Learning Computational Complexity

Graphical Abstract

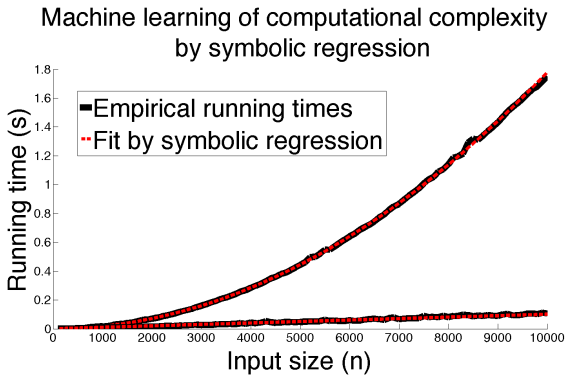

Figure 1. Instead of constraining ourselves to theoretical analysis of algorithms, we can use tools in machine learning to determine their computational complexity. In particular, we make the radical proposal to determine the functional form using *symbolic regression*, and use that as the complexity. For example, consider running two sorting algorithms, insertion sort (slower) and quicksort (faster). The black lines are the actual running times. Clearly, they look like promising functions to fit. Indeed, symbolic regression packages can find extremely good fits, as shown by the red lines. In our proposed framework, the complexity of insertion sort is: $t = 1.7781375 \times 10^{-8}\ n^2$, whereas for quicksort it is: $t = 1.0103297254990829295448512038832 \times 10^{-5}\ n$.

1. Introduction

Recall learning about computational complexity, and the first time encountering the definition for Big-O notation:

Definition 1. *Let $f, g : \mathbb{R} \to \mathbb{R}$ be arbitrary functions. If there exists a constant c and real number x_0 such that for all $x \geq x_0$, $|f(x)| \leq c\,|g(x)|$, then we denote $f(x) = O(g(x))$.*

Copyright © ACM [to be supplied]. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

For most students, this definition is challenging to parse! When we consider what exactly this definition is trying to capture, we see that we are simply interested in the growth rate of a function. For convenience, we will consider this function to be the running time of an algorithm for the remainder of this paper. Instead of trying to confuse and scare away students using Big-O, we can communicate the same underlying point by simply having them plot the sequence of running times t encountered as the input size n grows large. Using the unreasonably-effective tools of modern machine learning, in particular that of *symbolic regression*, we can find the curve of best fit for the running times, and use the resulting parametric functional form to infer the computational complexity of an algorithm from its running times.

In fact, we are so confident in the spectacular capabilities of machine learning that we will take a leap of faith and define the computational complexity of an algorithm to be the functional form found through the above procedure.

Figure 1 shows an example of this concept. See the figure captions for details. Note that it has indeed confirmed that insertion sort has running time that grows quadratically as the input size. As for quicksort, we see that our method has now achieved the once-impossible *linear-time sorting* (with only pair-wise element comparisons and no additional assumed structure). The astute reader who was educated in the conventional system may raise an eyebrow at this result, since it is commonly known that the average-case complexity of quicksort is $O(n \log n)$; however, in light of the wisdom of Stephen Boyd, who once said something similar to "$\log n$ is basically ≤ 50, and $\log \log n$ is basically ≤ 6" [personal communication], we see that there is some degree of truth to linear-time quicksort.

2. Potential Applications

See Figure 2 for a theory-related example. See the figure caption for details. See Figure 3 for a systems-related example. See the figure caption for details.

3. Conclusion

Due to lack of space[1], we have only been able to scratch the surface of this exciting new paradigm. We are certain that there will be much citation-producing follow-up work, in all fields of theory, systems, and artificial intelligence.

[1] For salami-slicing convenience, we interpret the 11-page limit in unary.

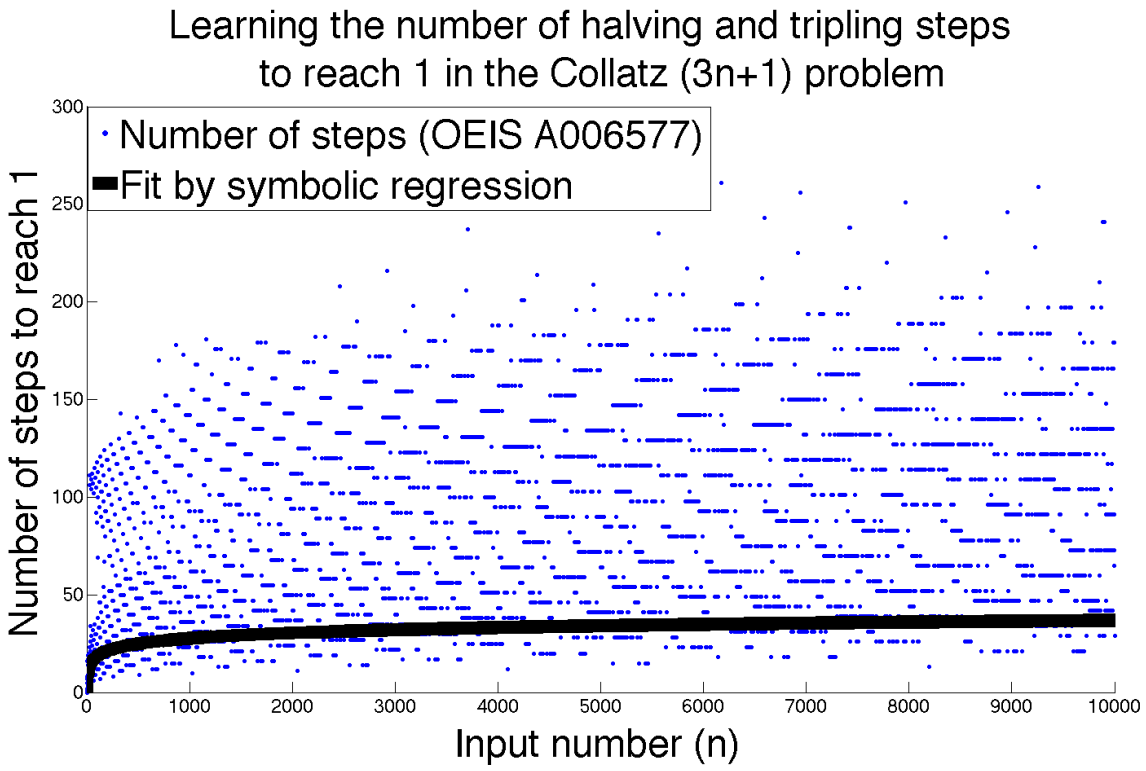

Figure 2. See the text in Section 2 for details. Number of steps $= \log(n^4) = 2\log(n^2) = 4\log n$.

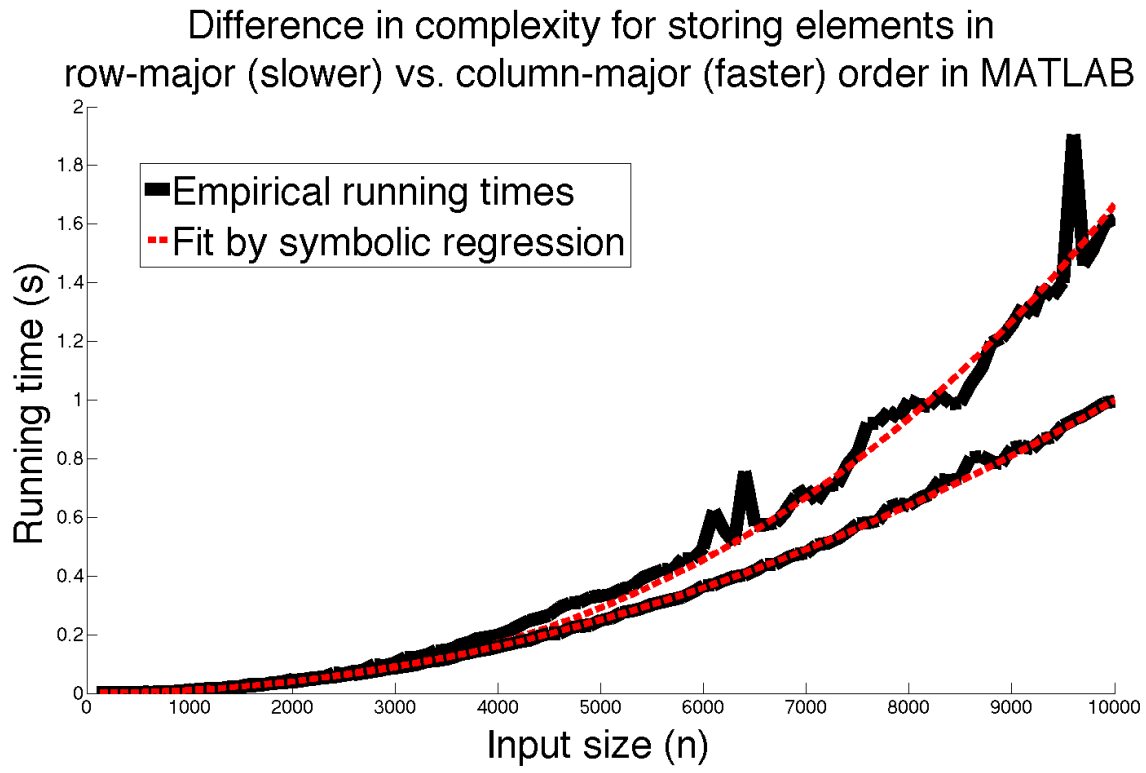

Figure 3. Hitting/missing the L1 cache causes differences in complexity classes. MATLAB stores arrays in column-major order, so accessing them in that order is faster. In column-major order, the complexity found is: $t = 10^{-8}\ n^2$, whereas in row-major order, the complexity found is: $t = 10^{-12}\ n^3 + 6.652 \times 10^{-7}\ n^2$. See the text in Section 2 for details.