Automated Reasoning in AI
Assignment 2: Sudoku as a CSP

Armon Toubman Torec Luik

May 29, 2011

1 Introduction

The popular Sudoku puzzle can be seen as a Constraint Satisfaction Problem (CSP). For this assignment, we will implement a CSP solver for solving Sudoku puzzles, including optimizations specifically for Sudokus. ***********Section X gives a short introduction to Sudokus as CSPs. Section X describes the solving algorithm we used. The optimizations we used are explained in section X. In section X we compare the effectiveness of the optimizations. Finally, section X concludes this report.*************

Hier een Sudoku template indien nodig:

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

.94...13..............76..2.8..1.....32.........2...6.....5.4.......8..7..63.4..8

2 Sudoku as a Constraint Satisfaction Problem

A completed (9x9) Sudoku puzzle has the numbers 1 to 9 appearing exactly once in every row, column and region. We can make the following translation Sudoku. In other words: cells we are not sure about.

Assignment: Cells with assigned values.

Constraints: As just stated, the numbers 1 to 9 appearing exactly once in every row, column and region.

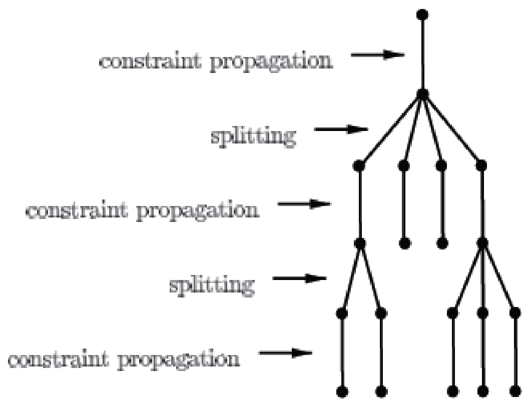

Figure 1: *Example of a CSP search tree. From Constraint Programming - lecture 3 (CSP) Algorithms.*

Domains of the variables: At the start, the possible values of not-assigned cells are 1 to 9.

Consistency: An assignment is consistent if the same value does not appear more than once per row, column and region.

Termination: A solution is complete if there are no variables (empty cells) left. Provided the initial problem is valid, the solving process should always terminate succesfully.

3 Search Algorithm

When we consider the Sudoku as a CSP with a sequence of variables, we can create a search tree for this CSP. Here the nodes of the tree are CSP's and the root is our starting CSP (i.e. the initial Sudoku). See also Figure 1. First constraint propagation is applied to the current CSP. Then, if the resulting CSP is not a leaf, one of its variables is split up. For every value in its domain, a new child will be added to the tree. This process contintues until the leaf is a correct solution (i.e. all variables are instantiated and adhere to the constraints).

3.1 The Backtracking Algorithm

For creation and traversal of our Search Tree, we started with the backtracking algorithm as described in the assignment description.

```
procedure BT(X:variables, V:assignment, C:constraints)
    if X={} then return V
    x := select a not-yet assigned variable from X
    for each value h from the domain of x do
```

```
        if consistent(V+{x/h}, C) then
            R := BT(X-{x}, V+{x/h}, C)
        if R != fail then return R
    end for
    return fail
end BT
```

In short, this algorithm takes a cell that is not yet assigned a value, and for each value in its domain creates a new branch in the search tree. The branches recursively create their own branches, forming a search tree. The tree is searched depth-first for a solution. The algorithm terminates if either a solution is found, or no solution exists in the search tree.

3.2 Optimizations

The Backtracking algorithm is, besides simple and effective, very inefficient. For each open cell in a Sudoku problem, the algorithm will create nine branches (one for each possible value). To solve this, we can use constraint propagation to eliminate a lot of these branches at an early stage. There are also specific Sudoku solving techniques that help rule out possible values from certain variables' domains. Pruning the search tree with these methods way results in a smaller search space and therefore lower runtimes of the solver. In this section we describe the pruning methods we used, which correspond to the earlier-discussed constraint propagation.

3.2.1 Constraint Propagation : Revise

Our Revise method is a version of the AC-3 method as described in Barták [1998]. The AC-3 (AC: Arc Consistency) method checks for arc-consistency, i.e. constraints between multiple variables. The domain of chosen variables is reduced to those values that are still possible in the current solution. It works as follows: When a value is assigned to a cell, because in a completed Sudoku each value should appear exactly once in each row, column and region, we can rule out the assigned value as a possibility for the other cells in its row, column and region. It is possible that this method implicitly creates a new assignment by ruling out a value from the domain of a cell with only two values left. Because of this, we apply the Revise method until no more changes are made. For efficiency, after the first revision, the Revise method only checks cells that could have been affected by the previous revision.

This method is capable of solving a Sudoku by itself without ever allowing the backtracking algorithm to branch, as we experienced with multiple Sudokus in the provided testset. In other sources, constraint propagation in the form of arc consistency is called MAC (Maintaining Arc Consistency). Also: maintaining arc consistency (MAC) . Now: local consistency notion is arcconsistency. . Arc consistency binary constraints: For every constraint C and every variable x with domain D, each value for x from D participates in a solution to C . Arc consistency: En revise bekijkt ook arc consistency (i.e. constraints tussen 2 variables)

3.2.2 Hidden Singles

3.2.3 Naked Pairs

3.2.4 Hidden Pairs

3.2.5 Heuristics

From slides: Complete labeling tree

ordering of variables does not differ in number of leaves

ordering of variables does differ in number of nodes

least number of nodes if the variables are ordered by their domain sizes in the increasing order

This is a bit like one of the heuristics with the lowest nr Children first stuff.

And later on: Heuristics for search algorithms

Variable selection

Select a variable with the smallest domain

select a most constraint variable

select a variable with the smallest difference between its domain bounds

Heuristics for search algorithms

Value selection

select a value for which the heuristic function yields the hightest outcome

select the smallest value

select the largest value

select the middle value

4 Implementation

Java blabla

5 Results

Zoiets:

6 Conclusion

References

Roman Barták. On-line guide to constraint programming, June 1998. URL `http://kti.mff.cuni.cz/~bartak/constraints/index.html`.

Revise	Hidden Singles	Hidden Pairs	Naked Pairs	sudoku_training	top95
x				8m56s	1h55m25s
x	x	x	x	25s	10s
x	x			29s	47s
x	x		x	23s	21s
x	x	x		28s	16s
x		x		8m9s	26m37s
x		x	x	1m21s	3m51s
x			x	1m17s	19m32s

Table 1: *The runtimes of our program using two different Sudoku sets with the different optimizations enabled (indicated with an x) or disabled. The times were averaged over three runs, except the revise-only trial which was only performed once.*

Revise	Heuristic 1	Heuristic 3	sudoku_training	top95
x			8m56s	1h55m25s
x	x		8m53s	1h57m3s
x		x	9m3s	1h57m40s
x	x	x	9m1s	1h56m52s

Table 2: *The runtimes of our program using two different Sudoku sets with only the revise algorithm and no, one or both heuristics enabled. These results are from a single run.*

Revise	Hidden Singles	Heuristic 1	Heuristic 3	sudoku_training	top95
x	x			29s	47s
x	x	x		29s	50s
x	x		x	30s	48s
x	x	x	x	29s	52s

Table 3: *The runtimes of our program using two different Sudoku sets with only the revise algorithm and the hidden singles algorithm and no, one or both heuristics enabled. The times were averaged over three runs.*