The CSP Sudoku

Informed Search Methods in AI

Assignment 1

September 12, 2020

Jón Steinn Elíasson
jonsteinn@gmail.com

Martin Titze
martin20@ru.is

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Sudoku is a popular logic puzzle invented by Howard Garns in the 70s but popularized and named in Japan in the following decade[1]. To solve it, one usually applies a mixture of logic and trial and error. There are several variations but we will only concern ourselves with the classical Sudoku, played on a 9×9 grid where there exists a unique solution.

1.1 Sudoku

For convenience and as is often done in computing, we will draw our Cartesian coordinate system with the y-axis increasing downwards. We start with a few definitions in order to understand the game.

Definition 1.1. A *cell* (a, b) is the region $[a, a+1) \times [b, b+1) \subseteq \mathbb{R}^2$ where $a, b \in \mathbb{N} = \{0, 1, 2, \ldots\}$.

Definition 1.2. A $n \times m$ *grid* $\mathcal{G}$, where $(n, m) \in \mathbb{Z}^+ \times \mathbb{Z}^+$, is the set of cells $\{(a, b) \mid a < n \text{ and } b < m\}$.

Definition 1.3. A *row* $r \in \{0, \ldots, m-1\}$ and *column* $c \in \{0, \ldots, n-1\}$ in a $n \times m$ grid are the sets of cells $\{(i, r) \mid 0 \leq i < n\}$ and $\{(c, j) \mid 0 \leq j < m\}$ respectively. A *block* in a $n^2 \times n^2$ grid is a set of cells $\{(x+i, y+j) \mid i, j \in \{0, 1, \ldots, n-1\}\}$ where $x, y \in \{0, n, \ldots, n^2 - n\}$ are fixed.

In Figure 1 we can see a 9×9 grid with examples of a cell, row, column and a block.

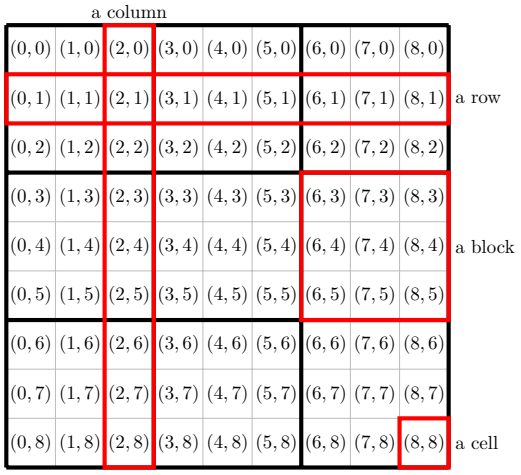

Figure 1: A 9×9 grid and all of its subcomponents.

Definition 1.4. Given a $n^2 \times n^2$ grid $\mathcal{G}$ and two cells $c_1, c_2 \in \mathcal{G}$ where $c_1 \neq c_2$, then c_1 and c_2 are said to be *peers* if there exists a row, column or block in $\mathcal{G}$ containing both.

On a 9×9 grid, each cell will have exactly 20 peers, with 8 from a row, 8 from a column and 4 from a block when excluding the ones already counted by the row and column.

Definition 1.5. Let $\mathcal{G}$ be a $n^2 \times n^2$ grid. A map $\phi : \mathcal{G} \mapsto \{1, 2, \ldots, n^2\}$ is called a *complete assignment* and a map $\varphi : \mathcal{C} \mapsto \{1, 2, \ldots, n^2\}$ where $|\mathcal{C}| < |\mathcal{G}|$ and $\mathcal{C} \subseteq \mathcal{G}$ is called a partial assignment. In both cases, $\phi(c)$ and $\varphi(c)$ are called an *assignment* of a cell $c \in \mathcal{G}$

Definition 1.6. An assignment (either partial or complete) ϕ for a $n^2 \times n^2$ grid is said to be *invalid* if there exists cells c_1 and c_2 in the domain of ϕ such that $c_1 \neq c_2$, c_1 and c_2 are peers and $\phi(c_1) = \phi(c_2)$. If an assignment is not invalid, it is *valid*.

As we only care about classical Sudoku we will shift focus from a general n^2 to the fixed number 9 in terms of definitions.

Definition 1.7. A (classical) *Sudoku puzzle* is a pair $(\mathcal{G}, \varphi)$ where $\mathcal{G}$ is a 9×9 grid and φ a valid partial assignment $\varphi : \mathcal{C} \to \{1, 2, \ldots, 9\}$ where $\mathcal{C}$ is a proper subset of $\mathcal{G}$ such that

$$|\{\phi \mid \phi \text{ is a valid complete assignment and } \varphi(c) = \phi(c) \text{ for all } c \in \mathcal{C}\}| = 1.$$

We refer to this unique valid complete assignment as a *solution* to a Sudoku puzzle. A Sudoku puzzle and its solution can be seen in Figure 2.

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

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

Figure 2: A Sudoku puzzle and its solution.

In essence, a Sudoku puzzle is a 9-coloring problem where each color represents a number from $\{1, 2, \ldots, 9\}$, the cells are vertices and two vertices have an edge between them if they are peers and we are provided with a few colorings such that the remaining ones can only be colored in a single way.

1.2 Sudoku as a CSP

There are different ways to model Sudoku as a CSP. We will let all cells be variables, regardless of them being assigned or not, and instead reduce the domains to singleton sets for assigned cells. The constraints are all non-equal binary constraints between peers. Alternatively, we could have omitted the assigned cells from the variables and introduced some unary constraints on their peers.

Definition 1.8. A *Sudoku CSP* for a Sudoku puzzle with partial assignment $\varphi : \mathcal{C} \to \{1, 2, \ldots, 9\}$ is a triple (X, D, C) with *variables* $X = \{x_0, x_1, \ldots, x_{80}\}$ where x_i represents the cell $c_i = (i \pmod 9, \lfloor i/9 \rfloor)$, *domains*

$$D_i = \begin{cases} \{\varphi(c_i)\} & \text{if } c_i \in \mathcal{C} \\ \{1, 2, \ldots, 9\} & \text{otherwise} \end{cases}$$

and *constraints* $C = \{(i, j) \mid i < j \text{ and } c_i \text{ and } c_j \text{ are peers}\}$ for $i, j \in \{0, 1, \ldots, 80\}$.

Our goal is to find the assignment of the variables, $(v_0, v_2, \ldots, v_{80}) \in D_0 \times D_1 \times \cdots \times D_{80}$, such that all constraints are satisfied. An element $(i, j) \in C$ represents the constraint that x_i and x_j must not be equal. Note that we restrict our constraints to include only ordered pairs as non-equality is a reflexive relation over $\{1, 2, \ldots, 9\}$. For example C would contain (x_2, x_{38}) but not (x_{38}, x_2). Figure 3 shows all variables of a Sudoku CSP on a grid as well as all constraints involving the variable x_{38}. The green cells represent the peers of x_{38} where x_{38} would be the second element of the pair in C, the blue cells represent the peers where x_{38} would be the first one and finally, x_{38} itself is highlighted with red. Since each variable has 20 peers and the order does not matter (with regards to counting), there are $\frac{9 \cdot 9 \cdot 20}{2} = \frac{1620}{2} = 810$ unique constraints in the entire CSP. We divide by 2 to avoid counting each pair twice.

Definition 1.9. A Sudoku CSP (X, D, C) is *arc-consistent* if for all variables $x_i, x_j \in X$ where x_i and x_j represent non-equal cells that are peers and for all values $v \in D_i$ the set $D_j \setminus \{v\}$ is not empty.

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
x_9	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}
x_{18}	x_{19}	x_{20}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}
x_{27}	x_{28}	x_{29}	x_{30}	x_{31}	x_{32}	x_{33}	x_{34}	x_{35}
x_{36}	x_{37}	x_{38}	x_{39}	x_{40}	x_{41}	x_{42}	x_{43}	x_{44}
x_{45}	x_{46}	x_{47}	x_{48}	x_{49}	x_{50}	x_{51}	x_{52}	x_{53}
x_{54}	x_{55}	x_{56}	x_{57}	x_{58}	x_{59}	x_{60}	x_{61}	x_{62}
x_{63}	x_{64}	x_{65}	x_{66}	x_{67}	x_{68}	x_{69}	x_{70}	x_{71}
x_{72}	x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}	x_{79}	x_{80}

Figure 3: The variables of Sudoku and the constraints involving the variable x_{38}.

2 Results

Our algorithms can be seen in appendix A. All our searches will suffer from the fixed expansion order. Suppose the domain for x_j is the singleton set $\{v\}$, and we assign v to x_i, a peer of x_j, with $i < j$. Then we do not check the constraint (i, j) until at depth j even though it is very clear that x_i can not be assigned v. This is fixed with our preprocessing as v would be removed from D_i. It can also be made better by clever back jumping.

In Figure 4 we can see the Sudoku puzzles provided for benchmarking. The performance of our algorithms without and with arc-consistency can be seen in Table 1, all of which produced a valid solution.

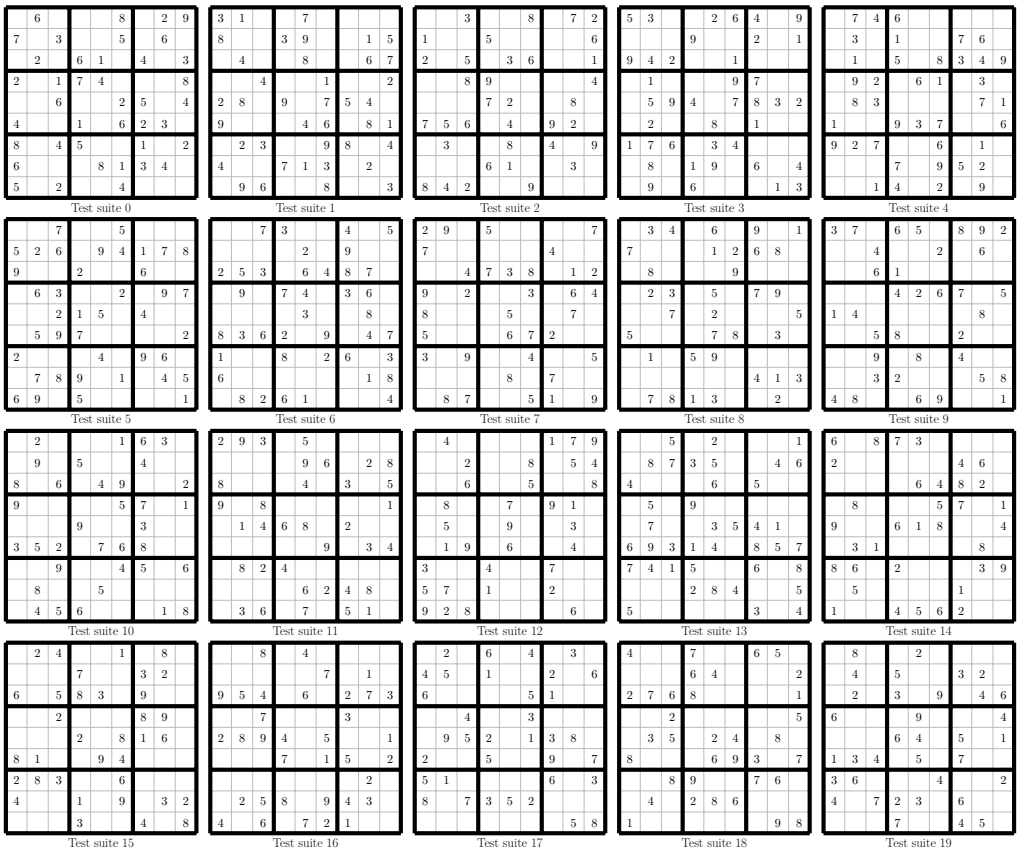

Figure 4: All puzzles in the provided test suite.

	BT		BJ		CBJ	
Test suite	Nodes	Time(s)	Nodes	Time(s)	Nodes	Time(s)
0	10352	0.3891	1476	0.0707	1014	0.0345
1	95775	4.0698	1863	0.051	1811	0.0456
2	185358	8.3867	1375	0.0587	1103	0.0441
3	210448	7.7151	27264	0.9803	16792	0.5794
4	228100	7.2007	14753	0.4094	13557	0.3429
5	274480	9.5221	41637	1.3549	22049	0.6689
6	284079	8.3254	5084	0.117	3869	0.0861
7	302346	12.7633	16895	0.7814	14021	0.5497
8	339699	24.4524	3413	0.2283	3023	0.1789
9	372341	16.7888	25137	1.27	14833	0.6722
10	486884	35.5491	7005	0.3473	6525	0.3178
11	1123918	64.8931	37668	1.918	32282	1.4768
12	1181914	58.1829	15281	0.5672	12757	0.4258
13	1247724	54.9106	87518	3.4041	24473	0.9547
14	1962926	79.79	87572	3.2575	69020	2.3439
15	2320445	112.1436	75454	2.9944	29344	1.1991
16	2744246	86.076	122345	3.672	37604	1.7481
17	3494453	165.2988	272558	10.2189	116351	4.2637
18	8370963	503.3993	29105	1.2658	21899	0.9765
19	12632761	605.4648	27060	0.8237	24086	0.7206

	BT		BJ		CBJ	
Test suite	Nodes	Time(s)	Nodes	Time(s)	Nodes	Time(s)
0	81	0.0019	81	0.0029	81	0.0025
1	81	0.002	81	0.0021	81	0.0029
2	81	0.0027	81	0.0021	81	0.0099
3	81	0.0025	81	0.0059	81	0.0036
4	81	0.0026	81	0.0037	81	0.0049
5	81	0.0042	81	0.0042	81	0.004
6	81	0.0046	81	0.0066	81	0.0046
7	81	0.004	81	0.0021	81	0.0024
8	81	0.0033	81	0.0029	81	0.0036
9	81	0.0022	81	0.0038	81	0.0035
10	81	0.0044	81	0.0033	81	0.0035
11	427	0.0119	353	0.0103	316	0.0097
12	81	0.0022	81	0.0031	81	0.0028
13	81	0.0023	81	0.0019	81	0.0045
14	81	0.0023	81	0.0021	81	0.0022
15	81	0.0024	81	0.0024	81	0.0023
16	81	0.0026	81	0.0023	81	0.0031
17	81	0.0093	81	0.0028	81	0.0076
18	81	0.0028	81	0.0031	81	0.0021
19	81	0.0051	81	0.0022	81	0.0025

Table 1: Performance of all three algorithms on the test suites without and with arc-consistency.

It is of no surprise that the performance increases as we become smarter with backtracking. Backjumping is an improvement on backtracking and conflict-directed backjumping is an improvement on backjumping. The most surprising results is probably that sometimes runs with less nodes expanded can take longer. This could be due to luck (or lack thereof) when checking constraints. As we stated earlier, each cell has 20 peers and in one run we could find the conflicting ones early on average while late in another.

Sudoku puzzles are often rated as easy or hard. These ratings do not necessarily translate to the algorithms we have implemented, i.e. an "easy" puzzle can be hard for a backtracking algorithm. In fact, puzzles can be designed to be hard for backtracking. The more assigned values the puzzle has, the easier - this most algorithms (and humans) can agree on. Another criteria for easiness is how many new assignments can be concluded by simple logic. That would however not make the problem any easier for backtracking algorithms, unless we preprocess the domains.

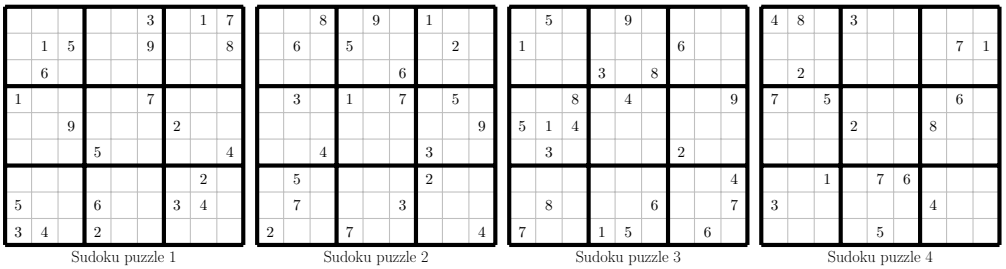

Figure 5: The additional Sudoku puzzles used for experiments with arc-consistency.

When the test suites were run with arc-consistency, only one was not solved directly by it and that was test suite 11. It required 427, 353 and 316 nodes to be expanded for backtracking, backjumping and conflict-directed backjumping respectively and all 20 test suites were solved within 0.012 seconds by all algorithms. With such a minuscule difference in performance, we needed different puzzles to compare the

algorithms with arc-consistency. We chose 4 puzzles from the additional puzzles provided and ran the three algorithms for all of them. The puzzle we chose can be seen in Figure 5. The performance of our algorithms can be seen in Table 2. Like before, the algorithms that are built as improvements outperform their prototypes, as is expected.

	BT		BJ		CBJ	
Sudoku puzzle	Nodes	Time(s)	Nodes	Time(s)	Nodes	Time(s)
1	2339	0.1174	1652	0.0741	764	0.03
2	160054	9.5492	90200	3.529	54765	1.734
3	540279	25.149	353457	13.6457	152490	5.3534
4	478270	30.9214	340839	17.3177	189604	8.8046

Table 2: Performance of all three algorithms on the additional puzzles with arc-consistency.

3 Conclusion

The project serves as a valuable lesson in improvements on backtracking. The overhead of the conflict set in the conflict-directed backjumping is minuscule and the algorithm outperforms the other two on all nontrivial puzzles. The same can be said for AC-3 which always seems beneficial to apply. These two as a pair form the most powerful tool we built in the assignment.

Still there were some Sudoku puzzles we struggled to solve (within a reasonable time) with what we implemented. Some improvements could involve heuristics on assignment- and expansion order. An example of a heuristic for assignments could be exploring values with fewer total assignments first. An example of a heuristic for variable order could involve the size of the domain or the number of constraints to unassigned variables or even a weighted combination of both.

There are also some algorithms we did not implement such as backmarking or dynamic backtracking as well as other preproccessing methods that could help solve those puzzles our implementations are not capable of solving within a reasonable time. There are even algorithms outside the realm of CSPs which have been used to solve Sudoku such as stochastic optimization algorithms as is discussed in [2].

Appendices

A Algorithms

The implementation of our algorithms can be found in [3].

A.1 Preprocessing

We preprocess the CSP to reduce the domains by enforcing arc-consistency. We do that with AC-3 as shown in Algorithm 1 that utilises the helper shown in Algorithm 2.

Algorithm 1 AC-3

Input: A constraint network cn that offers access to peers and constraints.

function MAKE_ARC_CONSISTENT(cn)
  $Q \leftarrow$ queue of all constraints ▷ both (a, b) and (b, a) are included
  while Q is not empty **do**
    $(i, j) \leftarrow$ deque(Q)
    if revise(cn, i, j) **then**
      for all $h \in \{x \mid x$ and i are peers$\}$ **do**
        if $h \neq j$ **then**
          enque$(Q, (h, i))$
        end if
      end for
    end if
  end while
end function

Algorithm 2 Revise

Input: A constraint network with access to domain array D and a consistency check c and variable indices i and j.

function REVISE(cn, i, j)
  $R \leftarrow \{v \in D[i] \mid \neg\exists w \in D[j]$ s.t. $c(i, j, v, w)\}$
  if $R \neq \emptyset$ **then**
    $D[i] \leftarrow D[i] \setminus R$
    return True
  end if
  return False
end function

Since we only store constraints as an ordered pair, one issue when translating the algorithm from [4] was that it expects both constraints (a, b) and (b, a) to be present in the queue at start, since it is not assuming constraints to be a symmetric relation. This is done with a helper function that is omitted in Algorithm 1.

A.2 Searches

Our algorithms for chronological backtracking, backjumping search and conflict-directed backjumping search can be seen in Algorithm 3, Algorithm 4 and Algorithm 5 respectively. Since our network is using zero-based indexing while [4] uses one-based, we must adjust our algorithms for that. We also add a return on success mechanism that disregards the jumping value. This was done because the algorithms in [4] are not returning

(or in our case, updating a parameter that is passed by reference) the solution but rather just printing it when found.

Algorithm 3 Chronological backtracking

Input: A constraint network cn with access to number of variables n, domain array D and constraint check c against all variables that have been assigned, the current variable i and the current assignment A.

```
function BT(cn, i, A)
    for all v ∈ D[i] do
        A[i] ← v
        if c(i) ∧ (i = n − 1 ∨ BT(cn, i + 1, A)) then        ▷ c(i) returns true if consistent
            return True
        end if
    end for
    return False
end function
```

Algorithm 4 Backjumping search

Input: A constraint network cn with access to number of variables n, domain array D and constraint check c that checks to what variable the current one is consistent up to, the current variable i and the current assignment A.

```
function BJ(cn, i, A)
    r ← −1
    for all v ∈ D[i] do
        A[i] ← v
        m ← c(i)                                          ▷ c(i) returns i if consistent
        if m = i then
            if i = n − 1 then
                return True, −1                           ▷ jump value does not matter on success
            end if
            (s, m) ← BJ(cn, i + 1, A)
            if s then
                return True, −1
            end if
            if m < i then
                return False, m
            end if
        end if
        r ← max(r, m)
    end for
    return False, r
end function
```

Algorithm 5 Conflict-directed backjumping

Input: A constraint network cn with access to number of variables n, domain array D and constraint check c that checks to what variable the current one is consistent up to, the current variable i, the current assignment A and a conflict set CS.

```
function CBJ(cn, i, A, CS)
    CS[i] ← {−1}
    for all v ∈ D[i] do
        A[i] ← v
        h ← c(i)                                   ▷ c(i) returns i if consistent
        if h < i then
            CS[i] ← CS[i] ∪ {h}
        else
            if h = i then
                return True, −1                    ▷ jump value does not matter on success
            end if
            (s, r) ← CBJ(cn, i + 1, A, CS)
            if s then
                return True, −1
            end if
            if r < i then
                return False, r
            end if
        end if
    end for
    r ← max(CS[i])
    C[r] ← CS[r] ∪ (CS[i] \ {r})
    return False, r
end function
```

References

[1] Wikipedia. Sudoku — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Sudoku&oldid=976543552, 2020. [Online; accessed 07-September-2020].

[2] R. Lewis. Metaheuristics can solve sudoku puzzles. *Journal of Heuristics Archive*, 13(4):387–401, 2007.

[3] J. Eliasson and M. Titze. SudokuCSP. https://github.com/JonSteinn/SudokuCSP, 2020.

[4] I. Miguel and Q. Shen. Solution techniques for constraint satisfaction problems:foundations. *Artificial Intelligence Review*, 15:243–267, 2001.