Læta’s Sudoku Story

I set out to write a sudoku solver just for the sake of personal fun. Of course, there are already millions of them out there; however, at the time, the gauge they all used to rate their success was speed. Since this held no interest to me, the quest I gave myself instead was to do it with logic only. To provide an extra level of personal achievement, I intentionally avoided looking online at anyone else’s ideas. I started out by manually solving some puzzles and then translating what I did as a human into algorithms. A majority of my resulting code was created in this manner. Eventually I had something I thought was decent so, naturally, I challenged a couple of my friends to a contest. They would each write a solver to compete against mine. Winner would be determined by adding a counter into the code to record how many times backtracking was needed. This counter indicated how many times our logic rules failed and the program had to resort to brute force. Because I felt a couple of my ideas were unusually creative, I did not restrict my friends from using the internet as I had done; in fact, it was at this point that I went online and actively searched for information to help them.

I admit, I was highly disappointed at what I found. I had such hopes of fantastic new algorithms to improve their code. While there are thousands of sudoku websites, they are all clones, simply parroting each other. They all have the same dozen or so methods, which they all feel the need to tediously explain in their own words, like somehow they can then take credit, but the truth is, none of them had anything original to say. They did have a few things which I had not thought of such as fish but, as I later realized, they had limited conceptual understanding of it. While both of my friends are significantly better programmers than I am, it was my imagination that won the day. Some of the things I came up with, I was unable to find anywhere else. Thus, I am writing them here so they can be of use to others. I don’t mind if you use them with credit given to me, although I shudder to think this could create a new wave of clones.

While my code is commented, I will use the space here to add further details and explain how mine differs from others. I wrote most of this several years ago, then updated it when Python 3.8.2 came out and just never got around to posting it online. I enjoy python; high level languages improve readability, and their syntactic sugar simplifies a lot. Almost immediately, you will see an example of this in the data structures I define. Rather than keep track of the numbers in each cell of the grid, I want to know which numbers are *possible* in each cell. Thus, if a cell has a 3 in it, I can remove 3 from the possibilities of the cells next to it.

During my internet search to help my friends, I was also collecting any very difficult puzzles I came across for us to test on. I found about half a dozen lists of what were claimed to be the hardest puzzles people could find. I merged all these lists together, sifted out the duplicates, removed any that my program could solve with zero backtracks, sorted the rest based on the count to estimate difficulty and saved it to a file. I wrote the chaining rule after this process so if anyone looks too hard, they might find a few out of order. Of the nearly 5.5 billion theoretically possible puzzles, these 1396 are the result, which my algorithm cannot solve with pure logic alone. I’m sure there are plenty of others; if you find any, please send them to me. I do not claim credit for any of them. From this file, I took the top 5, which are the easiest, the middle 5 and the final 6 (hardest) and coded them into the program so you have test cases to run without calling on the file. If you want to use the file, keep it in the same folder as the program or else you’ll need to change the path on line 243.

The methods at the top are just for utility and should be self-explanatory. The prop() method is when things start getting interesting. Every time there is a change in the grid, this will spread that change and try to deduce new conclusions. It does this by taking each group and running the logic rules on it. A group here is defined as all the cells in the same row, column, or box. The logic rules are sorted by how powerful they are so the one I call void permutation is first because it reduces the count more than any other rule (in fact, it’s almost as good as all the other rules combined while only using 1/3 of the time).

Void permutation solves naked subsets and hidden subsets. I am particularly proud of this one for two reasons. Most people write a method to find pairs, another method to find triples, another for quads etc. I analyzed all these to find what they had in common, kept those sections and made the rest into a variable. That means these 8 lines can find all pairs, triples… any number, all in one. That is the permutation part. The other noteworthy thing I did is the void part. After I wrote all the algorithms for how humans solve sudoku, I started imagining how a machine might do it. If it wasn’t limited to human thought, what strategy might it use? I realized it could look at all the possibilities of the blank cells. This means, rather than look where numbers are, this looks where they aren’t. This algorithm runs on the spaces of the puzzle rather than the numbers. I believe this increases its effectiveness because, just looking at the numbers is a special case; however, looking at where they could be is a generalization which can catch more information.

The next most useful rule is surprisingly, singleton. Due to efficiency, this was pulled out of rules() and placed back in prop(). It catches full house and last digit situations.

Next is void 2 of 3. This solves locked cases. Again, I repeated my use of the void strategy by focusing on the empty cells rather than the full ones.

The final four are all mathematically technical. I was unaware of them as a layman and didn’t add them until after I started searching around on the internet. They are rather costly resource wise and have little payoff since they are significantly less useful than the previous rules. But they do help, and when combined with the other rules, are able to squeeze out a few more counts.

Pair exclusion. Wow. This has got to be the most misunderstood concept in the entire sudoku community. This is the name I use to refer to ALL fish: x-wing, swordfish, jellyfish etc. I analyzed these, not to write code, but to truly understand how they work. What I discovered caused me to do a double take. I had to rethink it. But the conclusion was the same. So, I searched on the internet. I searched hard. There was no hint of what I had found. Remember the definition of a group: all the cells in the same row or column or box. Simply start with two cells in the same group. Follow them along parallel rows or columns to two more cells in a different group. With these endpoints, you can exclude other locations **within those groups** the same as you would with any other fish. Let’s use an x-wing as an example. It starts with two cells in the same row, follows them along columns to two cells in a different row. This is the normal case because it requires you to start with a row, but do you really need to? Nope. Your initial cells could be in the same box instead. Follow these two cells vertically to a different box and you are still able to use exclusion. The groups don’t even have to be the same. You could start in a box and end with a column. Why not? The logic is the same from any group to any other group. These new grouped fish… g-fish… goldfish?... is a new way to exclude possibilities that I have not seen documented anywhere.

Initially, I wrote my goldfish for 2 pairs (x-wing). I actually wrote it several times with a different focus in each algorithm. On my first try, I defined the sections of code by the shape the pairs made: band, stack or rectangle. This worked. But I was curious and wanted to experiment. I’m really glad I did because it led to an astonishing discovery. My next attempt concentrated on the final pair: were they in the same row, column or box. This also worked. But while testing it, I found some fascinating overlap. My void 2 of 3 algorithm was already finding the box portion of this 2 pair exclusion! At first it may seem like black magic but, if you follow the logic thru on each rule, it really does make sense that they perform the same job via vastly different means. Because pair exclusion is the most expensive in terms of runtime, I elected to remove this box part from my program. This is one of the sources of error in the following table.

Next, I wrote 3 pair exclusion (swordfish). I wrote this algorithm several times as well. The first way, I used a set of constraints as the means of determining the presence of a swordfish. But when I rewrote it to simply check for the end configuration, I noticed the constraints took on an unintended side effect: they shortened the runtime. So, I merged these two styles and spent some time experimenting to optimize the variables without changing the count: lines 378-379 are the result. They are completely unnecessary to the logic; they are only there because they cut the runtime in half. Part of the reason for this is that it does not allow all three pairs to be in the same band or stack. I don’t have proof, but I think those are impossible. If it turns out they aren’t however, then this would be a source of error in my code.

Next, I was curious why people only expand 2 pairs upwards to 3 and 4 pairs but never downward to one pair so I decided to try it and see what would happen. To my surprise, it worked. But it also overlaps void 2 of 3 so I sadly removed it. Doing so however, drastically changes the following table. With 1 pair included, we get the 7.7m number shown; without it, the count jumps up to almost 30m!

Finally, I tackled the fabled 4 pair exclusion (jellyfish). As in swordfish, I used constraints to shorten runtime. However, unlike swordfish, these are based on logic and won’t introduce a new source of error. They are rather unwieldy tho, taking up 11 lines (400-410). As you probably do, I too have the urge to condense them down using union() to improve readability; however, doing so quadruples runtime. It should be noted jellyfish are very rare. In my whole file, there are only about a dozen IF all my other logic rules are active. These also behave very strangely. They seem to require other rules to remove possibilities in order to gain a foothold. Thus, if I comment out some rules, the number of jellyfish found will decrease. Due to this oddness, each of my search spaces will find jellyfish in different puzzles. Even when one is found and it removes a possibility of its own, it's not important enough to lower the count. In fact, in my whole file, the total count will not change with jellyfish active. All it does is increase the runtime by an order of magnitude. That means my laptop can run the file in 4 min without jellyfish or 40 min with it. Due to its lack of usefulness and absurd runtime, jellyfish starts off commented out.

Next is something I call chaining but I don’t like this name. Several other concepts use this name as well so it is very ambiguous; however, I couldn’t think of anything better.

Next, pair deduction is what others commonly refer to as coloring.

Lastly, the obscure hook, sometimes called x-y wing, is the least useful.

Here is the data I collected when I commented out all the rules except one to get their power rankings. I ran the entire file using the sorted possibilities heuristic.

|  |  |  |
| --- | --- | --- |
| **Logic Rules** | **Total Count** | **Average Count** |
| All Together | 8,610 | 6 |
| Void Permutations | 63,810 | 46 |
| Singleton | 238,228 | 171 |
| Void 2 of 3 | 1,331,081 | 953 |
| Pair Exclusion | 7,795,138\* | 5584 |
| Chaining | 13,733,355 | 9838 |
| Pair Deduction | 26,776,624 | 19,181 |
| Hook | 30,134,702 | 21,586 |
| None (pure brute force) | 33,263,953 | 23,828 |

\*my code for pair exclusion has changed so this number is no longer accurate.

I actually wrote several other rules but, upon testing I discovered, while they do work to find numbers and lower the count, all their information was redundant and more easily found by other rules such as permutations. As they did not provide any unique information, for the sake of speed, I removed them.

Speaking of speed, void permutations, singleton, void 2 of 3 and chaining all improve the runtime. Meanwhile, pair exclusion, pair deduction and hook all increase it. Pair exclusion is the worst by far. With a count of almost 9k for the file, swordfish is only responsible for about 80. The time to get those 80 doubles the runtime. So, if you care about speed, comment that function out. Of course, all of my remarks thus far about speed and counts are largely meaningless for most puzzles; they only apply to these very few which require backtracking. For the rest, they will be solved almost instantly so runtime is insignificant and you can leave everything uncommented (except jellyfish).

If all these rules should fail to complete the grid, the solve function is brute force. Since I only care about the count, that is all I have it display but, if you want to see the puzzle’s answer, change line 484 from:

if 0 not in grid: return True

To these 3 lines:

if 0 not in grid:

printgrid(grid)

return True

There are also three search spaces here: an unsorted control and two heuristics which usually get better results.

At the very bottom are the controls. Use either 502 or 503 if you only want to solve one grid. These two lines are paired with 505-508. If you prefer to solve multiple grids at once, comment out 502-508 and choose to uncomment one line among 510-513.

There is one more topic I wish to address. I failed my quest. I was not able to find logic rules which solve every sudoku. I did however, come closer than any other code I’ve seen. 1396 out of 5.5b is amazing! This doesn’t mean someone can’t do better but it sadly won’t be me because I have developed some medical issues which have put an end to my programming days. If you run every puzzle in my file, the average count is only 6 so, if you can squeeze out only 1 or 2 more numbers, it would solve several hundred of them right away. The problem is that other people out there claim to have done it already. From what I can tell, this boils down to a difference in definitions. To them, using only logic rules means no brute force. To me, it means never placing a number unless you KNOW it is correct, i.e. no backtracking. A backtrack means you tried something which turned out to be incorrect and have to undo it to make another choice. It amounts to a failed guess which, to me, is not logic. Suppose for example, instead of using brute force on each consecutive cell in the grid, you use it on each number, meaning you find the location of every number 1, then every number 2 etc. This is one (of several) definitions I saw for the technique called nishio. Since it’s not called brute force, those other people say it solves the sudoku only using logic rules. They argue they are testing a single number until they find a contradiction in order to eliminate possibilities. But really, the same argument can be applied to brute force, can’t it? Brute force is technically logical in identically the same way. This view is unfortunately very common and there are many ways of disguising backtracking behind other names. Some examples are: ariadne’s thread, trial and error, forcing, tabling, bowman bingo, nishio, airoot and dancing links. That is why I didn’t code any of these; if I’m going to use backtracking, traditional brute force is much simpler. If you have written your own code and wish to convert it to count how many times your logic rules fall short, simply comment out any rules that use backtracking except for brute force, then move the counter from after the recursive call to the line directly *before* it. I do not believe anyone has solved all the puzzles on my list without using some form of backtracking, but I remain hopeful that new rules may be created in the future to do so 😊