Solving Sudoku's with AC-3

Bram Pulles – S1015194

November 23, 2022

Contents

1 Introduction

Sudoku is a popular puzzle, it attracts attention from puzzle enthusiasts, mathematicians and computer scientists. Over the years computer scientists have worked on various ways of solving and generating Sudoku's. One technique to solve Sudoku's is based on the algorithm AC-3 (short for arc consistency), which is a technique more generally used in constraint satisfaction problems. In this paper we take a look at how AC-3 can be used to solve Sudoku's. We also implement the algorithm and benchmark the performance using a variety of heuristics.

To this end, we start by taking a look at the pseudocode of the algorithms for AC-3, the depth first search (DFS) which uses AC-3 and the heuristics used in the DFS. Next, we discuss the benchmark results comparing the performance of the different heuristics. Finally, we consider the conclusions which can be drawn from this work. In the appendix there are a few notes on the implementation itself.

2 Algorithms

Sudoku's can be solved using a plain DFS algorithm. However, this might result in long solving times. AC-3 is an improvement on top of the DFS. Using AC-3 the number of possible values for every field in the Sudoku can be greatly decreased during search. This results in a way smaller search space for DFS and therefore greater performance.

A heuristic can be applied to select the best field to consider at every iteration in DFS. Yet another heuristic can be utilised to provide an ordering on the possible values to be tried for this field. Depending on the heuristics chosen this can greatly affect the performance of the DFS.

2.1 AC-3

The AC-3 algorithm consists of three basic types. First, we have all the variables, which for Sudoku is one variable for every field in the Sudoku. Second, we got a domain of all the possible values for every variable, for Sudoku this is `[1, ..., 9]` for all the empty fields and just `[n]` for every field containing a number `n`. Third, we have a set of relations which for every field provides all the other fields in relation to this field, i.e. in the same row, column or block.

Using just these three pieces of information we can perform the AC-3 algorithm, see 1. The AC-3 algorithm works over arcs, which are ordered tuples consisting of two fields which have a relation to one another. We start the algorithm with a worklist of arcs we want to consider. While we have arcs left in the worklist we try to reduce a domain, and if we reach an empty domain we known that we cannot create arc consistency. This information can be used to prune branches in the DFS! If we do not reach an empty domain but the AC-3 algorithm terminates, we have reduced all the domains thus decreasing the size of the search space to be considered in the DFS.

The contents of the worklist when we start the AC-3 algorithm can be of two different types. Before we start the DFS algorithm we use AC-3 consistency to reduce the domains of all the fields. In this situation we start with a worklist containing *all* the possible arcs between any two fields. During the DFS we initialize the worklist based on the field which we are trying to fill with a value. The worklist then consists of all the arcs containing the field we are considering and any field which has a relation to this one.

2.2 DFS

The DFS algorithm is very basic, see 2. We have all the normal ingredients and no specialties. We have the terminating condition, which checks if we have found a solution by looking at the domains. And, we have the recursive part which loops over all the values for a certain field and tries them recursively if the arcs are consistent. We also include a timeout which is essential for automated benchmarking. Note that before we run the DFS we run the AC-3 algorithm and it is possible that we already find a solution in which case DFS is not even needed.

Algorithm 1 AC-3

Require: domains of all fields, relations between fields, worklist of arcs to be checked
Ensure: arc consistent domains for all fields

```
1: function AC3(domains, relations, worklist)
2:     while worklist is not empty do
3:         field one, field two := arc := pop worklist
4:         if ARC-REDUCE(arc) then
5:             if domain of field one is not empty then
6:                 worklist := worklist ∪ all arcs from field one to any other relating field
7:             else
8:                 return false
9:             end if
10:        end if
11:    end while
12:    return true
13: end function
14: function ARC-REDUCE(arc)
15:    if domain size of field two > 1 then
16:        return false
17:    end if
18:    value := the value in the domain of field two
19:    if value in domain of field one then
20:        domain field one := domain of field one \ value
21:        return true
22:    end if
23:    return false
24: end function
```

Algorithm 2 DFS with AC-3

Require: domains of all fields, relations between fields, heuristic 1 for choosing a field, heuristic 2 for choosing a value, timeout event signaler
Ensure: solved sudoku

```
1: function DFS(domains, relations, heuristic 1, heuristic 2, timeout)
2:     if timeout then
3:         return null
4:     end if
5:     if all domains have size 1 then
6:         return solution
7:     end if
8:     field := best empty field using heuristic 1
9:     values := best ordering of values from domain using heuristic 2
10:    for value in values do
11:        if arc consistent when setting value then
12:            solution := DFS(domains, relations, heuristic 1, heuristic 2, timeout)
13:            if solution is not null then
14:                return solution
15:            end if
16:        end if
17:    end for
18:    return null
19: end function
```

2.3 Heuristics

The heuristics are used for two different purposes, choosing the next field to consider and choosing the next value of this field to consider, we call these field and value heuristics respectively. We have implemented 5 different field heuristics and 3 different value heuristics.

The field heuristics are as follows:

- NOP (No OPeration): This heuristic always returns the same value and is useful as a control group in the experiments. Note, in practise we get a lexicographical ordering.
- LF (Least Finalized): Fields with few related finalized fields, i.e. fields with a domain size of one, are scored better.
- MF (Most Finalized): Fields with many related finalized fields, i.e. fields with a domain size of one, are scored better.
- LRV (Least Remaining Values): Fields with a small domain are scored better.
- MRV (Most Remaining Values): Fields with a big domain are scored better.

The value heuristics are as follows:

- NOP (No OPeration): This heuristic always returns the same value and is useful as a control group in the experiments. Note, in practise we get a lexicographical ordering.
- LCV (Least Constraining Values): Values putting few constraints on other domains are scored better.
- MCV (Most Constraining Values): Values putting many constraints on other domains are scored better.

The complexities of the heuristics are easily determined as all of them are a single line of code.

- NOP: $\mathcal{O}(1)$ trivially.
- LF and MF: $\mathcal{O}(1)$ since we count the number of related fields with a domain size of 1. More specifically, we need to check $8 \cdot 3 = 24$ domains.
- LRV and MRV: $\mathcal{O}(1)$ trivially.
- LCV and MCV: $\mathcal{O}(1)$ since we check for every related field if the value considered is in their domain. More specifically, this takes at most $24 \cdot 9 = 216$ value check operations.

3 Benchmarks

Before we can actually start benchmarking the code we need a considerate amount of Sudoku puzzles to get statistically significant results. We have tried various different databases of puzzles, but we found that in most datasets the puzzles were either too hard or too easy. Finally, we found a good dataset which is sampled with a uniform probability conditional on the number of clues. This dataset is created in such a way as to be representative of Sudoku's in general and therefore a great set to benchmark on.[1]

3.1 Results

The results of the benchmarks can be seen in table 1.

[1] The dataset is from `tdoku`, see `https://github.com/t-dillon/tdoku/blob/master/benchmarks/README.md`.

field	value	total	timeouts	mean time	mean count	max count
nop	nop	200	51	2.1821	8873	53970
nop	lcv	200	38	1.6438	5435	45879
nop	mcv	200	41	1.4703	5322	51701
lf	nop	200	198	4.0879	27367	27367
lf	lcv	200	196	2.2875	16815	37611
lf	mcv	200	198	4.654	30614	30614
mf	nop	200	3	0.6186	1463	19936
mf	lcv	200	4	0.6745	1591	23407
mf	mcv	200	4	0.6864	1568	20541
lrv	nop	200	60	1.6605	6542	47999
lrv	lcv	200	58	1.6626	8350	61334
lrv	mcv	200	61	1.2935	4876	51165
mrv	nop	200	198	9.1331	72412	72412
mrv	lcv	200	197	0.0746	390	44874
mrv	mcv	200	199	0.1839	1485	1485

Table 1: Benchmark results on Sudoku's from the unbiased dataset. The `field` column describes the field heuristic used. The `value` column describes the value heuristic used. The `total` column describes the total number of Sudoku's evaluated, in this case 200. The `timeouts` column describes the total number of timeouts that occurred during DFS, with a timeout set to 10 seconds. The `mean time` column describes the mean of all the benchmark times (without timeouts) in seconds. The `mean count` describes mean number of nodes evaluated during all DFS. The `max count` describes the maximum number of nodes evaluated during all DFS.

3.2 Discussion

There are a lot of things that can be seen from the benchmark results. Let us first take a look at the field heuristic. First we see that some setups, namely all with the LF or MRV heuristic nearly always reached a timeout. We can also see that the MF heuristic reached almost no timeouts. The LRV and NOP field heuristics performed considerably worse than the MF heuristic. It is interesting to see that the field heuristic, opposed to the value heuristic, mostly determines the performance of the algorithm.

Looking at the value heuristic there seem to be contradicting results. When using a NOP field heuristic the LCV value heuristic seems to perform significantly better than the NOP value heuristic. However, when we using a MF field heuristic the LCV value heuristic seems to perform slightly worse than the NOP value heuristic. This is surprising as the NOP heuristic provides effectively no ordering. This might be explained by the number of operations performed when calculating LCV which can be more than 200 operations. This seems insignificant, but it can explain the minor degradation in performance when compared to the NOP heuristic. Also, the MF field heuristic results in an optimally small domain and thus a small number of values need to be ordered, so the ordering would likely have less of an impact. This hypothesis is further supported since the LCV and MCV heuristics perform basically equivalently under MF.

4 Conclusion

When solving Sudoku's with DFS and AC-3, the best performing field heuristic is MF, compared to NOP, LF, LRV and MRV. The field heuristic largely determines the performance of the algorithm. The best value heuristic under MF is NOP, compared to LCV and MCV. However, in general, the value heuristic does not have much influence on the algorithms performance.

A Implementation

The implementation is done in Python. Two input formats are supported. The block format from the assignment can be read, this is extended so that multiple of these blocks can be put in one single file, see `data/sudoku.txt`. The line format which is common on the internet is also supported, see e.g. `data/unbiased.txt`.

The AC-3 graph is implemented as a class in which all the relations are once created at initialisation. Whenever arc consistency is tested all the updates done to domains, i.e. removing values from field domains, are saved in a history. If no arc consistency can be reached or the DFS fails the history is used to reset the AC-3 domains.

Some files contain a main which was used for testing, see `ac3.py`, `solver.py` and `sudoku.py`. Benchmarks are run with `benchmark.py` utilising 15 out of 16 cores in my CPU to drastically decrease the benchmark time from 8.3 hours to a mere 30 minutes.

Correctness of the results has been checked with `isvalid`, see `sudoku.py`.