Software Implementation of a CSP Solver

Baris Demirdelen & Maurits Bleeker
University of Amsterdam
E-mail: `barisdemirdelen@gmail.com & maurits-bleeker@hotmail.com`

Abstract. For this research project we implemented a CSP solver. CSP solvers could be used to find a solution for any constrained problem, but for evaluating our CSP solver we will focus on solving regular Sudoku puzzles. First we implemented the basic framework for a CSP solver. Solving a Sudoku with the basic CSP solver (pure depth first search) resulted in an average solving time larger than 500 seconds (measured on 100 Sudoku's) with over a million backtracks per Sudoku. After implementing constraint propagation and some heuristics we finally reduced a average Sudoku solving time to 0,24 seconds on our machine with on average 22,44 backtracks per Sudoku. In this paper we will give an overview of the implemented heuristics and optimizations, and how they contributed in reducing the average solving time and backtracks per Sudoku.

1. Introduction

A constraint satisfaction problem is a problem that is defined by a set of variables and constraints. To solve the constraint satisfaction problem an assignment has to be found for each variable that satisfy all the constraints for the specific problem. Every variable has a domain of possible values, a variable could range over this domain. Constraints are restrictions that restrict the possible values of a domain for a variable. For this project we have implemented an efficient CSP (Constraint Satisfaction Problem) solver, and we will explain the main components of our CSP implementation in this paper. For every component we will give the pseudo-code of our implementation and the contribution of adding that component to reduce the average solving of solving a Sudoku puzzle.

When the constraints of a problem are correctly implemented the CSP solver should always find a possible solution for the problem if one exists. For testing the functionality of our CSP solver we focused on solving Sudoku puzzles. Our model contains 81 variables (one variable for every cell in the Sudoku). Every variable has three constrains: all other variables in the same row should be different, all other variables in the same column should be different, and finally all other variables in the same block (the 3x3 square) should be different. These are the rules for a general Sudoku puzzle. The constraint that can be used to model this problem is the `alldifferent` constraint (e.g. all variables in one row must be different). All the variables that are involved in an `alldifferent` constraint must be pairwise different.

This paper has been divided into 9 chapters. First we will briefly discuss the main framework of the CSP solver. After that we will go into more detail about constraint propagation and optimization heuristics we used for this CSP implementation. Finally, we will discuss some possible extensions and improvements for this CSP solver.

2. Backtracking Search

We implemented a recursive backtracking search algorithm, which then became the main framework of the CSP solver.

Algorithm 1 Backtracking CSP solver

```
procedure RecursiveSearch(assignment)
    constraintPropagation(assignment)
    if isComplete(assignment) then
        return assignment
    variable ← selectUnassignedVariable(assignment)
    orderedValues ← orderDomainValues(variable)
    for value in orderedValues do
        save current assignment and domains
        assignment ← assignment + (variable, value)
        if forwardCheck(variable, assignment) then
            result ← recursiveSearch(assignment)
            if result is an assignment then
                return result
        restore to previous assignment and domains
    return unsatisfiable
```

In Algorithm 1 *recursiveSearch*() is the main recursive function of the CSP solver. It tries to assign values to all variables one by one, if assignment is consistent, it will try to find a value for another variable. If the problem becomes unsatisfiable, it backtracks to previous node. This will continue until the problem is solved or it becomes unsatisfiable.

isComplete() checks if all variables have an assigned value, if this is the case a solution has been found for the problem. We will cover *constraintPropagation*() in Chapter 3, *selectUnassignedVariable*() in Chapter 4, *orderDomainValues*() in Chapter 5 and *forwardCheck*() in Chapter 6.

3. Constraint Propagation

Constraint propagation is the backbone of every CSP solver. We used arc consistency as our local consistency check.

3.1. Maintaining Arc Consistency(MAC)

A model is arc consistent if for every value in a variable's domain, a value can be assigned to any other variable and which satisfies the constraints of those variables as well. We implemented the AC−3 algorithm by Alan Mackworth[1].

Algorithm 2 Maintaining Arc Consistency(MAC)

```
procedure ArcConsistency
    arcs ← generateAllArcs()
    while arcs not empty do
        arc ← arcs.pop()
        if removeInconsistentValues(arc) then
            arcs ← arcs + generateAllArcsLeadingTo(arc.variable2)
```

Algorithm 3 Removing Inconsistent Values

```
procedure RemoveInconsistentValues(arc)
    removed ← false
    for currentValue in arc.variable1.domain do
        if there doesn't exist a consistent value for arc.variable2 then
            arc.variable1.domain ← arc.variable1.domain − currentValue
            removed ← true
    return removed
```

In Algorithm 2, *generateAllArcs()* generates tuples for all variable pairs leading from variable1 to variable2 for all constraints. These tuples are stored in the set *arcs*. Two variables can only end up in a tuple if they are involved in the same constraint. *arcs.pop()* removes one arc tuple from the list of tuples and assigns this tuple to the variable *arc*. *generateAllArcsLeadingTo(arc.varaible2)* generates arc tuples for all variables that are leading to variable2. When you remove a value from the domain of a variable, you have to recheck all the tuples that have that variable as its second variable to check if they are still arc-consistent.

Algorithm 3 removes all values in the domain of variable1 that violates arc-consistency with variable2.

We found out that just by maintaining arc consistency, the problem space reduces dramatically in comparison with pure depth first search, and solving a Sudoku becomes much faster. See Table 1 for the experiment results. We ran all the experiments on a test set of 100 Sudoku's. For every experiment the average solving time and the average amount of backtracks has been measured.

Table 1. Effects of MAC On 100 Sudoku's

Algorithm	Average Solving Time	Average Backtracks
Pure Depth First Search	>500s	>1.000K
MAC	2,95s	555,2

4. Variable Ordering Heuristic

When searching a CSP tree, the solver needs to assign a value to a variable to continue searching. It turns out that choosing the right variable to assign a value can dramatically reduce the search space. We used Minimum Remaining Values and Maximum Degree Heuristics in our CSP solver to achieve this goal.

4.1. Minimum Remaining Values Heuristic(MRV)

The goal of the Minimum Remaining Values Heuristics is to always choose the variable with the least amount of remaining values in its domain. See Algorithm 4 for the pseudo-code of this implementation.

Algorithm 4 Minimum Remaining Values Heuristic(MRV)

```
procedure SelectUnassignedVariable
    unassignedVariables ← unassigned variables
    minVariable ← variable from unassignedVariables with the smallest variable.domain size
    return minVariable
```

MRV is a very cheap heuristic, and for solving Sudoku's, it reduced the size of the problem space to a great extent. See Table 2 for results.

Table 2. Effects of MRV On 100 Sudoku's

Algorithm	Average Solving Time	Average Backtracks
MAC	2,95s	555,2
MAC+MRV	0,92s	139,7

4.2. Maximum Degree Heuristic(MD)

MRV could not choose a variable in cases that the domain size of two variables are the same. For these situations, we used a Maximum Degree Heuristic(MD) as a tiebreaker.

The degree of a variable is the number of unassigned variables that are constrained with our variable. MD goes through all the tied variables in MRV and chooses the one with the maximum degree. Choosing the the variable with the maximum degree, will have influence on the most unassigned variable domains, with will reduce the search space the most, and that is the intuitive reason behind choosing this heuristic.

The degree of a variable is measured the the procedure in Algorithm 5.

Algorithm 5 Calculating Variable Degree

```
procedure GetVariableDegree(variable, unassignedVariables)
    constraintedVariables ← empty set
    for constraint in constraints that contain variable do
        for otherVariable in constraint.variables do
            if unassignedVariables contains otherVariable then
                constraintedVariables ← constraintedVariables + otherVariable
    return constraintedVariables.size
```

Calculating the degree of a variable isn't as cheap as MRV because it needs more calculations to find the degree of a variable, but it still helps immensely. See Table 3 for results.

Table 3. Effects of MD On 100 Sudoku's

Algorithm	Average Solving Time	Average Backtracks
MAC+MRV	0,92s	139,7
MAC+MRV+MD	0,61s	77,82

5. Value Ordering Heuristics

After choosing the right variable to continue searching, it makes sense that choosing 'most promising' value for that variable as well. We used a Least Constraining Value Heuristic to choose our values, with again the goal to reduce the search space of the CSP solver.

5.1. Least Constraining Value Heuristic(LCV)
Least Constraining Value Heuristic chooses the value that constraint other variables less. We calculate how constraining a value is by looking at other variables' domains and calculating how much of their domain would be reduced if we choose this value for our variable. We want to order values by this heuristic to search the most reduced search space first.

Algorithm 6 Least Constraining Value Heuristic(LCV)

```
procedure OrderDomainValues(variable)
    ruleOutCounts ← empty list
    for value in variable.domain do
        ruleOuts ← 0
        for otherVariable in variables in same constraints with our variable do
            ruleOuts ← ruleOuts + getNumberOfRuleOuts(variable,value,otherVariable)
        ruleOutCounts ← ruleOutCounts + ruleOuts
    order variable.domain based on ruleOutCounts
```

In Algorithm 6 we order the domain values based on LCV. *getNumberOfRuleOuts()* returns the count of removed values from otherVariable's domain in case the value is assigned to our variable. We got mixed results with our Sudoku solving tests with LCV heuristic. In some heuristic combinations, it helped. In other combinations it was too expensive to be helpful. See Tables 4 and 5 for the results.

Table 4. Effects of LCV On 100 Sudoku's

Algorithm	Average Solving Time	Average Backtracks
MAC+MRV	0,92s	139,70
MAC+MRV+LCV	0,81s	116,01

Table 5. Effects of LCV On 100 Sudoku's Cont.

Algorithm	Average Solving Time	Average Backtracks
MAC+MRV+MD	0,61s	77,82
MAC+MRV+MD+LCV	0,63s	81,68

6. Look Ahead Techniques
Look ahead heuristics is a generic term describing heuristics that tries to evaluate the effects of a branching. We used Forward Checking as our look ahead technique, to evaluate what the consequences will be of the branching step.

6.1. Forward Checking(FC)

Forward checking is evaluating the effect of a specific assignment to a variable. Given the current partial solution and a candidate assignment to evaluate, it checks whether another variable can take a consistent value.

Algorithm 7 Forward Checking(FC)

```
procedure ForwardCheck(variable, assignment)
    nextCheck ← a set with one element only
    while nextCheck not empty do
        currentVariable ← nextCheck.pop()
        for constraint in constraints containing currentVariable do
            consistent ← constraint.ruleOut(currentVariable, assignment)
            if not consistent then
                return false
            deducedVariables ← all unassigned variables with one value remaining in their domain
            assignment ← assignment + deducedVariables
            nextCheck ← nextCheck + deducedVariables
    return true
```

In Algorithm 7 *constraint.ruleOut(currentVariable, assignment)* rules out values from its variable's domain, given assignment to *currentVariable*. If a variable's domain size becomes zero it is inconsistent and algorithm returns false. If a variable's domain size becomes one, algorithm marks it as a deduced variable, adds it and its values to assignment and queues it to be checked next. This means we only go another level of forward checking if there is a deduced variable. See Table 6 for the results.

Table 6. Effects of FC On 100 Sudoku's

Algorithm	Average Solving Time	Average Backtracks
MAC+MRV+MD+LCV	0,63s	81,68
MAC+MRV+MD+LCV+FC	0,40s	39,41

7. Constraint Specific Propagation

Some constraints have special properties that enables them to be propagated easier or faster than other constraints. We implemented the ability for custom constraints to have specific propagation algorithms. We implemented such a specific propagation technique for `alldifferent` constraint that we use to solve Sudoku's.

7.1. AllDifferent Specific Propagation

For `alldifferent` constraints, if the size of distinct values in its variables' domains is less then the size of its variables, we can say that the `alldifferent` constraint is unsatisfiable. This is very easy to implement and in problems that heavily use `alldifferent` constraints such as Sudoku's it makes a huge difference. See Table 7 for the results.

Table 7. Effects of AllDiff-Specific On 100 Sudoku's

Algorithm	Average Solving Time	Average Backtracks
MAC+MRV+MD+LCV+FC	0,40s	39,41
MAC+MRV+MD+LCV+FC+AllDif-Specific	0,24s	22,44

8. Discussion

We have managed to improve the CSP solver quite a bit, but still further improvements can be made. Algorithms such as backjumping with dependency graphs or a branch and bound algorithm could be implemented for further reduction in problem space. Also we haven't tackled local consistency checks other than arc consistency, although we saw that arc-consistency had a huge impact on the performance (both for solving time and backtracking count). Path consistency or a general k-consistency algorithm could be implemented, but these are expensive algorithms, so the efficiency of the solver should be inspected.

Moreover, more optimizations could be made from the software standpoint. We went for a more readable and human understandable approach in our program, although a more optimized software could perform better. There may also be general optimizations to be made like reducing nested for loops, using clever hash maps, etc.

Another point for further research is the order of the evaluating the effects on the solving time of a heuristic or an optimization technique. Due to time limitations, we could not make any statement about which combination of heuristics was the most effective on average, or which combinations of heuristics performed better for different kinds of Sudoku's.

9. Conclusion

In conclusion, we built an efficient and stable CSP solver with common heuristics and constraint propagation. With our iterative implementations and optimizations, we managed to reduce problem space dramatically. For solving Sudoku's we have improved the solver from more than 500 seconds solving time and more than a million backtracks on average, to 240 milliseconds solving time on our machine with on average 22,44 backtracks. This could still be further improved by extra optimization techniques and software improvement, but for this research project we think that we have produced a reliable and efficient CSP solver.

Appendix

A. Test Machine Specifications

- Intel Core i7-5500U CPU @ 2.40GHz
- 12GB RAM @ 1600MHz
- Windows 10 64−bit Operating System

References

[1] A.K. Mackworth. Consistency in networks of relations. Artificial Intelligence, 8:99-118, 1977.