

AI : PRINCIPLES & TECHNIQUES
AC-3 ALGORITHM & HEURISTIC STRATEGIES

IMPLEMENTATION OF THE AC-3 ALGORITHM WITH HEURISTICS FOR SUDOKU CONSTRAINT SOLVING

By:
Juan Navarro — s1097545
Luca Laber — s1089333

Table of Contents

INTRODUCTION.....	2
METHOD.....	2
General idea of Sudoku.....	2
Sudoku as a CSP.....	2
AC-3 Algorithm Explanation.....	3
AC-3 Algorithm Implementation.....	4
COMPLEXITY.....	4
DISCUSSION.....	6
CONCLUSION.....	6
ACKNOWLEDGEMENTS.....	6

INTRODUCTION

A key component of artificial intelligence is Constraint Satisfaction Problems (CSPs), which provide an organized method for resolving issues involving decision-making under limitations. Typically, a CSP is composed of three main parts:

1. **Variables**, which represent elements of the problem that need to be assigned values.
2. **Domains**, which specify the possible values that can be assigned to each variable.
3. **Constraints**, which define relationships between variables and restrict the combinations of values that the variables can simultaneously take.

The goal of solving a CSP is to assign values to all variables from their respective domains in a way that satisfies all constraints. An example of a CSP is logic puzzles like Sudoku.

The AC-3 (Arc Consistency 3) algorithm is one of the most widely used techniques in CSP solving. It works by enforcing **arc consistency**, a property that ensures every value in the domain of a variable has a supporting value in the domain of its connected variables. By iteratively reducing variable domains and maintaining arc consistency, AC-3 simplifies the CSP, potentially solving it or preparing it for further techniques like backtracking.

This report details the CSP framework for Sudoku, the implementation of the AC-3 algorithm, and the integration of heuristic strategies. Results from the experiments provide a comprehensive understanding of the algorithm's efficiency and its potential for solving CSPs effectively.

METHOD

General idea of Sudoku

As stated above, an example of a CSP is Sudoku. But what is Sudoku?

Sudoku is a popular puzzle game that challenges players to fill a 9x9 grid with digits from 1 to 9, subject to specific constraints. The grid is divided into 81 cells, and then further grouped into nine 3x3 sub-grids (also known as blocks). The objective of Sudoku is to fill all cells in the grid such that the following constraints are met:

1. Each row must contain the digits 1 to 9 without repetition.
2. Each column must contain the digits 1 to 9 without repetition.
3. Each 3x3 sub-grid must contain the digits 1 to 9 without repetition.

Sudoku puzzles come with varying levels of difficulty, determined by the number of initially pre-filled cells and the logical complexity required to deduce the remaining numbers. Some puzzles are simple and can be solved through straightforward elimination, while others require advanced strategies.

Sudoku as a CSP

In the context of Constraint Satisfaction Problems, Sudoku can be described as follows:

- **Variables:** The 81 cells in the grid, each representing a position that needs to be filled with a digit.
- **Domains:** The possible values for each cell, initially ranging from {1, 2, 3, 4, 5, 6, 7, 8, 9}.
- **Constraints:**
 - No two cells in the same row, column, or 3x3 sub-grid can have the same value.

These constraints create a highly unified problem where the assignment of a value to one variable can significantly impact the domains of others. Solving Sudoku as a CSP involves systematically narrowing the domains of variables while satisfying the constraints, making it an ideal problem for testing the AC-3 algorithm.

AC-3 Algorithm Explanation

The AC-3 algorithm is a key preprocessing method in Constraint Satisfaction Problems (CSPs), aimed at reducing the complexity of solving by enforcing **arc consistency**. An arc, representing a direct relationship between two variables, is considered consistent if every value in the domain of the first variable has at least one corresponding valid value in the domain of the second, as defined by their constraint. The algorithm iteratively reviews all arcs in the CSP and removes any value that violates the consistency.

By systematically refining the domains of variables, AC-3 simplifies the CSP, often making it easier to solve with other techniques like backtracking. If any domain becomes empty during this process, the problem is determined to have no solution.

In Sudoku, the constraints on rows, columns, and sub-grids are naturally binary, making AC-3 a practical approach to reducing the possible values for each cell. The algorithm's efficiency depends on the order in which arcs are processed, and integrating heuristics—such as prioritizing variables with smaller domains or higher connectivity—can further enhance performance.

```
function AC-3 (CSP) returns boolean           // based on consistency
    persistent: queue           // list of Arcs, initially all Arcs
    while queue is not empty do
         $(X_m, X_n) \leftarrow \text{REMOVE-FIRST}(\text{queue})$ 
        REVISE (CSP,  $X_m$ ,  $X_n$ )
        if new size of  $D_m = 0$  then return false
        if  $D_m$  has been changed then
            for each  $X_k$  in {neighbors of  $X_m$ } -  $X_n$  do
                if  $(X_k, X_m)$  not on queue then add  $(X_k, X_m)$  to queue
    return true
```

AC-3 Algorithm Implementation

The idea behind the implementation of the algorithm was as follows. Because we were working with Constraint Satisfaction Problems, our idea was to build one ourselves:

1. We first converted the state of the sudoku into a triplet of the CSP, by obtaining the Variables (each of the fields in the 9x9), Domains (If the variable is finished, take the value and set as domain, else make it have possible values 1-9) and Constraints (all fields must be different).
2. We then proceeded to the creation of arcs between 2 variables (X_a, X_b), for which if a variable is finalized, we make arcs to all non-finalized (2 finalized don't need arc because they already have values not conflicting), else if the variable is not finalized, we create an arc to all other variables in the row, column and sub-grid.
3. We then iteratively take the first arc in the queue of all arcs, and if there is a conflicting value in the domain of the variable X_b we remove it from the domain of X_a .
 - a. Once the domain of X_a comes to a single value, assign the value to the corresponding grid.
 - b. If the Domain of $X_a = 0$, return false; since it cannot be solved.
 - c. If the Domain of X_a changes, add the arc (X_k, X_a) where k is the neighbor of $X_a - X_b$.
4. Once all arcs have been satisfied and all variables have a domain with 1 value, return true; the sudoku is solvable.

To make the `valid_solution` function, we take each row, column & sub-grid and compare it to a goal variable which is a list of numbers 1-9. If this function returns True and the `solve` returns True, the AC-3 algorithm is able to solve the sudoku, in this case sudokus 1,2,5.

COMPLEXITY

To provide us with a clear picture of the practical efficiency of the AC-3 algorithm with different heuristics, we can use the **number of arcs used** and **runtime** to therefore draw meaningful complexity conclusions. In other words:

1. **Number of Arcs Used:**
 - o Indicates the amount of work the algorithm performs. Since AC-3 processes arcs iteratively, this metric reflects the computational load of the algorithm.
2. **Runtime:**
 - o Reflects the actual time taken for the algorithm to complete, which includes both the number of arcs processed and the efficiency of the heuristic in prioritizing arcs.

	Sudoku 1	Sudoku 2	Sudoku 3	Sudoku 4	Sudoku 5
default	N. of arcs used = 1945 Runtime = 0.051s	N. of arcs used = 1993 Runtime = 0.053s	N. of arcs used = 1892 Runtime = 0.054s	N. of arcs used = 1307 Runtime = 0.043s	N. of arcs used = 1787 Runtime = 0.042s
mrv	N. of arcs used = 1610	N. of arcs used = 2258	N. of arcs used = 2666	N. of arcs used = 2312	N. of arcs used = 2429

	Runtime = 0.05s	Runtime = 0.059s	Runtime = 0.056s	Runtime = 0.045s	Runtime = 0.049s
degree	N. of arcs used = 2253 Runtime = 0.076s	N. of arcs used = 2120 Runtime = 0.077s	N. of arcs used = 2053 Runtime = 0.074s	N. of arcs used = 1358 Runtime = 0.06s	N. of arcs used = 2017 Runtime = 0.068s
finalized	N. of arcs used = 1000 Runtime = 0.078s	N. of arcs used = 1020 Runtime = 0.079s	N. of arcs used = 1040 Runtime = 0.082s	N. of arcs used = 940 Runtime = 0.086s	N. of arcs used = 920 Runtime = 0.071s
mrv degree	N. of arcs used = 2228 Runtime = 0.089s	N. of arcs used = 2144 Runtime = 0.095s	N. of arcs used = 2030 Runtime = 0.092s	N. of arcs used = 1359 Runtime = 0.076s	N. of arcs used = 1977 Runtime = 0.076s
mrv finalized	N. of arcs used = 1000 Runtime = 0.076s	N. of arcs used = 1020 Runtime = 0.088s	N. of arcs used = 1040 Runtime = 0.091s	N. of arcs used = 940 Runtime = 0.073s	N. of arcs used = 920 Runtime = 0.079s
degree finalized	N. of arcs used = 1000 Runtime = 0.099s	N. of arcs used = 1020 Runtime = 0.109s	N. of arcs used = 1040 Runtime = 0.103s	N. of arcs used = 940 Runtime = 0.086s	N. of arcs used = 920 Runtime = 0.093s
mrv degree finalized	N. of arcs used = 1000 Runtime = 0.111s	N. of arcs used = 1020 Runtime = 0.122s	N. of arcs used = 1040 Runtime = 0.124s	N. of arcs used = 940 Runtime = 0.091s	N. of arcs used = 920 Runtime = 0.103s

When looking at the results above we can see that the “**default heuristic**” processes arcs in the order they are added, therefore making it efficient for the simple puzzles (with minimum constraints) but not for the harder ones as more arcs are processed. On the other hand, “**mrv**” has its focus on the variables with the smallest domains, thus reducing the arcs for the simple puzzles. When comparing the “**finalized**” heuristic to the other heuristics we can see that the number of arcs drastically is reduced throughout all puzzles, yet the runtime is higher than other heuristics. When using a combined heuristic like “**mrv finalized**” we see that it outperforms all other heuristics, therefore we can say there is a balance between the number of arcs and the runtime. However, adding more heuristics (for example “**degree**”) may increase runtime without significant improvement in arcs processed

In conclusion, the AC-3 algorithm has a worst-case time complexity of “**O(n*log(n))**”, where “**n**” is the total number of arcs. Thus demonstrating the algorithm’s repetitive nature. While all heuristics share this theoretical worst-case complexity, their practical efficiency varies significantly based on how the arcs are prioritized.

DISCUSSION

Our study reveals that the choice of heuristic significantly impacts the practical efficiency of the AC-3 algorithm when solving Sudoku puzzles. While all heuristics operate under the same theoretical complexity, their real-world performance varies. Advanced heuristics like “**mrv**” and “**finalized**” enhance efficiency by strategically prioritizing arcs, thus reducing unnecessary computations. Combining heuristics, such as in “**mrv finalized**”, strikes an effective balance between reducing the number of arcs processed and maintaining reasonable runtime.

The findings above suggest that while the AC-3 algorithm is inherently efficient, its practical performance can be significantly improved by adapting heuristic strategies to the problem at hand. The ability to adjust heuristics based on puzzle characteristics could further enhance efficiency.

CONCLUSION

The implementation of the AC-3 algorithm for solving Sudoku puzzles has shown how thinking about the heuristic design can impact practical performance. Beyond the theoretical efficiency of the algorithm, this study demonstrates that molding the heuristic strategies to the structure and complexity of the problem can significantly enhance its effectiveness.

By experimenting with a range of heuristics, we highlighted the balance between computational effort and runtime, showcasing the flexibility of AC-3 as a tool for CSPs. This research reinforces the algorithm’s adaptability and potential for more broad applications.

In summary, AC-3 proves to be a robust and efficient framework for solving CSPs, particularly when enhanced with appropriate heuristic strategies. Future work can build on these findings by exploring dynamic or hybrid heuristics and extending the approach to more complex problem domains.

ACKNOWLEDGEMENTS

On behalf of Group 44, we want to thank the teacher, TAs for their guidance, and the content provided in brightspace that helped us understand what was needed. We also want to thank Konstantinos Konstantinou and Levente Bodi for their assistance and collaboration when working on the project.