Assignment 2: Sudoku Solver
Project Report

Andrea Minichova - s1021688

November 30, 2021

Contents

1 Introduction

In this report, we will discuss the implementation of a sudoku solver using constraint satisfaction. In section 2, we describe sudokus, constraint satisfaction problems, the AC-3 algorithm. We relate these concepts to sudokus. In Section 3, we discuss our concrete implementation of the AC-3 solver and different heuristics. In section 4, we show results of several tests and reason about complexity while comparing different heuristics. Lastly, we conclude the report by stating other possible improvements to the implementation.

2 Project description

2.1 Sudoku

Sudoku is a 9 by 9 grid, containing smaller 3 by 3 sub-squares. Each cell can have a value between 1 and 9. In the initial configuration, some cells are set and the goal is to fill in the remaining cells while conserving the constraints. The constrains are following: every value needs to be unique in its corresponding row, column and sub-square. Below is an example of an initial sudoku and the same sudoku after it is solved.

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

(a) Initial sudoku 1

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

(b) Solved sudoku 2

Figure 1: Sudoku example from [1]

2.2 Constraint Satisfaction Problems

As defined in [4], constrain satisfaction problems (CSPs) are mathematical questions whose objects need in a problem must satisfy given constraints over variables, which is reached through different constraint-satisfaction methods.

Formally, a CSP is defined as a triple $\langle X, D, C \rangle$ where:

- $X = \{x_1, x_2, ..., x_n\}$ is a set of n variables

- $D = \{d_1, d_2, ..., d_n\}$ is a set of corresponding domains
- $C = \{c_1, c_2, ..., c_n\}$ is a set of constraints

In the case of sudoku, there are in total 81 variables (cells), a domain of a variable is initially a list of values from 1 to 9. Finally, there are 27 constraints: all 9 rows, 9 columns and 9 subsquares need to have different values.

2.3 AC-3 Algorithm

The AC-3 algorithm is one of the algorithms for solving CSPs [3]. The algorithm makes use of arcs. Each arc is a set of two variables, which constraint each other. The goal of the algorithm is to keep removing values from domains of variables in order to reach arc consistency. Arc (x, y) is consistent, if for every value in d_x, there exists a value in d_y such that it satisfies the constraint.

The algorithm keeps a priority queue of arcs and runs while the queue still contain arcs. In every iteration, an arc, say (x, y) is taken from the queue, and revised for consistency. If the revision caused changes in d_x, we need to check for consistency from the other direction, hence we add arc (y, x) to the queue. The algorithm terminates once there are no more arcs, hence we have satisfied all constraints. If this is the case, we return true. If we end up with one of the domains being empty, it is impossible to satisfy arcs for the given instance of the problem, and the algorithms returns false.

Following is the pseudo-code (inspired by [3] and [2]):

```
function AC-3(CSP):
    queue = GENERATE-ARCS(CSP)     // list of arcs, initially all arcs
    while queue is not empty do:
        (Xm,Xn) = REMOVE-FIRST(queue)
        REVISE(CSP, Xm, Xn)
        if new size of Dm = 0:
            return false
        if Dm has been changed:
            for each Xk in {neighbors of Xm} { Xn:
                if (Xk, Xm) not on queue:
                    add (Xk, Xm) to queue
    return true

function REVISE(CSP, Xm, Xn):
    bool change = false
    for each vx in D(x):
        find a value vy in D(y) such that vx and vy satisfy the constraint R2(x, y)
        if there is no such vy:
            D(x) = D(x) - vx
            change = true
     return change
```

3 Implementation

3.1 Structure

In our implementation, the AC-3 algorithm is implemented inside a `Solver` class, which constains a sudoku object for solving. `Sudoku` class contains a grid which is a two-dimensional list. This list consists of objects of class `Cell`. Each cell constains its `value`, `domain` and `neighbors` (which is essentially just a list of references to other cell objects). The class `Arc` contains 2 cells (`fst`, `snd`) and `heuristics`. The `__lt__(self, other)` function inside the `Arc` class then behaves differently based on the given heuristics.

3.2 AC-3 Algorithm for Sudoku

In the case of sudoku, every cell creates an arc with each of its neighbors (as these are the cells that constraint it). The arcs are then put into the queue. The arc generation can be seen in the following code snippet:

```
def generate_arcs(self):
    """ Initializes the priority queue to contain all arcs """
    q = PriorityQueue()
    for i in range(9):
        for j in range(9):
            for neighbor in self.sudoku.grid[i][j].neighbors:
                arc = Arc(self.h, self.sudoku.grid[i][j], neighbor)
                q.put(arc)
    return q
```

Listing 1: Function for arc generation

The function `solve` implements the AC-3 algorithm. The algorithm is pretty much identical to that of the pseudo-code, but extended with incrementing the `self.steps` counter in each iteration. The counter is an attribute of `Solver` and serves as a measure to compare different heuristics. The `self.q` is the queue, which is initialized by the previously mentioned arc generation function.

```
def solve(self):
    """ AC-3 algorithm """
    """ Returns true if constraints can be satisfied, returns false
     otherwise"""
    while not self.q.empty():
        self.steps += 1
        arc = self.q.get()
        if self.revise(arc):
            if not arc.fst.domain:
                return False
            for neighbor in arc.fst.get_other_neighbors(arc.snd):
                newarc = Arc(self.h, neighbor, arc.fst)
                if newarc not in self.q.queue:
                    self.q.put(newarc)
    return True
```

Listing 2: The AC-3 algorithm for sudoku

The revise function also follows directly from the pseudo-code:

```
def revise(self, arc):
    """ Returns true if domain of arc[0] is revised, returns false
    otherwise """
    revised = False
    orig_domain = list.copy(arc.fst.domain)
    for value in orig_domain:
        if not self.consistent(value, arc.snd):
            arc.fst.remove_from_domain(value)
            revised = True
    return revised
```

Listing 3: Revising function

Finally, the function to check whether an arc is consistent. Here, we check whether there exists a value in the domain of the second element of arc, `cell`, which is different to the value from domain of `arc.fst`, which we passed to the algorithm as `value`. If this is the case, we know immediately that the this value can stay in the domain of `arc.fst` and we return true. Otherwise the return false to indicate that the value should be removed.

```
def consistent(self, value, cell):
    """ Returns true if arc is consistent """
    for value2 in cell.domain:
        if value != value2:
            return True
    return False
```

Listing 4: Revising function

3.3 Heuristics

We have implemented and tested four different heuristics. As mentioned before, each heuristic is simply just a different ordering of arcs inside the priority queue.

First, we show that we order cells based on the size of their domain:

```
def __lt__(self, other):
    """ Operator < for domain size (necessary for priority queue of
     arcs) """
    return len(self.domain) < len(other.domain)

def __gt__(self, other):
    """ Operator >= for domain size (necessary for priority queue
    of arcs) """
    return len(self.domain) >= len(other.domain)
```

Listing 5: Comparison functions for cells

The following code snippet shows the implementation of `__lt__(self, other)` inside the class `Arc`. We can see that different heuristics result in different orderings:

```
def __lt__(self, other):
    """ Operator < for arcs (necessary for priority queue) """
    # no deliberate ordering
    if self.heuristics == 0:
        return True

    # order both elements inside by smaller domain
    if self.heuristics == 1:
        if len(self.fst.domain) == len(other.fst.domain):
            return self.snd < other.snd
        else:
            return self.fst < other.fst

    # order both elements inside by bigger domain (best)
    if self.heuristics == 2:
        if len(self.fst.domain) == len(other.fst.domain):
            return self.snd > other.snd
    else:
        return self.fst > other.fst

    # first domain is smaller, second domain is bigger
    if self.heuristics == 3:
        if len(self.fst.domain) == len(other.fst.domain):
            return self.snd > other.snd
        else:
            return self.fst < other.fst

    # first domain is smaller, second domain is bigger
    if self.heuristics == 4:
        if len(self.fst.domain) == len(other.fst.domain):
            return self.snd < other.snd
        else:
            return self.fst > other.fst
```

Listing 6: Comparison function for arcs

4 Results

4.1 Tests

As shown above, we have implemented 4 different heuristics:

1. Ordering both elements by smaller domain
2. Ordering both elements by bigger domain
3. Ordering first element by smaller domain and second element by bigger domain
4. Ordering first element by bigger domain and second element by smaller domain

In the table below, we can see how many steps the algorithm took to solve the sudokus

(alternatively, to find out the constraints cannot be satisfied). We show the average number of steps for all sudokus, as well as the average number of steps for only the solved sudokus. Neither of the heuristics could solve sudokus 3 and 4, while sudokus 1,2 and 5 were solved by all.

-	**Sudokus**					-	-
Heuristics	**1**	**2**	**3**	**4**	**5**	**Avg All**	**Avg Solved**
None	4319	4275	3727	2669	4102	3818.4	4232.0
1	6616	6688	5847	4768	6112	6006.2	6472.0
2	5123	4724	4321	3135	4373	4335.2	4740.0
3	3993	3781	3300	2473	3463	3402.0	3745.6
4	3993	3781	3300	2473	3463	3402.0	3745.6
Solved?	Yes	Yes	No	No	Yes	-	-

Table 1: Testing AC-3 algorithm over 5 sudokus

4.2 Interpretation

We can clearly see that implementing no heuristics is not a good option. Generally, a well-chosen heuristics will help to satisfy the arcs faster. While having no heuristics at all already gave quite satisfying results, we have been able to push the number of steps down a bit. Heuristics number 3 (sorting each element differently) gave the best results. Interesting observation is that heuristic number 3 and number 4 are symmetrical and result in the same number of steps. Furthermore, heuristics number 2 (sorting by bigger domain) gives a small deterioration. Heuristics number 1 (sorting by smaller domain), gave the worst results, worse than having no heuristics at all.

5 Conclusions

We have described our implementation of sudoku solver through the AC-3 algorithm. We have seen that experimenting with different queue orderings can be beneficial. The algorithm managed to solve 3 of 5 provided sudokus. It would be interesting to further implement the AC-3 algorithm in combination with backtracking to be able to tackle the sudokus which are not solvable using AC-3 only. Furhtermore, there might be more, complicated heuristics (which do not only treat queue ordering based on domain lenght) to see what effect that would have on the results.

References

[1] Andries E. Brouwer. Digit patterns and jigsaw puzzles. https://homepages.cwi.nl/~aeb/games/sudoku/solving28.html. [Online; accessed 29-November-2021].

[2] Paul Kamsteeg and Elena Marchiori. Ai: Principles and techniques, constraint satisfaction problems (slides). https://brightspace.ru.nl/d2l/le/content/260099/viewContent/1416696/View. [Online; accessed 30-November-2021].

[3] Wikipedia contributors. Ac-3 algorithm. https://en.wikipedia.org/wiki/AC-3_algorithm. [Online; accessed 30-November-2021].

[4] Wikipedia contributors. Constraint satisfaction problem. https://en.wikipedia.org/wiki/Constraint_satisfaction_problem. [Online; accessed 29-November-2021].