# Exploring Performance Tradeoffs in a Sudoku SAT Solver

**CS242 Project Report**

Hana Lee (leehana@stanford.edu)

December 15, 2017

## 1 Summary

I implemented a SAT solver capable of solving Sudoku puzzles using constraint programming in all three of the languages we've discussed in CS242: Lua, Ocaml, and Rust. I analyzed the performance of the same algorithm implemented across the different languages, comparing both average runtime across a set of 50 "easy" Sudoku puzzles and a set of 10 "hard" puzzles. The major concept I explored for this project was the tradeoff between expressiveness and performance; while Lua is a more expressive language and easier to code, due to my Python background, Rust can match the speed of very performant languages like C/C++. I performed a holistic evaluation of all three languages, taking into account both expressiveness and performance, in an effort to pick the "best" language of the three for the average coder in everyday use.

## 2 Background

Sudoku is a logic-based, combinatorial number-placement puzzle. A puzzle usually consists of 81 cells laid out in a 9x9 grid, with the grid split into 3 rows and 3 columns to create 9 sections of 9 cells each. Numbers from 1-9 must be placed into every cell such that the same number occurs exactly once in every column, row, and section. The puzzle begins with certain cells filled in with values that serve as constraints, narrowing down the puzzle to a single solution; a Sudoku is only considered valid if one and only one solution exists.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 3 | | 2 | | 6 | | |
| 9 | | | 3 | | 5 | | | 1 |
| | | 1 | 8 | | 6 | 4 | | |
| | | 8 | 1 | | 2 | 9 | | |
| 7 | | | | | | | | 8 |
| | | 6 | 7 | | 8 | 2 | | |
| | | 2 | 6 | | 9 | 5 | | |
| 8 | | | 2 | | 3 | | | 9 |
| | | 5 | | 1 | | 3 | | |

Figure 1: The first example from the easy50 set of Sudoku puzzles.

| 1 | | 2 | |
|---|---|---|---|
| 2 | | 4 | |
| | | 1 | 4 |
| 4 | 1 | 3 | |

Figure 2: A 4x4 Sudoku initialized with a partial solution.

There are several known algorithmic techniques for solving Sudoku using a computer. The simplest and slowest method is to use backtracking, a form of depth-first search that exhaustively tries each possible value for each empty cell, backtracking if the new placement violates any constraints[1].

Because the backtracking algorithm is not overly difficult to implement but is computationally demanding, it makes a good choice of algorithm for a comparison of programming difficulty and computational efficiency between the three languages we have studied in CS242: Lua, OCaml, and Rust. While Lua is an example of an expressive language that is fairly easy to pick up for a novice programmer, OCaml and Rust are compiled down to machine code for high computational efficiency.

## 3 Approach

Constraint programming when applied to a Sudoku puzzle is essentially a backtracking algorithm. The constraints are expressed in conjunctive normal form, where one clause exists for each cell that has not yet been assigned a number value. The clause contains literals representing each possible number that could be assigned to the cell, not including numbers which are already present in the same row, column, or sector of the grid. For example, consider the 4x4 Sudoku puzzle in Fig. 2. The constraints for this puzzle as a CNF are:

$$(a_2 = 3 \vee a_2 = 4) \wedge (a_4 = 3) \wedge (b_2 = 3) \wedge (b_4 = 1 \vee b_4 = 3) \wedge (c_1 = 3) \wedge (c_2 = 2 \vee c_2 = 3) \vee (d_4 = 2)$$

To solve a Sudoku puzzle using the SAT solver from Assignment 5, I converted 50 "easy"-rated puzzles into CNFs and began with a partial assignment based on the initial values of the cells. In order to avoid running into contradictions while solving the puzzle, I needed to add an additional step to the SAT solver. After picking a variable assignment that satisfies a clause, the solver must check all of the other clauses to see if they contain a literal which would conflict with the new assignment. If they do, the literal must be removed from the clause. If any clauses become empty this way, we have made an invalid assignment.

For example, if the solver chose to satisfy the first clause in the CNF for Figure 2 by choosing the assignment $a_2 = 3$, it would then check the rest of the clauses for conflicts. It turns out that since $a_4$ is in the same row as $a_2$, the value 3 must be removed from the list of possibilities for that cell. However, this causes the clause for $a_4$ to become empty, which means there is no number that can be assigned to that cell without causing conflict, and the algorithm must backtrack.

## 4 Implementation

I implemented the solver described in the previous section in each of the three languages: Lua, OCaml, and Rust. The OCaml and Lua implementations were built from a heavily modified version of my Assignment 5 code, while the Rust implementation was coded from scratch. I compared the time spent coding up each algorithm and the pitfalls I ran into

| | Lua | OCaml | Rust |
|---|---|---|---|
| Hours | 3.5 | 4 | 3 |
| Difficulty | 5 | 7 | 4 |

Figure 3: Hours spent and "programming difficulty" score out of 10.

along the way in order to give each language a "difficulty score" from 1-10, where 1 is coding "Hello world!" in Python and 10 is coding a SAT solver in x86 assembly.

Before I began, I expected the Lua implementation to be the easiest to code, due to Lua's similarity to Python, its general expressiveness as a language, and the relative ease of the Lua programming projects during this course compared to the OCaml and Rust projects. I expected Rust to be the most efficient implementation and the second easiest to code behind Lua, and I expected OCaml to be very difficult to code and performance-wise somewhere in the middle between Lua and Rust.

### 4.1 Lua

I ran into more difficulty with the Lua implementation than I had anticipated I would. It turned out that my familiarity with Python may have hindered my progress as much or more than it helped. Lua does not support automatic integer division or indexing into strings like Python does, and it uses one-indexing for lists instead of zero-indexing. These non-intuitive features of the language introduced bugs into my program that took a lot of time to track down and fix.

For a novice programmer without much exposure to features that I tend to take for granted, like zero-indexing and automatic integer division, these quirks of Lua may not affect their ease of use at all. However, the amount of frustration I experienced during what I expected to be a relatively straightforward modification of my existing SAT solver code was enough to rate the Lua algorithm 5/10 programming difficulty.

### 4.2 OCaml

OCaml, unlike Lua, is very dissimilar to any language I have worked with before. I expected the OCaml implementation of the SAT solver to be very tricky, especially since I would have to implement it using recursion and functional programming rather than iteration and imperative programming as I did for the other two languages.

The hardest part of coding up the algorithm in OCaml was actually not the algorithm itself; it was processing the input files and converting the puzzles into CNF form. I was able to reuse a large portion of the SAT solver code provided in the starter code for Assignment 5, and making the modifications described in the Approach section went fairly smoothly.

Overall, planning out my approach to the algorithm using recursion and list comprehension rather than iteration took the majority of the time allocated to this implementation. Given the difficulty I had wrapping my head around the details of the implementation, even though I did not run into any significant bugs, I rated the OCaml algorithm 7/10 programming difficulty.

### 4.3 Rust

Unlike the Lua and OCaml implementations, I did not have an existing SAT solver implementation for Rust. With this in mind, and given my preference for expressive, dynamically typed languages like Python over languages with stricter syntax that are

| | Lua | LuaJIT | OCaml | Rust |
|---|---|---|---|---|
| No heuristic | 2.18 | 0.42 | 0.18 | 0.42 |
| MRV heuristic | 0.08 | 0.02 | 0.006 | 0.03 |
| Algorithm fix | – | – | 0.0006 | 0.0009 |

Figure 4: Average runtime in seconds per puzzle, across 50 easy puzzles.

| | Lua | LuaJIT | OCaml | Rust |
|---|---|---|---|---|
| No heuristic | 199.6 | 41.8 | 12.5 | 97.7 |
| MRV heuristic | 11.7 | 2.4 | 0.9 | 9.1 |
| Algorithm fix | – | – | 0.06 | 0.1 |

Figure 5: Average runtime in seconds per puzzle, across 10 hard puzzles.

meant for systems programming like C++ and Rust, I was expecting to spend a large chunk of time wrestling with the Rust implementation.

I was pleasantly surprised to find that the Rust implementation was straightforward. Installing the Rust package for Sublime made coding significantly easier because I was able to catch errors and fix them before even compiling the code, thanks to the error highlighting and inline typechecking. While Rust does support some level of dynamic typing (or at least is able to infer types without having them explicitly assigned), I found it made life easier to simply include the types for each variable as they were initialized.

I found Rust to be nearly as expressive as Lua and more in line with my programming assumptions (like zero-indexing and integer division). It also took less time to implement a full solution from scratch than it did to adapt an existing Lua solution to the new problem. I rated the Rust algorithm 4/10 programming difficulty.

## 5 Performance

I compared the average runtime of the algorithm per puzzle in the set of 50 easy puzzles. During implementation, I made very little effort to directly optimize, since the goal of the project is to explore the tradeoff between expressiveness and performance for "everyday use" of the languages under consideration. I relied on the language compilers and other mechanisms "under the hood" to do the optimization for me, in order to get an estimate of how efficient each language would be for an average user.

The only direct concession to performance I made was to test the runtime of the Lua implementation using both the normal compiler and the LuaJIT compiler (a high speed interpreter and compiler for the Lua language). My reasoning for benchmarking the LuaJIT implementation was that LuaJIT is an easily accessible performance improvement for the average Lua user, given that it can be downloaded, installed, and used to run Lua programs as normal without much additional effort.

The results of the initial performance comparison were surprising. While the vanilla Lua program was the slowest by a factor of 4 as expected, running the program with LuaJIT increased its performance to be on par with the Rust implementation. Both implementations took 0.42 seconds on average to solve a single "easy"-rated Sudoku puzzle. However, in a surprising turn of events, the OCaml implementation blew both of them out of the water with an average solve time of 0.18 seconds. Given all of the praise I had heard about Rust's computational efficiency and how little I had heard about OCaml, I was surprised to find Rust so easily outmatched.

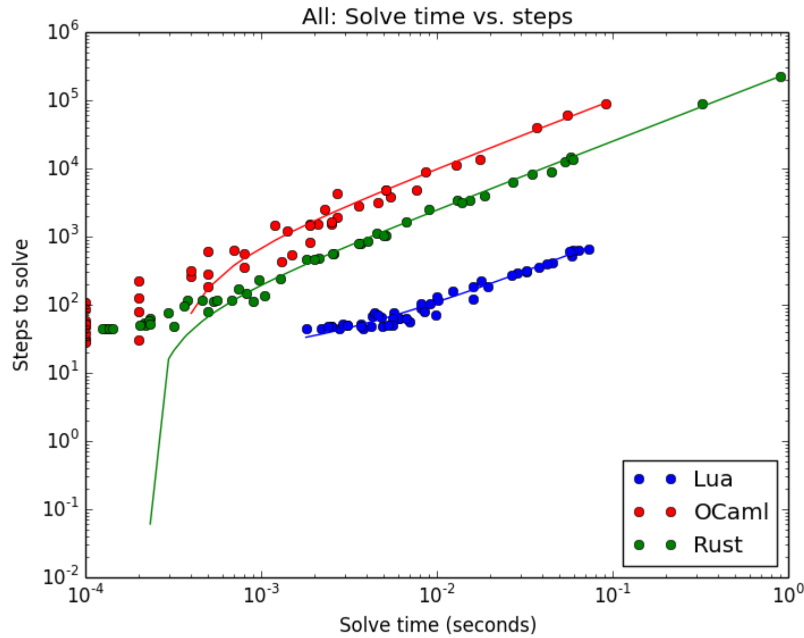


Figure 6: Solve time in seconds plotted against number of steps for the 50 easy puzzles.

### 5.1 Harder puzzles

I did a followup performance comparison by running the solvers on a set of 10 "hard" puzzles. On hard puzzles with only a few values provided to start with, the backtracking algorithm is very slow. The runtime ratio between the LuaJIT and OCaml implementations remained constant, while the Rust implementation slowed down considerably, to more than two times as slow as the LuaJIT implementation and almost ten times as slow as the OCaml implementation.

### 5.2 MRV heuristic

One simple tweak to the backtracking algorithm that is known to lower solve time is the "minimum remaining values" heuristic[2]. When this heuristic is used, the next clause to receive a satisfying assignment is not simply the first clause in the CNF, but rather the clause containing the fewest literals out of all the clauses in the CNF.

This heuristic was easy to implement in the Lua solver, and lowered the average solve time for the set of 50 easy puzzles to 0.07 seconds. Likewise, adding the heuristic to the Rust solver took a very short amount of time and lowered the average solve time to 0.03 seconds.

Adding the heuristic to the OCaml solver was more complex and took a significant amount of programming time (about 30 minutes), but lowered the average solve time to 0.006 seconds. OCaml therefore remains the speediest implementation, although the gap narrows when even a simple heuristic is introduced.

### 5.3 Steps vs. time

In an effort to get to the bottom of why the Rust implementation was so much slower than the Lua and OCaml implementations, I plotted the number of steps needed to solve each puzzle against the time taken to solve that puzzle. A "step" is defined in the backtracking

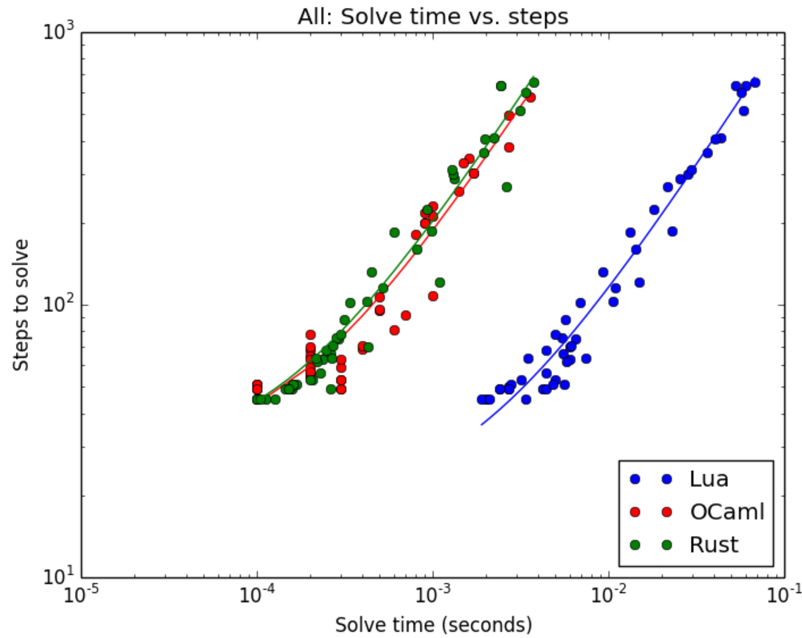


Figure 7: Final solve time in seconds plotted against number of steps for the 50 easy puzzles, after the algorithm fixes in the OCaml and Rust implementations.

algorithm as an assignment of a value to a cell; the algorithm often must attempt to assign several different values to a cell before finding an assignment that satisfies all constraints. This plot is shown in Figure 6.

From the plot, it becomes clear that despite my attempts to implement the backtracking algorithm in the exact same way across the three languages, differences in implementation arose regardless. If the implementation were exactly the same in each program, the number of steps to completion would be constant for each puzzle; however, we can see that the OCaml implementation generally takes the most steps, while Lua takes the fewest. This happens because the Lua algorithm terminates as soon as it finds a valid assignment, while the OCaml implementations continue to iterate over all possible values for each clause. This discrepancy is due to my unfamiliarity with OCaml and the difficulty I had with "terminating early" from a list comprehension-based implementation.

Once I was able to tweak the algorithm in the OCaml implementation to behave the same way as the Lua implementation, the OCaml average solve time for the 50 easy puzzles dropped even lower, to 0.0006 seconds. Even more impressive, when tested on the set of 10 hard puzzles, the OCaml average solve time dropped to an incredible 0.06 seconds - 40 times faster than the Lua implementation.

One mystery remained; the Rust algorithm appeared to take more steps per puzzle than the Lua algorithm, despite being implemented with early termination just like the Lua algorithm. Upon investigation, I discovered that a bug in the Rust algorithm caused it to fail to eliminate all conflicting value candidates at each step in the backtracking algorithm. Once I fixed the bug, the new plot (Figure 7) revealed that the OCaml and Rust algorithms were actually neck-and-neck in computational speed. Small differences in the average solve time remained - OCaml was still about 1.5 times as fast as Rust, on both easy and hard puzzles - but on any sets other than very large ones, this difference is negligible.

### 5.4 OCaml vs. Rust

Why is the OCaml implementation able to beat the Rust implementation, even slightly? Both languages are compiled down to machine code before execution, so the answer could be in the level of optimization applied by each compiler. It could be a difference in implementation; the Rust algorithm performs expensive clones of the CNF and current assignment before taking each step, while the OCaml algorithm builds a new CNF by list comprehension. Because memory management is abstracted away in OCaml, it's hard to tell how values are passed around and when copies are made. Memory management could be the answer to the question of why OCaml's performance seems better than Rust's.

## 6 Conclusion

The overall purpose of this project was to evaluate the usefulness of the three languages we studied in this class for an application where both expressiveness and performance are important. As a programmer, I gravitate towards forgiving, dynamically typed languages like Python and JavaScript, yet these languages are not known for their performance or type safety. This project was, in part, an experiment to see whether my preferences would still hold true when performance is a legitimate concern.

After observing the performance of all three algorithms, I have concluded that OCaml, a language I formerly harbored deep resentment for due to its extreme dissimilarity from my languages of choice, is actually quite useful in the right circumstances. With the simplest algorithmic implementation and without any effort at optimization on my part, it still outstrips the other two languages in computational efficiency. The development process was also not nearly as painful as my first attempts at programming in OCaml for this class.

After this project, I may even use OCaml for future projects, a possibility I would never have considered without observing the results of my experiment. On the right project, a couple of extra hours of programming and debugging can be a very fair tradeoff for incredible gains in efficiency.

## 7 References


[1] Simonis, Helmut (2005). "Sudoku as a Constraint Problem". Cork Constraint Computation Centre at University College Cork: Helmut Simonis. Retrieved 14 December 2017.

[2] Norvig, Peter. "Solving Every Sudoku Puzzle". Peter Norvig (personal website). Retrieved 14 December 2017.