Patryk Stroński 242564

Lab 2 – part 1 – SUDOKU

Introduction

Sudoku is a simple game about filling numbers into 9x9 grid. In order to solve sudoku, I used 2 approaches: **backtracking with domain calculated on the fly** and **backtracking with forward checking.** Moreover I used Rust programming language. Both of the methods are effective enough for this problem but it rendered that backtracking with domain-on-the-fly calculation won the duel.

CSP definition

Each field in Sudoku is a variable. Therefore the CSP is 81th dimensional one.
Each variable stands for a field in a sudoku 9x9 grid. We have V1 … V 81.
Each variable has domain which in the beginning is:
D1 … D81 ⊂ {1,2,3,4,5,6,7,8,9}
The initial state of each variable is:
V1 … V81 ⊂ {-1,1,2,3,4,5,6,7,8,9}
The unfilled variables have -1 from the beginning, whereas the filled ones have a number.
There are three rules. Based on them three constraint schemes can be defined.
1. Each row needs to have all values appearing only once
P1: V(x + (y-1) * 9) ⊂ [1,2,3,4,5,6,7,8,9] && V(x + (y-1) * 9) != V(x' + (y-1) * 9). && x' != x && x ⊂ [1,9], x' ⊂ [1,9], y ⊂ [1,9]
x – position on horizontal axis
y – position on vertical axis
x' – position on horizontal axis
V(k) – variable of index k

2. Each column needs to have all values appearing only once
P2: V(x + (y-1) * 9) ⊂ [1,2,3,4,5,6,7,8,9] && V(x + (y-1) * 9) != V(x + (y'-1) * 9). && y' != y
x ⊂ [1,9], x' ⊂ [1,9], y ⊂ [1,9]
x – position on horizontal axis
y – position on vertical axis
y' – position on vertical axis
V(k) – variable of index k

3. Each square needs to have all values appearing only once
Square indices:
s = x/3 * 3 + y/3 * 3 * 9
i = { s, s+ 1, s + 2, s + 9, s + 10, s + 11, s + 18, s + 19, s + 20 }
P3: V(i) ⊂ [1,2,3,4,5,6,7,8,9] && V(i) != V(i'). && i' != i && s == s'
x ⊂ [1,9], x' ⊂ [1,9], y ⊂ [1,9]
s – square starting index (left upper corner)
i – index calculated from s.
x – position on horizontal axis
y – position on vertical axis
i' – another index calculated from s (the same s)
V(k) – variable of index k

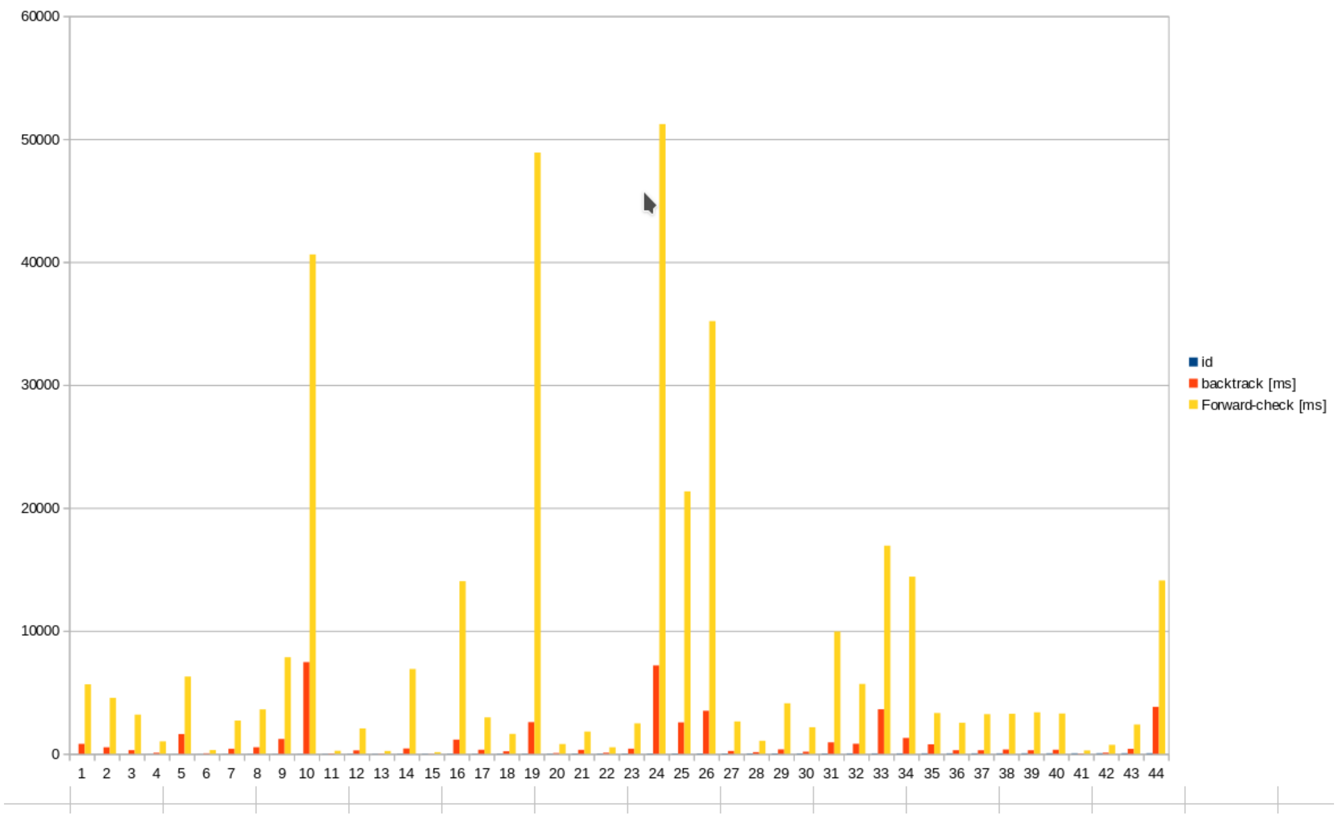
Domain on the fly

This approach I used to get rid of as many computations as possible. Also, the memory is saved in this case, as I do not store the vector with domains anywhere, it is calculated on the fly. Every time before the field is evaluated, the domain for this field is calculated based on its coordinates and other fields. The result of this evaluation is a basis for next evaluations. This has multiple advantages, especially in my implementation: the checks are only narrowed to the fields involved (square, row, column) and no other operations than fetching the current domain are done. Thanks to it, memory and time is saved.

Forward checking

Here more computations appear, therefore it happened to be more time consuming than the first option. The domains are precalculated each time a number is inserted to the table. This approach is problematic because each domain calculation takes time as the square, row and column for each place must be taken into account. The advantage is that when the program notices that for some value the domain of some other forthcoming field is empty, the algorithm doesn’t go deeper but immediately nullifies the field and checks next value.
The forward checking I implemented also is a simple one. If the domain computations were spared, the values would differ.

Results:



Conclusion

Based on the following diagram we can simply notice, that the backtracking with heuristic approach did the job far better than forward check. This is because of number of operation that were needed in each operation. Forward checking was calculating domain for each member every time the member

was changed, whereas backtracking was calculating domain on the fly, getting information based on 21 members amongst 81 (9 + 8 +4). Also the calculation was done when needed which means no vector was holding the values.
On the other hand, the results could have been much better if the forward check was profiled more – the domain updates were restricted only to column, square and row variables.