2020

ACIT4420 Final Project

SUDUKO SOLVER

ØYSTEIN AUNE SVERRE – S326287

Content

Section 1 – Task

In this project we will be creating a sudoku solver. The program will receive a sudoku game and based on the sudoku, output a solution or information. To create this, we must first know the rules of sudoku. Afterwards we will present additional features needed in this project.

Sudoku rules

Sudoku is a game based on a simple set of rules. There is a grid system of cells that can contain a number from one through nine. The sudoku is solved when all the cells have a number in them and none of the number placements break the placing rules. The grid system is typically a board with the dimension 9x9, resulting in 81 cells. The divisions of this board are essential to understand the rules for placing numbers. The board is split into rows, columns, and regions.

The row is an entire line from the left side, through to the right side, resulting in nine cells. All these cells must contain all the numbers from one to nine, once. This applies to columns as well, only from top to bottom, also resulting in nine cells. The third rule is based on regions. Regions are 3x3 subsections of the board, each containing nine cells. These start in the upper left corner and combined they form the entire board. The third rule is that all numbers from one to nine must appear only once in the region. In the picture below you can see a highlighted row and column. Regions are differentiated with background colors of white and bright blue.

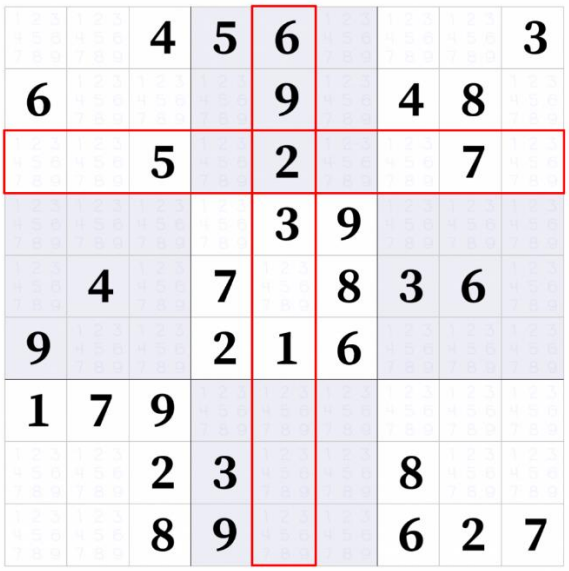

Figure 1 - Picture from https://sudokulinks.com/ showing what areas apply when placing a number in sudoku. Regions are highlighted by white and bright blue backgrounds. A row is shown horizontally using a red border and a column is displayed with a red border vertically.

Knight's move

The knights move is an additional rule that can be added to sudoku to create a bigger challenge. This sudoku solver must support this rule as an option to include when solving sudokus. Just like the previous rules, this rule creates another scenario to consider when placing a number into a cell. Including looking at rows, columns and regions, when placing a number one must consider numbers

that are reachable using the knight's move. This is a move from chess and involves moving two steps forward and one step sideways, creating an "L". If the number you are about to place is already reachable by the knight's move, then the placement is invalid.

King's move

The king's move is also an additional rule that needs support, and it is very similar to the knight's move. The main difference is the chess move that inspired it. A king can only move to cells adjacent and diagonal to itself. Meaning that when placing a number and respecting the king's move, the number must not be found in the 3x3 region around the selected cell.

Input

The program must support some inputs to have the required functionality. These inputs are as follows:

- A two-dimensional array, 9x9. This is the sudoku starting point.
- A true or false for using the king's move.
- A true or false for using the knight's move.

Output

Which features the program utilizes are based on the input provided. Information that can be given back to the user is:

- Is the incomplete sudoku provided solvable?
- Is the complete sudoku provided solved correctly?
- What is the solution to an incomplete sudoku?

Section 2 – Design

The plan is to create a script that uses brute force to test different sudoku solutions. When a sudoku is provided it has certain cells filled out, meaning we just need to test different numbers in the unplaced cells to find the correct solution. This will require a couple of key methods to work.

- Check the rules. Provide the method with coordinates and a number to see if it fits into the board without breaking the rules of sudoku.
- Go through board. Loop through the board and look for empty cells.
- Check solution. Go through placed numbers and verify their placement in relation to the others (could utilize the first method).

The board will be a two-dimensional array, originally containing 81 cells with question marks. They represent an unknown value. When a number is chosen for placement it replaces the question mark. When a sudoku is submitted to the program certain cells will have values already, creating a starting point where only the question mark cells need to be processed. Because the board is a two-dimensional array it can be easily processed using two for-loops within each other. The first for-loop goes through the y coordinate and the second loop goes through the x coordinate. The second for-loop is processed for every y coordinate, meaning the entire width of the board each time the first for-loop changes height. Resulting in 100% board coverage.

The plan is to go through each question mark, checking all possible numbers that could fit, from one to nine, and choosing one that fits at random. At some point a previous placement could break the sudoku, creating an unsolvable board. The board is at that point reset back to its original formation and the process is restarted. When the board is full and no numbers break any rules, that means the brute force has found a solution and the process is complete.

The main core of the program is the rule checker that has sudoku parameters built in. It checks that the numbers adhere to the rules of rows, column, regions and if enabled, knight's move and king's move. This method can be used by any method, responding only if a number is suitable to the chosen location.

This approach requires testing a lot of different sudoku combinations, but the high performance of modern computers makes it very doable.

Section 3 – The process

This section will go through the process of creating the sudoku solver. Not all code will be explained in detail, but the parts considered important to the understanding of the process are discussed. Referenced code can be found in the appendix.

Brute forcing

The first thing created was the board since it was necessary for testing methods. The original board composition was necessary to keep saved because we must restart the process from scratch when the solution breaks. This was done by copying the board for altering, but ended up causing problems during testing, even when the two-dimensional array was copied using the "copy()" method. This was first fixed when changed to a "deepcopy()" solution, ensuring the original was left untouched.

A while-loop that continuously deep-copied the original board and started a brute-forcing process was created. This while loop was only stopped by one of three factors, a solved sudoku, an unsolvable starting point or by running for too long. In the code below you can see the beginning of the while-loop. The brute-forcing is restarted on line six, sending the board, its dimensions, the running length and the use of knight's and king's move.

```
1.  while running and logic and runs < runs_max:
2.      board = copy.deepcopy(board_orig)
3.      print("")
4.      print("Bruting from scratch... {}/{}".format(runs, runs_max))
5.
6.      board = brute_check(board, 9, 9, 100000000, False, False)
```

Figure 2 – While-loop starting the brute-force from scratch. The "brute_check()" function is called on line six.

The "brute_check()" method is itself a while-loop, checking for question marks in the board. It is only stopped when the received run-length is achieved, the board is impossible to solve or the sudoku is solved. As can be seen in the code below, when a question mark is discovered, the sudoku is deemed unfinished and sent to "brute_random()" to place random, but rule-abiding numbers. If nothing can be placed it will return the board as impossible, and then return to the previous method above.

```
1.  while not finished:
2.          check = False
3.          for y in range(board_ver):
4.              for x in range(board_hor):
5.                  if board_input[y][x] == "?":
6.                      check = True
7.          if check:
8.              board_input, impossible = brute_random(board_input, board_ver, board_hor
    , impossible, check_knight, check_king)
9.          else:
10.             finished = True
```

Figure 3 - Another while-loop looking for question marks in the board. If one is found, the board is sent to "brute_random()" to find random, but fitting integers.

The “brute_random()” method also goes through the board. The intention is to find question marks to replace. When one is found the coordinates are sent to the “check_rules()” method with an array of possible numbers to test. The array contains the numbers one through nine in a random order. The random order is to ensure that the fitting number that is picked differs every time the brute force process is done. In the code below, the array of numbers is randomized on line two. Then, on line five, if some numbers work, the number is placed on the location previously containing a question mark.

```
1.  num = [1,2,3,4,5,6,7,8,9]
2.  random.shuffle(num)
3.  works, num_int = check_rules(board_input, x, y, num, check_knight, check_king)
4.  if works != False:
5.      board_input[y][x] = num_int
6.  else:
7.      impossible = True
8.      # print("{} at {}:{} didn't work".format(num_int, y, x)) # debug line
9.      return board_input, impossible
```

Figure 4 - The eligible numbers are defined on line one, shuffled on line two and unfitting candidates are removed in the “check_rules()” method.

Check the rules

As stated before, the method for checking numbers receives coordinates and an array of numbers to test. At first this array is processed using a for-loop, checking each number on the given coordinate. This number, at the given position, has three mandatory and two optional rules to test. If the number breaks any rule in the position it will be removed from the array. When all numbers in the array have been checked, the array containing valid numbers are returned.

Check row

Checking the row for identical numbers is simple. One for-loop running through the x-axis, but maintaining the y-axis, checks for the given number, only ignoring the original coordinate where the number will be placed if eligible. In the code snippet below, you can see how this is done and that “line” is changed to “False” on line four if the number is found on the line.

```
1.  #Check for n in row
2.  for j in range(9):
3.      if board_input[y][j] == num[n] and (j != x):
4.          line = False
5.          # print("{} found in y line j{}".format(num[n], j)) # debug line
```

Figure 5 - Checks for the number on the line, defining a Boolean on line four if the rule is broken.

Check column

Checking is column is identical to the row, but it checks the y instead of the x.

Check region

Checking the region is a bit complicated because we must define the region based on the coordinate to check. We can do this by selecting the four corners of the region built around the coordinate. When we have the four corners, we can go through the region using two for-loops. As we can see in the code below, we know that if y is two or less, the y for the start of the region is zero and the max is two. If y is five or less, the y for the start of the region is three and the end is five. If none of this applies, the last y possible for the regions is starting at six and ending at eight. The two defined parameters, “y_region_min” and “y_region_max”, including two values for x min and max, are then used to define the region to check for the number.

```
1.  if y <= 2:
2.      y_region_min = 0
3.      y_region_max = 2
4.  elif y <= 5:
5.      y_region_min = 3
6.      y_region_max = 5
7.  else:
8.      y_region_min = 6
9.      y_region_max = 8
10.
11. #Check for num in block
12. for j in range(y_region_min, y_region_max+1):
13.     for i in range(x_region_min, x_region_max+1):
```

Figure 6 - The two bottom corners are defined using if/else-statements on line one to nine, then two for loops checks the region.

Check knight’s move

For checking the knight’s move, we do a similar process to checking the region, but change the size of the region so it contains all possible knight movements. It was important to restrict the region being checked if the coordinates were close to the border, meaning it would big when centered, but cut when touching the edge of the board. We then do a complicated if-line inside that loop that checks in four directions and “forks” into two paths from that original direction. In the example on line two below, you can see it checks y+2 and then requires x to be either +1 or -1. If the number being checked is found on this coordinate it is breaking the knight’s move constraint.

```
1.  if board[j][i] == num and not (j == y and i == x) and \
2.                          ((j == y+2 and (i == x+1 or i == x-1)) or
3.                           (j == y-2 and (i == x+1 or i == x-1)) or
4.                           (i == x+2 and (j == y+1 or j == y-1)) or
5.                           (i == x-2 and (j == y+1 or j == y-1))):
```

Figure 7 - If statement that checks for knight's move in relation to the original coordinate.

Check king's move

Checking for the king's move is simple compared to the knight's move. The region is defined such as with the knight's move, but it is smaller and no "forks" are required. It is still important to cut the region if coordinates to check is touching the border. This region is then checked, only ignoring the original coordinates.

Complications

When everything was put together it was functional, but not efficient. Providing a simple sudoku for solving with relatively few unknowns gave a perfect solution after half an hour. Because the solution was randomly generated and had to get the right combination, getting the correct solution is a statistical inevitability, but takes an increasing amount of time for every unknown value. Even though this approach was working, it was not useful as a sudoku solver. There had to be a more efficient way to solve the sudoku.

Its main issue is the fact that the program keeps retrying combinations it has already done when the sudoku breaks. The main culprit that was wrongfully placed is hard to track down without creating an extremely complicated method. A failed attempt at a cleverer sudoku solver was created. The goal was to try to be able to backtrack, but it was too difficult to execute. Following one of the tips on the project description, another solution was created using recursion instead of brute force. The rule-checking method could still be used, but all the redundant for/while-loops could be removed. All methods were also changed to use global values and the rule-checking method now returned true/false statements. The code was rewritten to be more efficient, but the main concepts were kept.

Recursion

The new recursive method is simply written compared to earlier code but can be hard to comprehend. The recursive method goes through the board looking for unplaced cells, just once. Then it enters a for-loop going through all possible candidates (one to nine). This was done in the earlier version, but in the rule-checking method instead. If a candidate from the for-loop fits into the position of the available cell, it will be placed there, and the new version of the board is then used when the recursive method calls on itself. After the method is done calling itself, the number that was placed into the board is removed, reverting the change. When the for-loop is done checking all candidates, it will execute a return command. This returns to the previous call of the method.

All this can be hard to interpret, but the main point is that only a valid, filled sudoku will make it to the outside of the for-loops because it will not find any unfilled cells, therefore it will not recursively call itself again. The recursive method goes through all possible, valid ways to fill the board, which is slow and expansive, but not random. An interpretation is not done twice. This recursive method goes through all possibilities like a tree with branches, turning back, switching branch, whenever the board runs out of options. You can see on line 11 below that when it reaches the end of the for-loop it sends the current board to the "check_unplaced()" method, returning the amount of question mark-filled cells. If valid, the board is then deep-copied into the variable called "board_solved" on line 12. Just one solution is necessary, but the number of solutions found is summed together on line thirteen.

```
1.  for y in range(board_hor):
2.      for x in range(board_ver):
3.          if board[y][x] == "?":
4.              for n in range(1, 10):
5.                  if check_rules(y, x, n):
6.                      board[y][x] = n
7.                      recursions = recursions + 1
8.                      check_board_recursion()
9.                      board[y][x] = "?"
10.             return
11. if check_unplaced(board) == 0:
12.     board_solved = copy.deepcopy(board)
13.     solutions = solutions + 1
14.     if logging:
15.         print("Found one solution")
16. return
```

Figure 8 – The recursive method, going through all missing values in the board. Every candidate is tested, one by one. When a solution makes it to the end it will be saved, and the number of solutions increases.

The method itself does not know what it is looking for and will keep calling itself until all possible combinations are processed, but only a full, valid solution is saved as a separate board. That saved board is the solution we were looking for. It is only saved because it did not execute the for-loop on line four and passed the test on line 11, having zero empty cells.

Interface

Creating an easy way to input an array into the program was simple. Using the “input()” method allows the user write in a value. The board is looped, asking for a value for each cell. The previous board placements are printed for each value, providing context. If a number is provided, it is placed at the current location. If an “x” is provided the program is exited. If any other input is provided it will be interpreted as an unfilled cell, resulting in a question mark. This allows for easy, efficient filling of the board, using just the keyboard. In the code below you can see the if/else-lines choosing how to interpret the input. “val” is the input given by the user.

```
1.  if val == "x":
2.      exit();
3.  elif val.isnumeric():
4.      board_orig[y][x] = int(val)
5.  else:
6.      board_orig[y][x] = "?"
```

Figure 9 - The if/else statements choosing between exiting the program, placing a numeric value, or creating an unfilled cell.

Combining the methods

When all the methods were functional, they needed to be tied together. If a full sudoku is provided, the program needs to check if the sudoku break any rules. If a sudoku is provided with unfilled cells it will attempt to solve it, then output the result. This is all done using if/else-statements. You can see below that if the board is not done (line one), and breaks no rules (line two), it will try to solve it (line seven).

```
1.  if check_unplaced(board) > 0:
2.      if not check_board_logic():
3.          print_board(board)
4.          print("Unsolvable")
5.      else:
6.          print("Given sudoku breaks no rules. Solving sudoku...")
7.          check_board_recursion()
```

Figure 10 - If/else-statements determining what methods should be called. The solving method is executed on line seven.

Settings

Creating an interactable program requires some values to be declared, and testing does not require the same behavior as regular use. Therefore, the beginning of the program has multiple values declared and this includes two settings. Logging and statistics. The logging is useful for debugging and makes the program print as it attempts to solve the sudoku. The statistics are useful if someone wants to view more information from the solving process (number of recursions, timespan, number of solutions). By default, “logging” is disabled, and “stats” are enabled. An individual using the sudoku solver will probably be more interested in the solution and its details rather than the process behind it.

Section 4 – Results

The final tactic attempted in section three was of much greater value than the brute force approach. Even though both methods take an increasing amount of time to solve based on how many unknown cells there are, the recursion approach of not retrying earlier combinations greatly benefits the program. When the solution is created randomly using brute force, and there are many cells to fill, that results in a lower probability of guessing the correct solution. That increases the waiting time noticeably. The recursion tactic is much more elegant.

Testing

When solving a simple sudoku using recursion it outputs the result in 14 seconds, as you can see in the example below. The original sudoku was generated on sudoku-solutions.com (Sudoku Solutions, 2020) using a "simple sample" and has 53 unknown cells. This was then solved and given back to the website, who approved the solution. Most of the time was spent writing the sudoku into the two different interfaces, not waiting the 14 seconds for the solution.

```
1.  [
2.  ["?","?",6,"?","?","?","?","?","?"],
3.  [7,"?","?",9,"?",3,"?",1,"?"],
4.  [9,3,"?","?",2,6,"?","?","?"],
5.  ["?","?","?","?","?",9,"?",4,8],
6.  ["?","?",8,"?",6,2,7,"?",1],
7.  ["?","?","?","?","?",8,"?",2,5],
8.  [4,7,"?","?",9,"?","?","?","?"],
9.  [5,"?","?",6,"?",1,"?",3,"?"],
10. ["?","?",9,"?","?","?","?","?","?"]
11. ]
12.
13. _____________________________________
14. | 8 | 2 | 6 | 1 | 4 | 7 | 9 | 5 | 3 |
15. _____________________________________
16. | 7 | 4 | 5 | 9 | 8 | 3 | 2 | 1 | 6 |
17. _____________________________________
18. | 9 | 3 | 1 | 5 | 2 | 6 | 8 | 7 | 4 |
19. _____________________________________
20. | 2 | 1 | 7 | 3 | 5 | 9 | 6 | 4 | 8 |
21. _____________________________________
22. | 3 | 5 | 8 | 4 | 6 | 2 | 7 | 9 | 1 |
23. _____________________________________
24. | 6 | 9 | 4 | 7 | 1 | 8 | 3 | 2 | 5 |
25. _____________________________________
26. | 4 | 7 | 3 | 8 | 9 | 5 | 1 | 6 | 2 |
27. _____________________________________
28. | 5 | 8 | 2 | 6 | 7 | 1 | 4 | 3 | 9 |
29. _____________________________________
30. | 1 | 6 | 9 | 2 | 3 | 4 | 5 | 8 | 7 |
31. _____________________________________
32. Sudoku solution
33. Time: 14.637967109680176 sec
34. Recursions: 175815
35. Solutions found: 1
36. King's move: False
37. Knight's move: False
```

Figure 11 - Original sudoku from sudoku-solutions.com (Sudoku Solutions, 2020) and the solved version. Line 33-37 shows information about the solution found, enabled by the "stats" setting. The solved version was accepted by the website.

More regular sudokus were wanted for testing though, so a sudoku book was purchased. It included 200 “evil” sudokus, with solutions in the back. This resulted in a big amount of efficient testing. The sudokus in the book get harder the further you read, and that had a correlation with the solving time when using the program. All tested sudokus ended up with successful results in the end. In the photo below you can see the book (Mer Sudoku Fra Djevelen, 2020).

Figure 12 - A book featuring 200 sudoku puzzles (Mer Sudoku Fra Djevelen, 2020).

In another test we used an example from a user on stackexchange.com (user27014, 2018) who is trying to prove you can create sudokus with multiple solutions. In his example just four cells are unknown, resulting in our sudoku solver solving it in record time. In just 0.002 seconds, eight recursions are executed, resulting in two different solutions, even though just the latest one is printed. In the statistics section, on line 34, we can see the number of solutions found is two, proving the sudoku has multiple solutions.

```
1.  [
2.  [2,9,5,7,4,3,8,6,1],
3.  [4,3,1,8,6,5,9,"?","?"],
4.  [8,7,6,1,9,2,5,4,3],
5.  [3,8,7,4,5,9,2,1,6],
6.  [6,1,2,3,8,7,4,9,5],
7.  [5,4,9,2,1,6,7,3,8],
8.  [7,6,3,5,2,4,1,8,9],
9.  [9,2,8,6,7,1,3,5,4],
10. [1,5,4,9,3,8,6,"?","?"]
11. ]
```

```
 _____________________________________
| 2 | 9 | 5 | 7 | 4 | 3 | 8 | 6 | 1 |
 _____________________________________
| 4 | 3 | 1 | 8 | 6 | 5 | 9 | 7 | 2 |
 _____________________________________
| 8 | 7 | 6 | 1 | 9 | 2 | 5 | 4 | 3 |
 _____________________________________
| 3 | 8 | 7 | 4 | 5 | 9 | 2 | 1 | 6 |
 _____________________________________
| 6 | 1 | 2 | 3 | 8 | 7 | 4 | 9 | 5 |
 _____________________________________
| 5 | 4 | 9 | 2 | 1 | 6 | 7 | 3 | 8 |
 _____________________________________
| 7 | 6 | 3 | 5 | 2 | 4 | 1 | 8 | 9 |
 _____________________________________
| 9 | 2 | 8 | 6 | 7 | 1 | 3 | 5 | 4 |
 _____________________________________
| 1 | 5 | 4 | 9 | 3 | 8 | 6 | 2 | 7 |
 _____________________________________
Sudoku solution
Time: 0.0019559860229492188 sec
Recursions: 8
Solutions found: 2
King's move: False
Knight's move: False
```

Figure 13 - Another example of the program solving a sudoku. This time proving that the example from stackexchange.com (user27014, 2018) can have two solutions on line 34.

Changes based on testing

It was realized at this point that the extra time spent waiting for the program to find all possible solutions, while interesting, was a waste of time if just one solution was wanted. Another input was created at the beginning of the program's execution, asking for the number of desired solutions. The recursion method will now add solutions to an array instead of saving the latest one. If the number of wanted solutions are found, the recursion will immediately return, ending the search. That means that an extremely complicated sudoku, that takes a long time to process, can potentially finish very quickly, if you want for example one solution. In the code below, on line six, you can see the simple check in the recursive method that returns if the number of solutions wanted is found. All found solutions are then printed later when the program is finished.

```
1.  for n in range(1, 10):
2.      if check_rules(y, x, n):
3.          board[y][x] = n
4.          recursions = recursions + 1
5.          check_board_recursion()
6.          if solutions_wanted <= solutions and solutions_wanted != 0:
7.              return
8.          board[y][x] = "?"
9.  return
```

Figure 14 - New check, on line six, that returns if the desired number of solutions is achieved.

Testing the additional rules

Testing the knight's move rule was successful when providing examples with clear violations but testing an example from the internet, with a correct solution available, was also desirable. A simplified version of an example from funwithpuzzles.com (Kumar, 2015) provided two different solutions with the knight's move enabled. This was an excellent way to display the new printing of all found solutions. On line 55 you can see wanted solutions is zero, resulting in finding all solutions. On line 56 and 58, the number of found solutions are two, and knight's move is enabled. The first solution, beginning on line 14, is the solution example provided by the website.

```
1.  [
2.  [9,3,1,"?","?","?","?","?","?"],
3.  [7,8,5,"?","?","?","?","?","?"],
4.  [6,4,2,"?","?","?",5,"?","?"],
5.  [4,2,3,"?","?","?","?","?",6],
6.  [1,5,9,"?","?","?","?","?","?"],
7.  [8,7,6,"?","?","?","?","?","?"],
8.  [3,6,7,"?","?","?",9,2,4],
9.  [5,9,4,"?","?","?",8,1,3],
10. [2,1,8,"?","?","?",7,6,5]
11. ]
12.
13. Solution 1:
14. _____________________________________
15. | 9 | 3 | 1 | 5 | 4 | 2 | 6 | 7 | 8 |
16. _____________________________________
17. | 7 | 8 | 5 | 6 | 3 | 9 | 2 | 4 | 1 |
18. _____________________________________
19. | 6 | 4 | 2 | 7 | 8 | 1 | 5 | 3 | 9 |
20. _____________________________________
21. | 4 | 2 | 3 | 8 | 7 | 5 | 1 | 9 | 6 |
22. _____________________________________
23. | 1 | 5 | 9 | 3 | 2 | 6 | 4 | 8 | 7 |
24. _____________________________________
25. | 8 | 7 | 6 | 9 | 1 | 4 | 3 | 5 | 2 |
26. _____________________________________
27. | 3 | 6 | 7 | 1 | 5 | 8 | 9 | 2 | 4 |
28. _____________________________________
29. | 5 | 9 | 4 | 2 | 6 | 7 | 8 | 1 | 3 |
30. _____________________________________
31. | 2 | 1 | 8 | 4 | 9 | 3 | 7 | 6 | 5 |
32. _____________________________________
33. Solution 2:
34. _____________________________________
35. | 9 | 3 | 1 | 5 | 4 | 2 | 6 | 8 | 7 |
36. _____________________________________
37. | 7 | 8 | 5 | 6 | 3 | 9 | 2 | 4 | 1 |
38. _____________________________________
39. | 6 | 4 | 2 | 7 | 8 | 1 | 5 | 3 | 9 |
40. _____________________________________
41. | 4 | 2 | 3 | 8 | 7 | 5 | 1 | 9 | 6 |
42. _____________________________________
43. | 1 | 5 | 9 | 3 | 2 | 6 | 4 | 7 | 8 |
44. _____________________________________
45. | 8 | 7 | 6 | 9 | 1 | 4 | 3 | 5 | 2 |
46. _____________________________________
47. | 3 | 6 | 7 | 1 | 5 | 8 | 9 | 2 | 4 |
48. _____________________________________
49. | 5 | 9 | 4 | 2 | 6 | 7 | 8 | 1 | 3 |
50. _____________________________________
51. | 2 | 1 | 8 | 4 | 9 | 3 | 7 | 6 | 5 |
```

```
52. _____________________________________
53. Time: 0.6679942607879639 sec
54. Recursions: 5456
55. Solutions wanted: 0
56. Solutions found: 2
57. King's move: False
58. Knight's move: True
```

Figure 15 - Two solutions printed when searching using the knight's move (see line 58).

Similarly, the king's move was successful in testing, but real-world examples were desired. An example was found on cross-plus-a.com (Cross+A, 2020) using the king's move. The result given from the sudoku solver was identical to the solution on the website and on line 37 you can see that the king's move is enabled.

```
1.  [
2.  ["?","?","?","?","?",5,"?",6,"?"],
3.  ["?","?","?",4,2,"?","?","?",1],
4.  ["?",7,"?","?",6,"?","?",3,"?"],
5.  ["?","?","?","?",3,"?","?","?",2],
6.  ["?",2,"?","?","?","?","?",9,"?"],
7.  [6,"?","?","?",7,"?","?","?","?"],
8.  ["?",4,"?","?",1,"?","?",8,"?"],
9.  [8,"?","?","?",4,7,"?","?","?"],
10. ["?",1,"?",9,"?","?","?","?","?"]
11. ]
12.
13. Solution 1:
14. _____________________________________
15. | 4 | 3 | 1 | 7 | 9 | 5 | 2 | 6 | 8 |
16. _____________________________________
17. | 9 | 6 | 8 | 4 | 2 | 3 | 7 | 5 | 1 |
18. _____________________________________
19. | 5 | 7 | 2 | 1 | 6 | 8 | 9 | 3 | 4 |
20. _____________________________________
21. | 1 | 9 | 5 | 8 | 3 | 4 | 6 | 7 | 2 |
22. _____________________________________
23. | 7 | 2 | 4 | 6 | 5 | 1 | 8 | 9 | 3 |
24. _____________________________________
25. | 6 | 8 | 3 | 2 | 7 | 9 | 4 | 1 | 5 |
26. _____________________________________
27. | 2 | 4 | 9 | 5 | 1 | 6 | 3 | 8 | 7 |
28. _____________________________________
29. | 8 | 5 | 6 | 3 | 4 | 7 | 1 | 2 | 9 |
30. _____________________________________
31. | 3 | 1 | 7 | 9 | 8 | 2 | 5 | 4 | 6 |
32. _____________________________________
33. Time: 0.0979762077331543 sec
34. Recursions: 5678
35. Solutions wanted: 0
36. Solutions found: 1
37. King's move: True
38. Knight's move: False
```

Figure 16 - Results from testing sudoku from cross-plus-a.com (Cross+A, 2020) using the king's move.

Conclusion

Even though the approach was changed mid-way, the result is an elegant, efficient solution with a good amount of use cases. In most cases the solver would be used to just find solutions to random sudokus, but it is also useful for validating sudoku solutions and mapping out the number of possible solutions and then printing them. In one experiment the solver was given an empty sudoku and theoretically it should then output the number of all possible sudoku combinations and print them. The program was left running overnight but crashed because the computer ran out of memory. This program could, if the system had enough capacity, map out all possible sudoku combinations, with or without the king and knight's move.

The addition of asking for the number of wanted solutions was added late in the process but is in retrospect incredibly useful with this approach of solving the sudokus. When searching for solutions all possible combinations will be attempted, resulting in a ridiculous amount of processing time if the given sudoku has few already determined cells. In this scenario the program could find the solution in 10 seconds or 10 minutes, based on when it attempts that successful combination. We do know is that if you give a sudoku with that many unknowns, there is a high probability of multiple solutions, but you would probably just want one. Therefore, asking for the number of wanted solutions can significantly cut down on waiting time.

Requiring the sudoku to solve using the knight and king's move constraint was less of an obstacle than expected because the tactic of having a rule-checking method. The world of sudoku solving has many different rules and scenarios that are implemented for a bigger challenge and with this approach we could add even more optional rules to this program. The hardest thing to code was the checking of the knight's move, but the approach in section three of "forking" out from the original coordinate was very effective, but not complicated to write.

The interface for using the program is simple, but effective. At least when compared to online sudoku solvers. The project did not require an interface, just inputs, so this solution is more than sufficient. An additional useful feature would be backtracking when inputting numbers into the board, in case of a mistake. If desired, one could comment out the "get_board()" method and input an array into the program code manually.

Looking at the project requirements, this solution fulfills all the necessary features, and then some. Creating the solver was a lot of fun and resulted in a bigger interest in sudoku and more on-hand experience with recursion.

Appendix

References

(2020). In M. Boström, *Mer Sudoku Fra Djevelen.* Lille Måne.

Cross+A. (2020). *Anti-King Sudoku*. Retrieved from Cross+A: http://www.cross-plus-a.com/sudoku.htm#Anti-King%20Sudoku

Kumar, R. (2015). *Anti Knight Search 9 Sudoku (Daily Sudoku League #121)*. Retrieved from Fun With Puzzles: https://www.funwithpuzzles.com/2015/07/anti-knight-search-9-sudoku.html

Sudoku Solutions. (2020). *Sudoku Solver*. Retrieved from Sudoku Solutions: https://www.sudoku-solutions.com/

user27014. (2018). *Examples of sudokus with two solutions [duplicate]*. Retrieved from Puzzling: https://puzzling.stackexchange.com/questions/67789/examples-of-sudokus-with-two-solutions

Code

Brute force-relevant code

```
def check_done(board_input, board_ver, board_hor):
    finished = True
    for y in range(board_ver):
        for x in range(board_hor):
            if board_input[y][x] == "?":
                finished = False
                break

    return finished


def brute_check(board_input, board_ver, board_hor, tries, check_knight, check_king):

    finished = False
    impossible = False
    while not finished:
        check = False
        for y in range(board_ver):
            for x in range(board_hor):
                if board_input[y][x] == "?":
                    check = True
        if check:
            board_input, impossible = brute_random(board_input, board_ver, board_hor, impossible, check_knight, check_king)
        else:
            finished = True

        tries -= 1

        if tries < 1:
            break

        if impossible:
            # print("Board impossible...") # debug line
            return board_input

    return board_input

```

```
37.
38. def brute_random(board_input, board_ver, board_hor, impossible, check_knight, check_
    king):
39.     for y in range(board_ver):
40.         for x in range(board_hor):
41.             if board_input[y][x] == "?":
42.                 print("")
43.                 num = [1,2,3,4,5,6,7,8,9]
44.                 random.shuffle(num)
45.                 works, num_int = check_rules(board_input, x, y, num, check_knight, c
    heck_king)
46.                 if works != False:
47.                     board_input[y][x] = num_int
48.                 else:
49.                     impossible = True
50.                     # print("{} at {}:{} didn't work".format(num_int, y, x)) # debug
     line
51.                     return board_input, impossible
52.
53.     return board_input, impossible
54.
55.
56. runs = 0
57. lowest = 1000;
58. running = True
59. start_time = time.time()
60. logic = True;
61.
62. size_h = 9
63. size_v = 9
64. board_orig = [
65.         [1,"?",4,"?","?","?","?",5,"?"],
66.         [6,"?",5,"?","?",9,"?","?",2],
67.         ["?","?","?","?","?","?","?","?",9,8],
68.         [8,"?","?",6,"?","?",9,"?","?"],
69.         [5,2,9,"?","?","?","?","?","?"],
70.         ["?","?","?",8,"?",7,"?",1,"?"],
71.         ["?",4,7,"?",6,"?","?","?",1],
72.         ["?","?","?","?",5,"?",3,8,"?"],
73.         [2,"?","?",7,1,"?","?","?","?"]
74.         ]
75.
76.
77. logic = check_board_logic(board_orig, 9, 9, False, False)
78. runs_max = 10000000
79. while running and logic and runs < runs_max:
80.     board = copy.deepcopy(board_orig)
81.     print("")
82.     print("Bruting from scratch... {}/{}".format(runs, runs_max))
83.
84.     board = brute_check(board, 9, 9, 100000000, False, False)
85.
86.     # print("NEW")
87.     # print_board(board, 9, 9) # Debug line
88.
89.     unplaced = check_unplaced(board, 9, 9)
90.     print("Unplaced figures: {}.0".format(unplaced)) # debug line
91.     if unplaced < lowest:
92.         lowest = unplaced
93.
94.     running = not check_done(board, 9, 9)
95.     runs += 1
96.     print("Lowest unplaced figures: {}".format(lowest)) # debug line
97.
98.
99. if not running:
```

```
            print_board(board, 9, 9)
            print("This solution works")
        elif not logic:
            print("The sudoku is unsolvable...")
        else:
            print("I gave up...")

        s = time.time() - start_time
        m, s = divmod(s, 60)
        print("--- {} minutes {} seconds ---".format(m, s))
```

Final solution

```
import time as time
import copy as copy
import sys as sys


# Returns the number of cells that have unknown values.
def check_unplaced(board_input):
    amount = 0
    for y in range(board_ver):
        for x in range(board_hor):
            if board_input[y][x] == "?":
                amount += 1

    return amount


# Checks placed numbers in relation to other placed numbers, returning false if the board is invalid.
def check_board_logic():
    for y in range(board_ver):
        for x in range(board_hor):
            if board[y][x] != "?":
                works = check_rules(y, x, board[y][x])
                if not works:
                    return False

    return True


# A recursive method, calling itself repeatedly, saving valid solutions to an array.
def check_board_recursion():
    global recursions
    global board
    global board_solved
    global solutions
    global logging
    global solutions_wanted
    global board_solutions

    if solutions_wanted <= solutions and solutions_wanted != 0:
        return

    for y in range(board_hor):
        for x in range(board_ver):
            if board[y][x] == "?":
                for n in range(1, 10):
                    if check_rules(y, x, n):
                        board[y][x] = n
                        recursions = recursions + 1
```

```
                        check_board_recursion()
                        if solutions_wanted <= solutions and solutions_wanted != 0:
                            return
                        board[y][x] = "?"
                return

    if check_unplaced(board) == 0:
        board_solved = copy.deepcopy(board)
        board_solutions.append(board_solved)
        solutions = solutions + 1
        if logging:
            print("Found one solution")
    return


# The rule checking method. Based on the settings it can check if a number fits
# into a board based on row, column, region, knight's move and king's move.
def check_rules(y, x, num):
    global logging
    global king_check
    global knight_check

    if x <= 2:
        x_reg_min = 0
        x_reg_max = 2
    elif x <= 5:
        x_reg_min = 3
        x_reg_max = 5
    else:
        x_reg_min = 6
        x_reg_max = 8

    if y <= 2:
        y_reg_min = 0
        y_reg_max = 2
    elif y <= 5:
        y_reg_min = 3
        y_reg_max = 5
    else:
        y_reg_min = 6
        y_reg_max = 8

    #Check for num in region
    for j in range(y_reg_min, y_reg_max+1):
        for i in range(x_reg_min, x_reg_max+1):
            if board[j][i] == num and not (j == y and i == x):
                if logging:
                    print("Can't place {} at y: {}, x: {} because of region".format(num, y, x)) # debug line
                return False

    #Check for num in row
            for j in range(9):
                if board[y][j] == num and (j != x):
                    if logging:
                        print("Can't place {} at y: {}, x: {} because of row".format(num, y, x)) # debug line
                    return False

            #Check for num in column
            for j in range(9):
                if board[j][x] == num and (j != y):
                    if logging:
                        print("Can't place {} at y: {}, x: {} because of column".format(num, y, x)) # debug line
```

```
111.                     return False
112.
113.             if knight_check:
114.                 if x > 1:
115.                     x_left = x-2
116.                 elif x > 0:
117.                     x_left = x-1
118.                 else:
119.                     x_left = x
120.
121.                 if x < 7:
122.                     x_right = x+2
123.                 elif x < 8:
124.                     x_right = x+1
125.                 else:
126.                     x_right = x
127.
128.                 if y > 1:
129.                     y_left = y-2
130.                 elif y > 0:
131.                     y_left = y-1
132.                 else:
133.                     y_left = y
134.
135.                 if y < 7:
136.                     y_right = y+2
137.                 elif y < 8:
138.                     y_right = y+1
139.                 else:
140.                     y_right = y
141.
142.                 #Check for num in knight's move
143.                 for j in range(y_left, y_right+1):
144.                     for i in range(x_left, x_right+1):
145.                         if board[j][i] == num and not (j == y and i == x) and \
146.                                 ((j == y+2 and (i == x+1 or i == x-1)) or
147.                                  (j == y-2 and (i == x+1 or i == x-1)) or
148.                                  (i == x+2 and (j == y+1 or j == y-1)) or
149.                                  (i == x-2 and (j == y+1 or j == y-1))):
150.                             if logging:
151.                                 print("Can't place {} at y: {}, x: {} because of knig
    ht's move".format(num, y, x)) # debug line
152.                             return False
153.
154.             if king_check:
155.                 if x > 0:
156.                     x_king_min = x-1
157.                 else:
158.                     x_king_min = 0
159.
160.                 if x < 8:
161.                     x_king_max = x+1
162.                 else:
163.                     x_king_max = 8
164.
165.                 if y > 0:
166.                     y_king_min = y-1
167.                 else:
168.                     y_king_min = 0
169.
170.                 if y < 8:
171.                     y_king_max = y+1
172.                 else:
173.                     y_king_max = 8
174.
175.                 #Check for num in kingsmove
```

```
176.                 for j in range(y_king_min, y_king_max+1):
177.                     for i in range(x_king_min, x_king_max+1):
178.                         if board[j][i] == num and not (j == y and i == x):
179.                             if logging:
180.                                 print("Can't place {} at y: {}, x: {} because of king
    's move.".format(num, y, x)) # debug line
181.                             return False
182.
183.             return True
184.
185.
186.         # Prints the given board with spacing and lines to represent a sudoku board.

187.         def print_board(board_input):
188.             print("")
189.             for y in range(board_ver):
190.                 print("_____________________________________")
191.                 for x in range(board_hor):
192.                     print("| {} ".format(board_input[y][x]), end= "")
193.                 print("|")
194.             print("_____________________________________")
195.
196.
197.         # Get the board from the user using input().
198.         def get_board(board_input):
199.             global board_ver
200.             global board_hor
201.
202.             for y in range(board_ver):
203.                 for x in range(board_hor):
204.                     current = False
205.                     for j in range(board_ver):
206.                         if current:
207.                             break
208.                         print("_____________________________________")
209.                         for i in range(board_hor):
210.                             if j == y and i == x:
211.                                 current = True
212.                                 break
213.                             print("| {} ".format(board_input[j][i]), end= "")
214.                         print("|")
215.                     print("_____________________________________")
216.                     val = input("Enter the next value. Any number will be placed, 'x'
    exits the program, anything else will become an unknown value ('?')")
217.
218.                     if val == "x":
219.                         exit();
220.                     elif val.isnumeric():
221.                         board_input[y][x] = int(val)
222.                     else:
223.                         board_input[y][x] = "?"
224.             return board_input
225.
226.
227.         # The original board array
228.         board_orig = [
229.             ["?","?","?","?","?","?","?","?","?"],
230.             ["?","?","?","?","?","?","?","?","?"],
231.             ["?","?","?","?","?","?","?","?","?"],
232.             ["?","?","?","?","?","?","?","?","?"],
233.             ["?","?","?","?","?","?","?","?","?"],
234.             ["?","?","?","?","?","?","?","?","?"],
235.             ["?","?","?","?","?","?","?","?","?"],
236.             ["?","?","?","?","?","?","?","?","?"],
237.             ["?","?","?","?","?","?","?","?","?"]
238.         ]
```

```
239.
240.       # Sudoku settings. Three settings are retrieved through input().
241.       king = input("Using the king's move rule? y/n")
242.       if king == "y":
243.           king_check = True
244.       else:
245.           king_check = False
246.
247.       knight = input("Using the knight's move rule? y/n")
248.       if knight == "y":
249.           knight_check = True
250.       else:
251.           knight_check = False
252.
253.       wanted_solutions_string = input("How many solutions do you want if your sudok
    u is not complete? Any input but a number above 0 will provide all possible solution
    s.")
254.       if wanted_solutions_string.isnumeric() and int(wanted_solutions_string) > 0:

255.           solutions_wanted = int(wanted_solutions_string)
256.       else:
257.           solutions_wanted = 0
258.
259.       board_hor = 9
260.       board_ver = 9
261.       logging = False
262.       stats = True
263.
264.
265.       # Sudoku stats. Printed at the end if "stats" is enabled.
266.       recursions = 0
267.       start_time = time.time()
268.       stop_time = time.time()
269.       solutions = 0
270.
271.       # Sudoku logical varibales. These need to be declared for later use. The get_
    board() method is called here.
272.       sys.setrecursionlimit(100000)
273.       print("Please provide a sudoku:")
274.       board_orig = get_board(board_orig)
275.       board = copy.deepcopy(board_orig)
276.       board_solved = copy.deepcopy(board_orig)
277.       board_solutions = []
278.
279.       # Start the time for tracking recursion time.
280.       start_time = time.time()
281.
282.       # The main execution of methods. First checks if the board is empty, then out
    puts results based on that.
283.       if check_unplaced(board) > 0:
284.           if not check_board_logic():
285.               print_board(board)
286.               print("Unsolvable")
287.           else:
288.               print("Given sudoku break no rules. Solving sudoku...")
289.               check_board_recursion()
290.               if solutions < 1:
291.                   print("Found no solutions...")
292.               else:
293.                   for i in range(len(board_solutions)):
294.                       print("")
295.                       print("Solution {}:".format(i+1))
296.                       print_board(board_solutions[i])
297.       else:
298.           if check_board_logic():
299.               print_board(board)
```

```
300.                 print("Solved correctly")
301.             else:
302.                 print_board(board)
303.                 print("Solved incorrectly")
304.
305.         # Stop tracking time.
306.         stop_time = time.time()
307.
308.         # If stats are enabled these are printed.
309.         if stats:
310.             print("")
311.             s = stop_time - start_time
312.             m, s = divmod(s, 60)
313.             print("Time: {} sec".format(s))
314.             print("Recursions: {}".format(recursions))
315.             print("Solutions wanted: {}".format(solutions_wanted))
316.             print("Solutions found: {}".format(solutions))
317.             print("King's move: {}".format(king_check))
318.             print("Knight's move: {}".format(knight_check))
```