Project Title: Solving Sudoku

Developed by: J. Marcus Hughes
Date: 2 August 2019

Project Description:

In this project, we learn how to solve difficult games like Sudoku. We will implement a Sudoku object, a tool to generate new puzzles, and software to solve a given puzzle. We will discuss greedy algorithms and complexity of problems.

Pre-Requisites:

Python Year 1 Course, a familiarity with recursion.

Objectives:

- Develop software to solve Sudoku puzzles
- Introduce search algorithms
- Understand depth-first search
- Familiarize with problem complexity

Concepts Covered:

- Constraint satisfaction
- Depth-first search
- Back-tracking search

Estimated Hours to Complete:

5 hours

Resources Needed:

None

Additional Information:

The approach and code is based on Peter Norvig's essay about solving all standard Sudoku puzzles.

PROJECT INSTRUCTIONS

Background

Ever tried to solve a Sudoku puzzle? They are slow and time consuming by hand, but the rules are very simple. It is a great task for a computer. In this project, we will write a program that solves any regular Sudoku puzzle.

Overview

We will approach this project in three parts: reading in puzzles, attempting to solve them a way you might be used to called constraint propagation, and then attempting to find the solution by searching for it through a kind of guess-and-check pattern. The overall steps are as follows:

1. Familiarize yourself with Sudoku's rules.
2. Familiarize yourself with the format of the example puzzles.
3. Write a function to read in one of the puzzle files.
4. Represent the possibilities remaining for each square as a dictionary mapping from square to a string of possibilities.
5. Implement constraint propagation using the idea that if a square has only one possible value that has to be it or that if a unit has only one possible place for a value, that value must be there.
6. Implement a depth-first search by guessing and checking for no contradictions.

PREPARING

Overview

- Understand the rules of Sudoku
- Introduce constraint satisfaction

Instructions

Understanding the project

Let's start by remembering how Sudoku works.

	1	2	3	4	5	6	7	8	9
A		2			8	7			
B			3			1		2	6
C								7	1
D	1			8			7	3	2
E	2		8						4
F	7	9	6			4			8
G	9	1							
H	6	5		9			2		
I				4	1			9	

Figure 1: An example Sudoku, puzzle number 10952, to solve from https://www.sudoku-solutions.com/. The solution is available at the end, Figure 2.

Standard Sudoku is a 9×9 grid of numbers, 1 through 9. For reference, we will refer to the rows by the letters A through I and the columns with numbers 1 through 9. Any square can then be identified by saying which row and column it is. The top left square then is A1 and the bottom right is I9. The board also has special *blocks*, groupings of 3×3 squares. The middle block consists of the squares D4, D5, D6, E4, E5, E6, F4, F5, and F6. We make it clear where the blocks are with bold lines. In general, we will refer to a row, column, or block as a *it*.

It would be fantastic to go through a puzzle with your student. You can go to a site like https://www.sudoku-solutions.com/ and pick the same puzzle number to talk through. Make sure they understand the square coordinates and the goal of Sudoku. You can mention how you go about thinking about an easy puzzle. Ask them to tell you the value in square F3. In the example puzzle, you can notice how square I1 cannot be a 1, 2, 4, 5, 6, 7, or 9. Thus, it's only a 3 or 8 since the other numbers are already on that row, column, or block. Similarly, E2 must be a 3.

The term unit will likely be confusing. Discuss it with your student for clarity.

You are initially given values in some squares. For example, the square A2 in Figure 1 has the value 2. The goal is to fill in the empty squares with numbers so that in the end every row has all the numbers 1 through 9, every column has the squares 1 through 9, and every block has all the numbers 1 through 9. We can succinctly state the rules to Sudoku: *A puzzle is solved if the squares in each unit are filled with some ordering of the digits 1 to 9.* Sudoku puzzles are set up so that there is always exactly one solution to them.

There are many strategies people use to solve Sudoku puzzles by hand. Feel free to read about them here. Before we think about how to get a computer to solve a puzzle, please stop and solve one yourself by hand. Good luck!

Our approach: constraint propagation and search

Computers are very literal. We have to tell them exactly what to do, not just solve this puzzle using this approach. We will use two ideas in our program: constraint propagation and search. They may sound like scary words, but they're really simple ideas. We'll go over them briefly here and then cover them in future sections. You'll want to emphasize that the easy puzzles can be solved with constraint propagation while the difficult ones require searching or some other approach.

Constraint satisfaction and propagation

Meet the Adams family: Leah and Lucy and their son Lucas. Imagine the Adams family is looking to buy a new house. Leah wants the master bedroom to have its own bathroom. Her wife, Lucy, loves to cook and wants the new house to have a big kitchen with stainless steel countertops. Everyone knows Lucas needs a big yard to run around in. Those are their constraints: a master bathroom, nice kitchen, and big yard. They go see a house and immediately knock it off their list because the kitchen is old and tiny. It does not satisfy all their constraints. They keep looking at houses until they find one that meets all their needs and thus satisfies all their constraints.

We can do the same thing with Sudoku! Each square has to have a different value than any other square in its row, column, and block. Take a look at square E2. Can it have a 1 in it? No! There's already a 1 in the same block on square D1. What about a 2? Nope, there's one in the same column on square A2. What about a 3? It's possible! There aren't any threes yet in the same row, column, or block. If you keep checking numbers you'll notice that only 3 is possible for that square. It's the only number that satisfies the constraints! So, we can just go ahead and fill it in. We're one step closer to solving the puzzle.

Go through a few examples with students. You might also talk about other constraint satisfaction problems. For example, scheduling doctor's appointments at a clinic is a constraint satisfaction problem. Each doctor needs to meet with their patients and has only so many open times. The patients also have busy schedules. Thus, the constraints are when they cannot meet together and the goal is to find a schedule so that everyone can meet as needed.

Searching

Sometimes we don't get so lucky to have squares where there is only one option. In those cases, we have to make some guesses and see if they pan out. Let's say we didn't notice that constraint satisfaction requires E2 to have value 3. We might guess that E5 will have a one in it; it does not violate any of our constraints. Then, the computer will keep making educated guesses. Either we get super lucky and find the solution by guessing. Unfortunately, we would actually likely reach a point where the rules is broken and some number is repeated in a unit. In this case, we might

realize that E2 must have a 3 in it. But, we already put an 3 in E5! That's a problem! We have learned that E5 cannot have a 3 though. Thus, in searching we systematically try all possibilities!

Why don't we just always search instead of doing that complicated constraint satisfaction procedure from before? It turns out there are just too many possibilities. If we were to try every possible solution to the Sudoku example, how many times might we try? Well we could first guess A1 is 1. That doesn't work though since D1 is 1. That's try number one. We do that again for A1 being 2 and 3. We've tried three times now! When we get to A1 being 4 there's no obvious problem with it. If you'll look at the solution, you'll notice A1 is actually 5, not 4. Thus, we might try a lot of things before we realize A1 cannot have a 4 in it.

Have the student multiply out possibilities. If each square has 9 possibilities then the grid has 9^{81} ways to fill it, most don't obey the Sudoku constraints. I'd suggest that you go through Tic-Tac-Toe with your student, searching for the optimal move. It's a small puzzle to try.

Our plan

We will approach this project with a few steps:

1. Figure out how to represent the Sudoku grid.
2. Implement constraint satisfaction
3. Implement a search algorithm

Designing tests

Before we start writing code, we should always think about some tests to make sure we know if we're correct as we work. You will want to find some example puzzles where you know the solution so that you can make sure your code outputs the correct results. We will also want to design tests for each portion, making sure that we can properly represent a grid, identify constraint satisfaction scenarios, and run the search.

New packages or functions

We are not going to use any new packages or functions in this project. We will instead by using tools you already know about, specifically dictionaries, in new ways.

Challenges

- Learn about how many possible Sudoku puzzles there are. It'll blow your mind. Here's a link to help.. The answer is astonishngly 6,670,903,752,021,072,936,960.
- Read this and then this to learn how we use logic and math to check Sudoku puzzle solutions. Expect the student to be scared by the notation here. You'll want to read it yourself to explain.

Homework

- Consider how to represent the board. Outline the methods we will want. It's okay if you don't make much progress on this. It's good to try.

Resources

- Human Sudoku strategies.
- Check out Peter Norvig's essay on solving all standard Sudoku puzzles. Peter Norvig is a famous computer scientist who has written the definitive artificial intelligence textbook.

EXECUTING: SUDOKU REPRESENTATION

Concepts

- Representing the board
- Strings as lists

Warm-up

1. Refresh yourself on lists. Specifically, if I give you two lists `[1, 2]` and `[a, b, c]`, how can you produce the list that pairs up all the elements, i.e. `[(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)]`? In math, we call this the Cartesian product. See if you can understand this solution:

```
import itertools
def cartesian_product(lsts):
    return list(itertools.product(*lsts))
```

   This is because we're using such a product to represent the squares and board state.

Instructions

Idea

Imagine we use a zero to represent a square that is empty. Then, we can just read off the squares like we were reading a book. For example, the first two rows of Figure 1 become `020087000003001026`. Clever, right? We can store a board as just a string!

This won't be beneficial when we're solving though. While we're solving, we want to know what possible values can be inserted. Thus, we will use a dictionary that maps the square to its possible values. In our example, the key `'E2'` would map to `'3'` and the key `'I1'` would map to `'38'`, meaning it is either a 3 or an 8. We will make both the key and the value strings. The value of the mapping will be all possible values in numerical order. We'll find this super handy later when we're doing the solving part. Discuss with your student on how there are many choices for representation and storage but we're using this one because it works with our solving algorithms.

Finally, how would we know all the values have been assigned? When all the values in our dictionary are just single numbers, only one number is possible. We can then check if its correct by making sure there are no repeated numbers in any units.

Steps

1. Create a method to parse an input. Let the constructor take a string like `020087000003001026` as input. You will first want to convert the string to a dictionary where the key is the coordinate and the value is what number is present in the square with that coordinate. For example, we would see something like

   ```
   {'A1':'0', 'A2':'2', 'A3':'0', ...}
   ```

   for the example in Figure 1.

2. Create a `print_grid` method for your grid that prints out the board in a pretty manner.

3. Write an `is_filled` method that checks if all the squares have been solved.

You may find it helpful to keep a list of all the squares and a dictionary mapping each square to a list of the other squares in its unit. If you find you're repeating an idea make it a variable to call from.

Example tests

Your `__str__` method should take
`003020600900305001001806400008102900700000008006708200002609500800203009005010300`
as input and create a grid like the following once solved:

```
. . 3 |. 2 . |6 . .
9 . . |3 . 5 |. . 1
. . 1 |8 . 6 |4 . .
------+------+------
. . 8 |1 . 2 |9 . .
7 . . |. . . |. . 8
. . 6 |7 . 8 |2 . .
------+------+------
. . 2 |6 . 9 |5 . .
8 . . |2 . 3 |. . 9
. . 5 |. 1 . |3 . .
```

Remember to write unit tests to make sure the last few steps perform correctly.

Challenges

- Implement a way to generate new puzzles and then hide squares from them. Here are two websites that might help you: number one and number two.
- What if instead of a 9×9 Sudoku board we wanted to do a 16×16 one? Generalize your code to take any square size.

Homework

1. Write unit tests for everything that has been designed so far: checking `is_filled`, `is_correct`, and `print_grid`. Since our code is so short, it would be okay to just use assertions in-line.

2. Add the functionality to read from a file.

EXECUTING: EASY PUZZLE SOLVING

Concepts

- Constraint satisfaction

Instructions

When you were solving a puzzle by hand, you might have noticed two helpful rules:

1. If a square has only one possible value, then eliminate that value from the square's peers.
2. If a unit has only one possible place for a value, then put the value there.

Suppose we start with an empty board, then your current solution would have every square mapped to `'123456789'` since every square could have any value. But, if you know that square `A1` has a 6 in it, then squares `A2, A3, A4, A5, A6, A7, A8`, and `A9` could not have a 6! Everything on `A1`'s row, column, and block cannot have a 6 in it. So, you'd immediately remove that option from all of them. Our algorithm will work just like that! Work through an example puzzle with your student to make sure they get the idea of propagation.

We start with an empty board and then add in the knowledge from the filled squares one at a time. You'll want to write a function `assign(solution, square, value)` that does just that. It assigns the known value to the coordinates specified in the variable square in our solution board. How do you do that? You'll write a helper function named `eliminate` that will take the solution board, the current square, and the value to eliminate as a possibility. For example, if our square originally had possibilities of 1, 2, 5, and 6 but we learn that the number must be 6 then we would eliminate 1, 2, and 5 iteratively by calling `eliminate`.

`eliminate` does three things:

1. Removes the value from that specific square
2. Checks if the square only has one option left. If that's true then we have figured out what that square is so we then eliminate the remaining option from all the peers of the square.
3. Then, we check each unit to see if we have found any number that can only exist on one square. We'll want to call our `assign` method.

Make sure you don't provide too much guidance here. This is one of the more difficult parts of the project and a great opportunity to struggle and learn. Give them kind words of support to encourage them.

Homework

- Once you get constraint propagation written, write a method that prints each step as the constraints get updated. This will help you manually check your logic. You can check that it works correctly.

EXECUTING: HARD PUZZLE SOLVING

Concepts

- Depth-first search

Warm-up

Try to use constraint satisfaction to solve the following input:

`4....8.5.3.........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4......`

Notice that you don't get to a solution? That's concerning! How do we go about resolving this? There are lots of approaches, but we will just use a simple search algorithm.

Instructions

Idea

This is where we use *searching* for the answer.

We could systematically try all possibilities until we hit one that works. It's easy to write this code but we run into a problem! It could take too long to actually run. We can have trillions and trillions of games to check. How can we cope with that? There are two choices.

First, we could try a brute force approach. Suppose we have a very efficient program that takes only one instruction to evaluate a position. With the best available hardware, out program would take longer than the entire age of the universe to solve one puzzle. Not very feasible.

The second choice is to somehow process much more than one possibility per machine instruction. That seems impossible, but fortunately it is exactly what constraint propagation does for us. We don't have to try all the possibilities because as soon as we try one we immediately eliminate many other possibilities. We can go from trillions to only a few dozen possibilities to check.

How do we search? First, you check that you haven't found a solution or a contradiction. If you have a solution, give it to the user! If you have a contradiction, quit wasting your time down this part of the search. If neither of those things, find an unfilled square, consider the list of possible values, and try one. Then, keep searching from that position. If you make sure we haven't already found a solution or a contradiction, and if not, choose one unfilled square and consider all its possible values. This is a recursive search called a a depth-first search because we (recursively) consider all possibilities under our attempted values before we consider a different value.

It's your responsibility here to check their understanding of DFS. You can try drawing a state tree where as you go deeper you fill in a square in the puzzle. Ask them questions like how tall and wide (depth and branching factor) the tree must be. Explain how breadth first search would work on this problem and how it's not feasible because of the high branching factor resulting in a memory overflow. There is a link at the bottom to use to explain BFS and DFS.

So, which square do we try a value from first? Let's try the square that has the fewest remaining possibilities. If a square can only have a 1 or a 2, we're pretty certain about the number. But, if it could have a 1, 2, 3, 4, 5, 6, 7, or 8 the only thing we know is that it's not a 9. We're more likely to guess wrong in the latter case than the former. And then just guess one of the options for that square.

Obviously, we could do something more intelligent than this guess and check. But, we'll leave that as a challenge.

Challenges

- Think about how you could implement the *naked twins strategy*.

Homework

Write unit tests to check your algorithm. Also, try timing the execution speed.

Resources

- A comparison of depth first search and breadth first search

EVALUATING AND ENJOYING

We want students to develop a habit of trying to grow from their work. That means they must reflect on their success and failures in the project. Some of the questions they should strive to answer are boiler-plate and will be the same across all the projects. You should add some questions that are project specific and specific to things you noted in your student along the way.

Concepts

- Determine how the project will be disseminated.
- Reflect on the successes of the project.
- Learn from the challenges of the project.
- Identify what is left to do on the project.

Instructions

Questions

Answer the following questions:

1. What took the most time in the project?
2. What was the easiest for me in the project?
3. What do I wish I had known before I started the project?
4. Did I enjoy the project? Why or why not?
5. Did I pass all my tests for the project?
6. If I had more time, what might I add on the project?
7. Why did we use a dictionary to represent the grid instead of a two-dimensional array? How would it have been different with a two dimensional array.
8. Why use depth first search instead of breadth first search?

Homework

- Use your code to solve this Project Euler problem.
- Think about how much harder the puzzle gets as the grid gets bigger. Read the associated article titled Is Sudoku or Chess Harder?.

Resources

- The Mathematics of Sudoku

APPENDIX

	1	2	3	4	5	6	7	8	9
A	5	2	1	6	8	7	3	4	9
B	4	7	3	5	9	1	8	2	6
C	8	6	9	2	4	3	5	7	1
D	1	4	5	8	6	9	7	3	2
E	2	3	8	1	7	5	9	6	4
F	7	9	6	3	2	4	1	5	8
G	9	1	2	7	5	6	4	8	3
H	6	5	4	9	3	8	2	1	7
I	3	8	7	4	1	2	6	9	5

Figure 2: The solution for Figure 1.