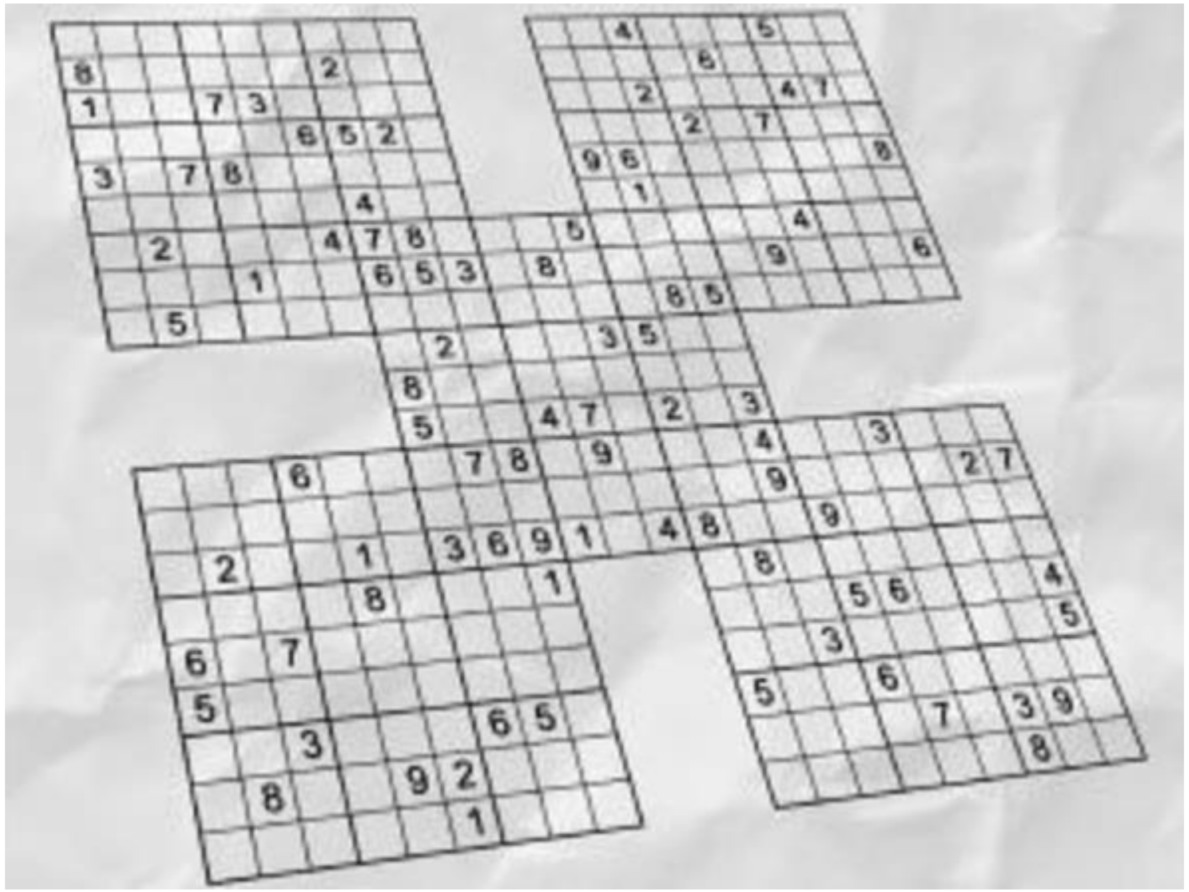
Contract Solver: All Together



```
        mat_print(self.package())
elif state == 5: #might as well do all
    count+=1
    self.suspectlist()
    if print_debug == 1:
        print("SUS is: ", count, ", " , grid_percent(self.package()),
        mat_print(self.package())
    count+=1
    self.row_unique()
    self.suspectlist()
    if print_debug == 1:
        print("ROW is: ", count, ", " , grid_percent(self.package()), "%")
        mat_print(self.package())
    count+=1
    self.box_unique()
    self.suspectlist()
    if print_debug == 1:
        print("BOX is: ", count, ", " , grid_percent(self.package()), "%")
        mat_print(self.package())
    count+=1
    self.col_unique()
    self.suspectlist()
    if print_debug == 1:
        print("COL is: ", count, ", " , grid_percent(self.package()), "%")
        mat_print(self.package())
else: #WE CUSTOMIZING
    count+=1
```

1. Suspect Fill
2. Row Unique
3. Box Unique
4. Column Unique

Guess Solver: Hard - Expert



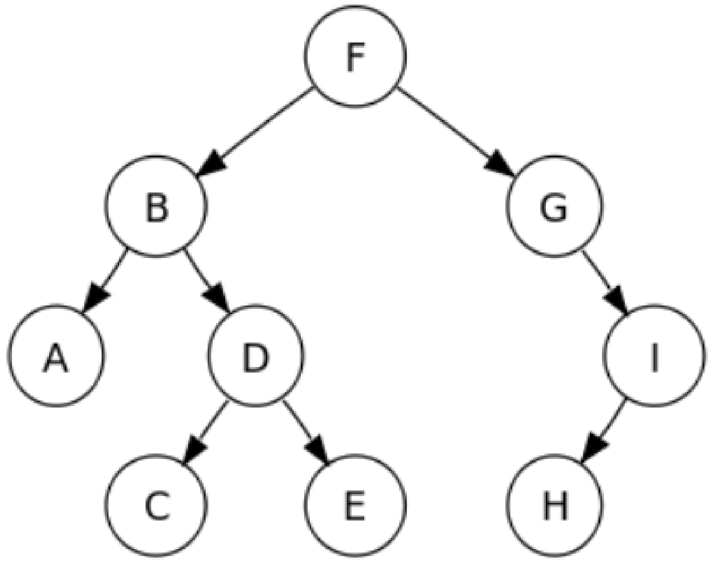
Guess Solver: Hard - Expert

- ➢After solving all the introductory puzzles online, I challenged the hard-expert puzzles. To my surprise, the Contract Solver only solved the grid by ~ +5% and gets stuck.

- ➢Solution:
  - ➢Create a smarter solver in case Contract Solver gets stuck…

Guess Solver: BST

- Method:



Binary Search Tree [2]

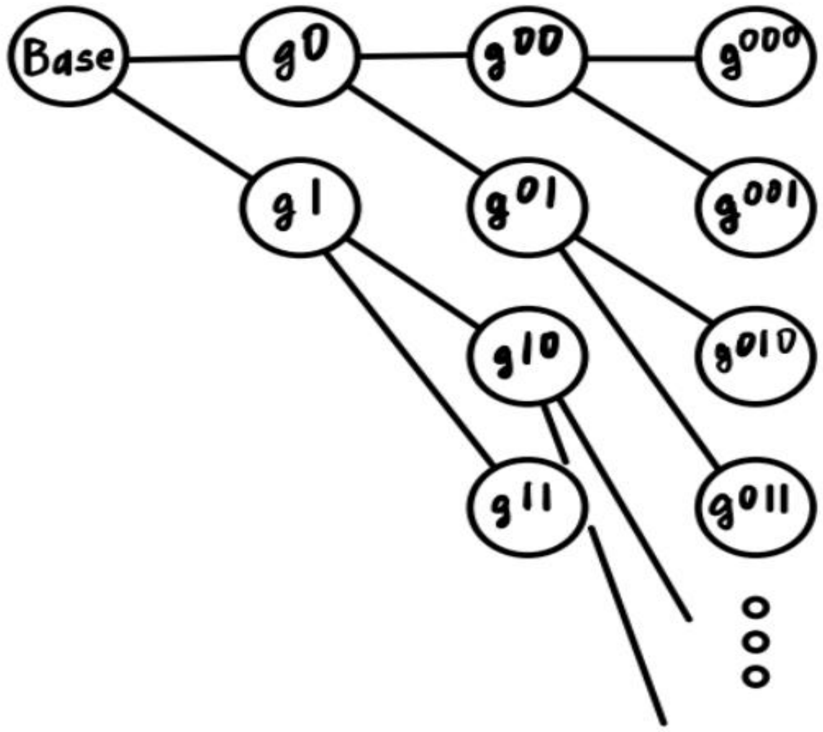
Guess Solver: BST

- Base Node will always have one right answer. We can trust the contract solvers.

- The 2 return cases:
  1. Stuck? Return Failed Grid
  2. Success? Return Solved Grid

Guess Solver: BST Instruction

1. Try guess 1. If it works continue. If it is stuck, go to step 2.
2. Try guess 2. If it works continue. If it is stuck, go to step 3.
3. Since both guess 1 and guess 2 are stuck, this entire endeavor has been futile. Go back to the parent's guess 2.



Guess Solver: BST Instruction

- That is the overall idea, but to implement it took some critical thinking.
- When is a puzzle stuck?

1. An empty unit has NO suspects (impossible), shows contradiction
2. Invalid (same number in a contract, like two “7”s in a box)

Guess Solver: Stuck?

- If the current guess is stuck, it must return a failed grid.
  - failed grid = [0 0 0 … 0]
- The purpose of a failed grid is so that later when I check it, the sum of the failed grid is 0. This lets me know **the guess has failed**!

Guess Solver: Done

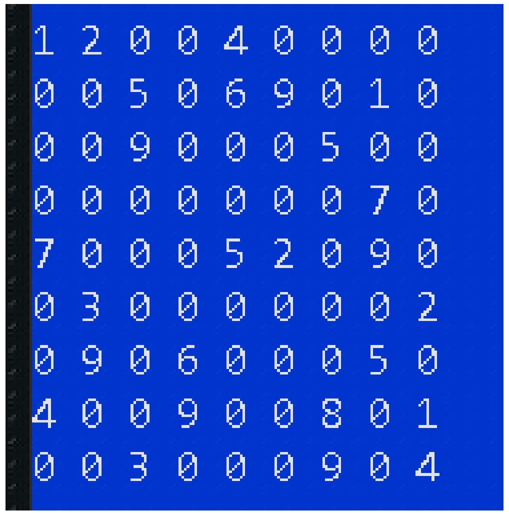
- We know the Guess Solver is done when **the total sum of the grid is** 405 and **the grid is valid**.
- If this condition is met, the grid will be considered solved, and recursively sent back to the top. The grid returned will be solved.

Guess Solver: Weakness

- After finishing the Guess All algorithm, I realized one fatal flaw.
- If I ever encounter a puzzle where the lowest suspect list is a 3, then my Guess All solver will also be rendered helpless.
- Future project:
  - Create an even more robust “guess_all_all” that increases the minimum suspect list to 3, 4, 5… until the minimum suspect list length has been reached.
  - If a suspect list of 3 is found, recursively call the function again, starting at 2.

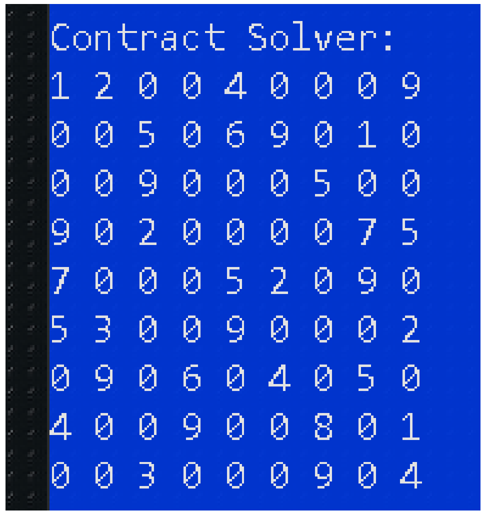
Summary

My program can solve every possible sudoku puzzle that has a minimum suspect list of 2.



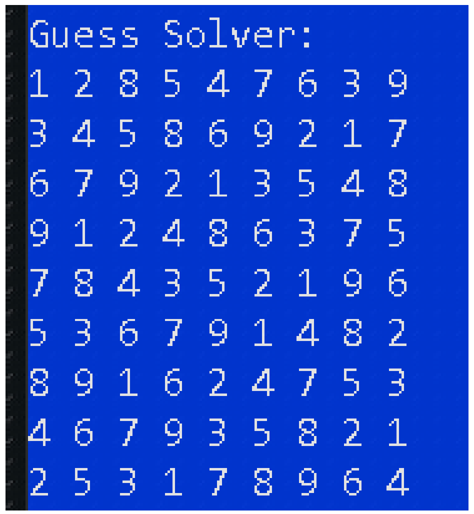
```
1 2 0 0 4 0 0 0 0
0 0 5 0 6 9 0 1 0
0 0 9 0 0 0 5 0 0
0 0 0 0 0 0 0 7 0
7 0 0 0 5 2 0 9 0
0 3 0 0 0 0 0 0 2
0 9 0 6 0 0 0 5 0
4 0 0 9 0 0 8 0 1
0 0 3 0 0 0 9 0 4
```

Unsolved (32%)



```
Contract Solver:
1 2 0 0 4 0 0 0 9
0 0 5 0 6 9 0 1 0
0 0 9 0 0 0 5 0 0
9 0 2 0 0 0 0 7 5
7 0 0 0 5 2 0 9 0
5 3 0 0 9 0 0 0 2
0 9 0 6 0 4 0 5 0
4 0 0 9 0 0 8 0 1
0 0 3 0 0 0 9 0 4
```

Contract Solved (40%)



```
Guess Solver:
1 2 8 5 4 7 6 3 9
3 4 5 8 6 9 2 1 7
6 7 9 2 1 3 5 4 8
9 1 2 4 8 6 3 7 5
7 8 4 3 5 2 1 9 6
5 3 6 7 9 1 4 8 2
8 9 1 6 2 4 7 5 3
4 6 7 9 3 5 8 2 1
2 5 3 1 7 8 9 6 4
```

Guess Solved (100%)

References

1. “History of Sudoku,” *The origin of the Sudoku Puzzle*. [Online]. Available: https://www.sudokudragon.com/sudokuhistory.htm. [Accessed: 21-Mar-2021].

2. Y. Wu, Y. Zhou, J. P. Noonan, and S. Agaian, “Design of image cipher using latin squares,” *Information Sciences*, 27-Nov-2013. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0020025513008359. [Accessed: 21-Mar-2021].

3. B. Carnes, “How to Play and Win Sudoku - Using Math and Machine Learning to Solve Every Sudoku Puzzle,” *freeCodeCamp.org*, 04-Oct-2019. [Online]. Available: https://www.freecodecamp.org/news/how-to-play-and-win-sudoku-using-math-and-machine-learning-to-solve-every-sudoku-puzzle/. [Accessed: 21-Mar-2021].