AI PRINCIPLES & TECHNIQUES

Sudoko

Quince Kersten *S1073935*
Gregors Lasenbergs *S1075747*
Radboud University *November 24, 2023*

1 Introduction

Constraint Satisfaction Problems offer a systematic approach to modeling and solving problems characterized by a set of interconnected variables and constraints. Constraint-satisfaction Problem is search problems with factored representation. They consist of

- `variables`: that can take up values
- `domains`: that hold the values for each variable
- `constraints`: between these variables that have list of variables involved and constraint that holds for these variables

Solution to this problem is complete and consistent assignment of values for each variable

By using richer representations in search algorithms, we can use more advanced techniques like making inferences and drawing logical conclusions. This way we may reduce complexity of the search, or even avoid search at all.

One of the forms of inferencing to simplify search is Arc consistency. In most cases it is proceeded locally, using backtracking search, step-by-step selecting one variable at a time, then selecting a value for that variable and checking if it's legal for both the variable and for the variables that are connected by constraints to this variable. Arc consistency combines the effect of several constraints by constraint propagation.
Arc consistency can also use different type of ordering and value selection heuristics to to speed up the search process, like `Minimum Remaining Value heuristic`, `Degree heuristic`, `Least Constraining Value heuristic`. In this report we will explore which of these heuristics work the best for solving Sudoku puzzle represented formally as a constraint satisfaction problem.

Contents

2 Method

Constraint Satisfaction Problems (CSPs) are a class of computational problems where the goal is to find a solution that satisfies a set of constraints. In a CSP, the problem is defined by a set of variables, each with a domain of possible values, and a set of constraints that specify relationships or restrictions among the variables. The challenge is to assign values to the variables in a way that satisfies all constraints simultaneously. Sudoku puzzle is the perfect puzzle to be represented as a CSP

2.1 Sudoku as CSP

Sudoku puzzle can be modeled as a Constraint Satisfaction Problem (CSP). In a CSP, you have a set of variables, each with a domain of possible values, and a set of constraints that restrict the values those variables can take.

In the context of Sudoku:

Variables: Each cell in the Sudoku grid is a variable. So, for a standard 9x9 Sudoku puzzle, you have 81 variables (9 rows * 9 columns).

Domains: The domain of each variable is the set of possible values it can take. In a standard Sudoku puzzle, the values are the numbers 1 through 9.

Constraints: The constraints in Sudoku are based on the rules of the game:

- Row Constraint: Each row must contain each number exactly once.
- Column Constraint: Each column must contain each number exactly once.
- Block (or Box) Constraint: Each 3x3 block must contain each number exactly once.

The goal is to fill in the grid such that every row, column, and block satisfies these constraints.

2.2 Arc Consistency

Arc consistency is a fundamental concept in constraint satisfaction problems (CSPs), a computational problem-solving paradigm. In a CSP, variables are subject to constraints dictating allowable combinations of values. For every pair of connected variables and their associated constraints, arc consistency ensures that the values in the domains satisfy the constraints. Enforcing arc consistency involves iteratively examining and revising variable domains to eliminate constraint-violating values, ultimately simplifying the problem's solution space.

2.2.1 Design

In Sudoku, variables represent individual cells on the puzzle board, each with a domain of possible values. Constraints are derived from game rules, stating that no two cells in the same row, column, or subgrid can have the same value (neighbors). The arc consistency design for Sudoku focuses on efficiently enforcing these constraints by examining pairs of connected variables or arcs, ensuring their values align with Sudoku rules.

The design of the arc consistency algorithm for Sudoku involves iteratively traversing every cell, reducing their domain based on neighbors. After decreasing a variable's domain, its neighbors with a domain larger than 1 are added to the queue. All variables in the domain are sorted using a heuristic, details of which will be discussed later.

2.2.2 Implementation

The implementation of arc consistency in solving Sudoku requires key functions. The `isValueConsistent` function checks if a given value is consistent with the neighbors of a specific cell. It's output is true or false based on whether the value for variable is consistent with neighbour nodes. While it's input is the variable and value from the domain of this variable.

```
 private boolean isValueConsistent(Field field, int value) {
    for (Field neighbor : field.getNeighbours()) {
        if (neighbor.getValue() == value) {
            return false;
        }
    }
    return true;
}
```

The `revise` function iterates through the domain of a cell, eliminating inconsistent values. It's output is true or false based on whether any inconsistent values were found and removed from the domain of variable. It's input is the current variable.

```
private boolean revise(Field currentField) {
    boolean revised = false;

    for (int value : new ArrayList<>(currentField.getDomain())) {
        if (!isValueConsistent(currentField, value)) {
            currentField.removeFromDomain(value);
            revised = true;
        }
    }

    return revised;
}
```

The `solve` function employs a priority queue to systematically explore and revise cells until a solution is found or an inconsistency is detected, necessitating backtracking.

```
public boolean solve() {
    PriorityQueue<Field> queue = new PriorityQueue<>((a, b) -> a.
        getDomainSize() -b.getDomainSize());

    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (sudoku.getBoard()[i][j].getValue() == 0) {
                queue.add(sudoku.getBoard()[i][j]);
            }
        }
    }
    while (!queue.isEmpty()){
        Field currentField = queue.poll();
        amountOfPolls++;

        if (revise(currentField)) {
            if (currentField.getDomainSize() == 0) {
                // Inconsistent assignment, backtracking needed
                return false;
            }
            for (Field neighbor : currentField.getNeighbours()) {
                if (neighbor.getDomainSize() > 1) {
                    queue.add(neighbor);
                }
            }
        }
    }
    return true;
}
```

2.3 Variable ordering and value selection heuristics

Arc consistency algorithms often rely on queue sorting heuristics to prioritize the order in which variables are processed. The purpose of these heuristics is to enhance the efficiency of the algorithm by guiding the selection of variables during the iterative enforcement of consistency. Sorting heuristics help identify and process variables with a higher likelihood of impacting the constraint satisfaction problem positively.

2.3.1 Design

In our implementation, we explore three different heuristics for variable ordering in the priority queue used in the AC-3 algorithm. The `Minimal Remaining Value` heuristic prioritizes variables with the least remaining values in their domain, aiming to quickly reduce the search space. The `Degree` heuristic, on the other hand, prioritizes variables with a higher number of neighbors, aiming to focus on variables with broader impact. The `No Heuristic` option uses a regular queue with no specific variable ordering, providing a baseline for comparison.

2.3.2 Implementation

To find out which variable choosing heuristics works the best for arc consistency, we made a Priority Queue in which we specified 2 different sorting methods. And we also had a regular Queue with no specific variable choosing heuristic.
For implementing Minimal Value Heuristic you have to prioritize variables that have the least possible values remaining in their domain.

1. Notation ((a, b) -> a.getDomainSize() - b.getDomainSize()) represents a lambda expression for a Comparator.

2. a.getDomainSize() - b.getValue() is the comparison logic. It calculates the difference between the values obtained from calling getDomainSize() on objects a and b.

3. (a, b) are the parameters of the lambda expression, in this context, two fields.

4. The result will be a negative integer if a.getDomainSize() is less than b.getDomainSize(), zero if they are equal, and a positive integer if a.getDomainSize() is greater than b.getDomainSize().

For Degree heuristic we did exactly the same as for Minimal Remaining Value but we used different function to get value to use for Comparator, we used a.getNeighbourAmount().

For no heuristic we just made a reguler queue like this: Queue<Field> queue = new LinkedList<>();

```
//Minimal Remaining Value heuristic
PriorityQueue<Field> queue = new PriorityQueue<>((a, b) -> a.
    getDomainSize() -b.getDomainSize());
//Degree heuristic
PriorityQueue<Field> queue = new PriorityQueue<>((b, a) -> a.
    getNeighbourAmount() - b.getNeighbourAmount());
//No Heuristic
Queue<Field> queue = new LinkedList<>();
```

3 Results

Sudoku puzzle nr	Solved	No heuristic	Degree heuristic	Minimal Remaining value heuristic
		(Poll amount)	(Poll amount)	(Poll amount)
1	YES	1205	1088	820
2	YES	1114	1026	1092
3	NO	829	892	880
4	NO	579	579	579
5	YES	1082	941	981

Table 1: Comparing different amount of polls for Arc consistency algorithm, based on different queue sorting heuristics like Degree heuristics, Minimal Remaining value heuristic and no heuristic. Also showing if solution was found using arc consistency.

3.1 Interpretation of results

The results in Table 1 showcase the performance of the AC-3 algorithm with different variable ordering heuristics. The `Minimal Remaining Value` heuristic consistently demonstrates lower poll amounts compared to both the `Degree` heuristic and the `No Heuristic` approach. This suggests that prioritizing variables with fewer remaining values leads to a more efficient exploration of the solution space.

4 Discussion

The results presented in Table 1 provide valuable insights into the performance of the AC-3 algorithm with different heuristics. The `Minimal Remaining Value` heuristic consistently outperforms both the `Degree` heuristic and the `No Heuristic` approach in terms of the number of polls required for solving Sudoku puzzles. This suggests that prioritizing variables with fewer remaining values in their domains results in a more efficient reduction of the search space.
The `Degree` heuristic, which prioritizes variables with a higher number of neighbors, performs reasonably well but shows slightly higher poll amounts compared to the `Minimal Remaining Value` heuristic. This indicates that focusing on variables with a broader impact may lead to a more extensive exploration of the solution space.
The `No Heuristic` approach serves as a baseline, and its performance falls between the two heuristics. While it lacks the optimization of the other heuristics, it still successfully applies the AC-3 algorithm to solve Sudoku puzzles.
Overall, the discussion highlights the importance of careful variable ordering in constraint satisfaction problems. The choice of heuristic significantly influences the efficiency of the AC-3 algorithm, impacting the number of polls and, consequently, the computational complexity of solving Sudoku puzzles.

5 Conclusion

In conclusion, this study has explored the application of the AC-3 algorithm with different heuristics for variable ordering in solving Sudoku puzzles. The `Minimal Remaining Value` heuristic emerged as the most effective, consistently reducing the number of polls required for successful puzzle resolution.
These findings contribute to a better understanding of constraint satisfaction problems and the role of heuristics in optimizing search algorithms. As future work, further experimentation with additional heuristics and exploring their impact on various constraint satisfaction problems could provide deeper insights into efficient problem-solving strategies.
The successful implementation of the AC-3 algorithm and the comprehensive analysis presented in this report lay the foundation for continued research in the field of artificial intelligence, particularly in the context of constraint satisfaction problems.

6 References

- Kwisthout, J. (2023). Game search [Slide show; Radboud Universiteit Nijmegen]. https://brightspace.ru.nl/d2l/le/content/431290/viewContent/2371537/View