AI: Principles and Techniques - Programming Assignment 2
Solving Sudoku using AC-3 and Heuristics

Chiara Benini
S1116239
Radboud University

February 3, 2025

1 Introduction

Constraint Satisfaction Problems (CSPs) are a fundamental topic in Artificial Intelligence. This report explores the implementation of the **AC-3 algorithm** for solving Sudoku puzzles and examines the role of **heuristics** in improving constraint propagation. Additionally, a **backtracking algorithm** is implemented to solve harder puzzles that AC-3 alone cannot solve. The study analyzes the performance of different heuristics and their effect on computational complexity.

2 Code Description

The implementation consists of multiple classes:

- **Field**: Represents a single Sudoku cell, storing its value, domain, and neighbors.
- **Sudoku**: Manages the entire Sudoku board and initializes constraints.
- **Game**: Implements the AC-3 algorithm, backtracking, and heuristic-based solving methods.
- **App**: Handles user input, file reading, and solver execution.

The AC-3 implementation is found in the `Game` class, where it iterates through the constraint queue to enforce arc consistency. The `solve()` function applies AC-3 first and, if necessary, switches to backtracking.

AC-3 Implementation

The **AC-3 (Arc Consistency 3)** algorithm is a fundamental technique in constraint satisfaction problems. The goal of AC-3 is to ensure that all variables satisfy binary constraints, thus making the problem easier to solve or even completely solving it. The AC-3 algorithm works as follows:

1. **Initialize the queue**: All arcs (variable pairs with constraints) are added to a queue. Here is where the different heuristics are applied, so the code handles queue initialization separately for convenience.

2. **Process arcs iteratively**: The algorithm dequeues an arc (X, Y) and checks whether any value in X's domain is inconsistent with all possible values in Y's domain.

3. **Revise the domain of X**: If a value in X's domain has no corresponding value in Y's domain that satisfies the constraint, it is removed.

4. **Propagate changes**: If X's domain is modified, all neighboring arcs (Z, X) are re-enqueued to ensure consistency.

5. **Termination**: The algorithm continues until no more changes occur in any domain or a domain becomes empty, indicating that no solution exists.

The function `revise(x, y)` ensures that every value in `x`'s domain has at least one supporting value in `y`'s domain. If a value in `x` is found to be inconsistent, it is removed. This process continues iteratively until the Sudoku puzzle is either simplified enough for direct solving or determined to be unsolvable.

This snippet represents the basis for the AC-3 algorithm; depending on the heuristic selected it will go over the queue in different orders.

Listing 1: AC-3

```
while queue: #while loop, stops when queue empties
    self.ac3_iterations += 1  # Track AC-3 iterations
    (x,y) = queue.popleft() #Eliminate current arc (already used)
        on the queue
    if self.revise(x,y):  #Reduce domain of X
        if len(x.get_domain()) == 0: #If there are no possible
           values left in the domain, the sudoku is not solvable
            return False
            for z in x.get_neighbours():
                if z != y and (z,x) not in queue:
                    queue.append((z,x))
```

Default Heuristic

This is the simplest approach, the most direct application of the AC-3 algorithm where arcs (constraints) are processed in the natural reading order. It starts from the top-left corner and moves row by row. Constraints are processed in the order they are encountered, **without any prioritization**. This algorithm works well for easy puzzles but is inefficient for hard ones, as it solves by itself three of the puzzles (1,2,5) but is not enough for all of them (3,4).

Listing 2: Default Heuristic

```
queue = deque()
    for i in range(9):
        for j in range(9):
```

```
                field = grid[i][j]
                for neighbour in field.get_neighbours():
                    queue.append((field, neighbour))
```

MRV (Minimum Remaining Values)

This heuristic prioritizes cells with the **fewest remaining possible values**. Before inserting constraints into the queue, the algorithm calculates how many possible numbers (1-9) can fit into each empty cell. The arcs are then sorted so that cells with fewer options come first. This one, helps solve difficult puzzles faster by reducing uncertainty early as it **prioritizes the most constrained variables**, making the puzzle easier to solve as choices narrow. Example If one cell can only contain 3, 7, while another can contain 1, 2, 3, 4, 5, 6, 7, 8, 9, MRV selects the cell with 3, 7 first.

Listing 3: MRV Heuristic snippet

```
arcs = []
            for i in range(9):
                for j in range(9):
                    field = grid[i][j]
                    for neighbor in field.get_neighbours():
                        arcs.append((len(field.get_domain()), id(
                            field), field, neighbor))

            arcs.sort(key=lambda arc: (arc[0], arc[1]))  # Sort
                by domain size first, then tie-break with id
                otherwise type error
            queue.extend([(x, y) for _, _, x, y in arcs])
```

Degree

This heuristic prioritizes cells that participate in **the most constraints**. Before inserting constraints, it calculates the number of neighboring empty cells for each cell. The arcs are sorted so that cells with the most neighbors come first. If combined with MRV, it would help break ties by picking the most influential cells and as such it's useful when multiple cells have the same small domain size. Example: A cell in the middle of a 3×3 block or a row with many empty cells will be prioritized because its value will influence more constraints. .

Listing 4: Degree Heuristic snippet

```
arcs = []
            for i in range(9):
                for j in range(9):
                    field = grid[i][j]
                    for neighbor in field.get_neighbours():
                        arcs.append((-len(field.get_neighbours())
                            , id(field), field, neighbor))

            arcs.sort(key=lambda arc: (arc[0], arc[1]))  # Most
                constrained variable first
```

```
9                queue.extend([(x, y) for _, _, x, y in arcs])
```

Constraint Propagation

This heuristic prioritizes constraints where one of the variables (Y) already has a known value. Constraints where a cell has an already assigned number are prioritized in the queue, so the information is propagated faster, leading to earlier domain reductions. It speeds up the elimination of impossible values and reduces the need for backtracking. Example: If a cell is known to be a 5, all constraints involving that cell (row, column, box) are immediately enforced before considering less constrained cells.

Listing 5: Constraint Propagation snippet

```
priority_arcs = []
            normal_arcs = []
            for i in range(9):
                for j in range(9):
                    field = grid[i][j]
                    for neighbour in field.get_neighbours():
                        if neighbour.get_value() != 0:
                            priority_arcs.append((field,
                               neighbour))  # Prioritize arcs
                               where Y is assigned
                        else:
                            normal_arcs.append((field, neighbour)
                               )
            queue = deque(priority_arcs + normal_arcs) #makes a
               complete queue with the priority arcs first and
               the other at the end
```

Backtracking

When AC-3 alone is not enough to fully solve the Sudoku, backtracking is used. Implementation:

- **Backtracking Search:** If AC-3 cannot assign values to all cells, backtracking is used to systematically explore different assignments.
- **Variable Selection**: The algorithm picks the next unassigned variable using the MRV heuristic.
- **Value Selection**: The algorithm sorts values using the Least Constraining Value (LCV) heuristic.
- **Consistency Check**: Before assigning a value, it ensures no conflicts occur.
- **Recursive Calls**: If a valid assignment is found, it continues to the next unassigned cell; otherwise, it backtracks and tries a different value.

Guarantees a solution if one exists but can be slow without good heuristics. When combined with MRV, Degree, and Constraint Propagation, it significantly reduces search

space. Example: If AC-3 narrows a cell's possibilities to 2, 4, backtracking will try 2 first. If that leads to failure, it will revert to 4.

Listing 6: Backtracking snippet

```
def backtrack(self, assignment):
        self.backtrack_iterations += 1

        if self.is_complete(assignment):
            print("Solution found!")
            return assignment

        var = self.select_unassigned_variable(assignment)
        if var is None:
            return None  # Prevents infinite recursion

        for value in self.order_domain_values(var, assignment):
            if self.is_consistent(var, value, assignment):
                var.set_value(value)
                assignment[var] = value


                result = self.backtrack(assignment)
                if result is not None:
                    return result  # Solution found
                # Backtrack: reset the Field's value and remove
                   from assignment
                var.set_value(0)
                del assignment[var]

        return None  # Failure, triggers backtracking
```

3 Complexity Analysis

3.1 Running the Complexity Study

To conduct the study, the application is started and the option to perform a complexity study is selected. Once initiated, the application follows these steps:

1. Multiple Sudoku puzzles are loaded from a specified folder.
2. Each puzzle is solved using several heuristics
3. For each heuristic, the solver is executed both with and without backtracking.
4. Performance metrics are recorded during the solving process.

3.2 Performance Metrics

The study measures the efficiency of the different solving approaches using three key metrics:

- **Time Taken:** This metric records the execution time (in seconds) required to solve each puzzle. It is crucial as it reflects the practical efficiency of the algorithm.
- **Number of Iterations:** This counts the number of iterations performed during the AC-3 (Arc Consistency) and backtracking processes. It indicates the computational effort needed to arrive at a solution.
- **Success Rate:** This metric indicates whether the solver successfully found a solution. It is important as it demonstrates the reliability of the algorithm under various configurations.

These metrics are used to compare the theoretical complexity of each algorithm with its actual observed performance.

3.3 Theoretical Complexity Analysis

Each algorithm considered in this study has a distinct theoretical complexity:

- **AC-3 (Arc Consistency 3):** With a time complexity of $O(c \cdot d^3)$, this algorithm efficiently filters out impossible values.
- **Backtracking (without heuristics):** Exhibits exponential time complexity $O(d^n)$, which becomes impractical for harder puzzles.
- **Backtracking with MRV/Degree heuristics:** By prioritizing cells that are more difficult to solve, the search space is significantly reduced, often resulting in a complexity around $O(d^{(n/2)})$.
- **Constraint Propagation:** With a time complexity of $O(n^2)$, this approach quickly reduces the search space for puzzles with well-structured constraints.

3.4 Results

These are the values extracted by running the complexity study function. Both summarized in table format and represented as a histogram for better visualization.

Heuristic	Backtracking	Solved (%)	Time (s)	Iterations
ConstraintPropagation	False	66.7%	0.23	2743
	True	100.0%	0.26	2857
Degree	False	66.7%	0.15	3291
	True	100.0%	0.19	3441
MRV	False	66.7%	0.18	3461
	True	100.0%	0.19	3537
default	False	66.7%	0.26	3263
	True	100.0%	0.17	3382

Table 1: Summary of performance metrics grouped by Heuristic and Backtracking (rounded values).

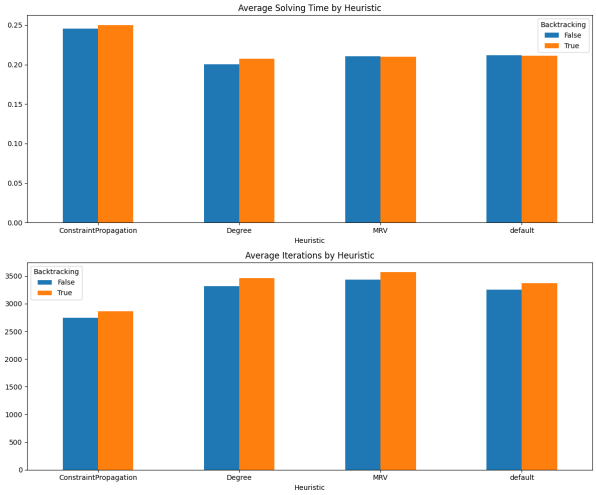

Figure 1: Histogram produced by code

4 Discussion

This section interprets the results, comparing observed performance with theoretical expectations and discussing strengths and weaknesses of different heuristics.

Impact of Backtracking The results show that backtracking is necessary for solving harder puzzles. Without backtracking, the solver only **succeeds in 66.7%** of cases, as AC-3 alone is insufficient for complex puzzles. When backtracking is enabled, the success rate reaches **100%**, confirming that heuristic-guided constraint propagation alone does not always lead to a complete solution.

4.0.1 Heuristic Performance Analysis

- **Constraint Propagation:** The best-performing heuristic in terms of execution time, as it prioritizes variables with known values and quickly eliminates impossible choices.
- **MRV and Degree Heuristics:** These heuristics optimize search order by prioritizing the most constrained variables, leading to reduced search space and more efficient solving. However, they require more iterations.
- **Default Heuristic:** The least efficient approach, as it processes constraints sequentially without considering variable difficulty. This results in higher execution time and unnecessary iterations.

4.0.2 Theoretical vs. Observed Complexity

Theoretical analysis predicted that:

- **AC-3 alone runs in** $O(c \cdot d^3)$, making it efficient for simple cases but not guaranteed to solve harder puzzles.
- **Backtracking alone is exponential** $O(d^n)$, but combining it with heuristics reduces the search space.

- **Constraint Propagation should perform best**, as it directly enforces logical constraints before resorting to search.

The experimental results align with these predictions. Constraint Propagation had the lowest iteration count, while MRV and Degree reduced search depth efficiently but required slightly more computational effort. The Default heuristic was the least efficient, as expected.

4.0.3 Limitations & Future Work

- The study only tested a limited number of puzzles. Future work could expand to larger datasets.
- A combination of heuristics (e.g., MRV + Constraint Propagation) might yield even better performance.
- Parallelizing the solver or implementing additional pruning techniques could further optimize results.

5 Conclusion

This study examined the effectiveness of different heuristics in solving Sudoku puzzles using the AC-3 algorithm and backtracking. The results confirm that AC-3 alone is insufficient for complex puzzles, while backtracking is necessary for a complete solution. Among the heuristics tested, Constraint Propagation was the most efficient, significantly reducing solving time and iteration count. MRV and Degree heuristics also improved performance, though at a higher computational cost.

The findings suggest that **heuristic selection plays a critical role in optimizing constraint satisfaction solvers**. Future work could explore hybrid approaches, combining multiple heuristics for even greater efficiency. Additional optimizations, such as parallelization or improved constraint pruning, may further enhance performance on more complex instances.

6 References

References

[1] S. Dey, "Solving Sudoku as a Constraint Satisfaction Problem Using Constraint Propagation with Arc-Consistency Checking and Then Backtracking with Minimum Remaining Value Heuristic and Forward Checking," *Sandipan's Web*, March 17, 2017. [Online]. Available: