BiGO: A Toolset to Support CS Students to Learn to Analyze Time Complexities of Algorithms

Toivonen, Tapani

ACM Press
BiGO: A Toolset to Support CS Students to Learn to Analyze Time Complexities of Algorithms

Tapani Toivonen‡
University of Eastern Finland
Joensuu, Finland
tapani.toivonen@uef.fi

Solomon Oyelere
School of Computing
University of Eastern Finland
Joensuu, Finland
solomon.oyelere@uef.fi

ABSTRACT

Analysis of algorithms is a part of computing curriculums throughout the world in both undergraduate and graduate levels. However, students in general find this topic to be difficult subject due its theoretical and mathematical nature. This paper presents BiGO, a tool to support the students to understand and learn to analyze the time complexity of algorithms in rigorous manner. BiGO has four main features: i. it enables students to encode any given primitive recursive function with a specific scripting language in order to ii. automatically compute the upper bounds of the script and to, iii. visualize the behavior of the upper bounds in comparison to the most commonly occurring time complexity classes and, iv. view how long it takes for a computer to process the algorithm based on the different input values. This paper outlines the pedagogical goals of the tool and the system implementation architecture. Furthermore, we discuss future developmental plans for the tool and an evaluation within undergraduate and graduate level courses such as Data structures and algorithms Design and analysis of algorithms.

CCS CONCEPTS
• Educational technology • Analysis of Algorithms • Theoretical computer science

KEYWORDS
Algorithm, Big O, Time complexity, CS education

1 Introduction

Theoretical computer science is a sub-field of computer science, which is focused on mathematical models of computing and computation. Research topics in theoretical computer science include for instance computational complexity theory [1], automata theory [2], cryptography [3], and analysis of algorithms [4]. Together with other topics in theoretical computer science, analysis of algorithms is usually taught in both graduate and undergraduate levels in the universities and other institutes of higher education as a part of computing degrees. Despite being important topic in computing as a discipline influencing to theoretical topics such as computational complexity but also to software engineering, students find analysis of algorithms in general among the most difficult topics in the curriculum [5].

Analysis of algorithms aims to rigorously prove the time or space requirements for the given algorithm. Other metrics are also used to analyze the efficiency of algorithms but usually the term analysis of algorithms is connected to the time and space bounds of algorithms. For the time complexity analysis of algorithms, Big O notation is commonly used [6]. Big O describes the running time required of an algorithm and is used when the analysis aims to show the upper bounds of the running time. That is, the worst case that the algorithm requires to run to the respect to the given input(s) as an asymptotic function. Multiple different complexity classes can be defined for the different asymptotic functions of Big O such as linear time algorithms, polynomial time algorithms and exponential time algorithms [7]. For the algorithms whose running time does not depend on the size of the input(s), the name constant time algorithms is used.

Multiple studies have risen the issues in teaching analysis of algorithms and have searched efficient ways to ease the process through visualizations and supporting software [8].
Such studies have produced software tools the help students to comprehend the running times, complexity classes and their relation to algorithms. The tools include visualizations, graphs and even estimates of the complexity classes for the given algorithm. Most widely known tool is Complexitor [11], which allows users to write algorithms in various programming languages such as Java and Python where Complexitor estimates the running time of the given algorithm by executing the algorithm with various inputs. The aim of Complexitor is not to give tight upper bounds nor lead to rigorous analysis of the algorithm. Instead, Complexitor aims to teach the users how the empirical analysis of algorithms can be conducted by comparing the known upper bounds to the execution times when the input values vary.

To support the students to learn rigorous analysis of algorithms, we introduce a tool named as BiGO that enables students to encode any given primitive recursive function with BiGO script (the programming language designed to be used with the tool), automatically calculate the upper bounds of the encoded algorithm, view the visualizations where the encoded algorithm is compared to the most commonly occurring algorithm complexity classes and finally viewing how long it would take for a computer to process the algorithm for the inputs given to the encoded algorithm. Primitive recursive functions are the functions that contain either loops or recursion where the loop invariants are immutable inside of the loop block. While there exist multiple functions that are not primitive recursive, such as the Ackermann function [16], most algorithms implemented in the courses on theoretical computer science are in fact primitive recursive.

BiGO offers extensive suite of examples for well-known algorithms such as search algorithm, graph and tree traversal algorithms, sorting algorithms and algorithms for well-known NP-Hard problems such as the subset sum problem [17] and Boolean satisfiability problem (SAT) [18] that are usually introduced in theoretical graduate level courses.

The key difference between BiGO and Complexitor is that BiGO does not estimate the upper bounds of the given algorithm. Instead, BiGO analyzes the generated parse tree of the given BiGO script and offers tight upper bounds. While the aim of Complexitor is in empirical analysis of the running time, BiGO aims to be used to learn more rigorous and mathematical methods to analyze the time complexities of algorithms.

BiGO was developed with the following pedagogical goals in mind:

1. BiGO script syntax resembles pseudocode and by converting algorithm to BiGO script, the user needs to understand the underlying structures of the algorithm by drawing a line between the constant and non-constant time operations.

2. BiGO tool highlights the different parts of the algorithm that have different influence in the upper bounds. Hence, the user is not given merely the upper bounds but instead shown how the upper bounds were derived.

3. BiGO does not estimate the running time but instead gives tight upper bounds based on the parse tree analysis and hence aims to support user to learn the rigorous analysis of the algorithms that is usually the case in the courses of theoretical computer science.

## 2. Related work

To support students to understand the theoretical behavior of algorithm and implement their own algorithm into source code, several learning tools have been developed over the years [5], [9], [11]. However, these set of tools have several features that does not support the understanding of pseudocode, which is the transition from algorithm to source code. BiGO script bear close resemblance to pseudocode that aids the learner to cognize the workings of the algorithm. Complexitor [11], which main focus is to support student to learn how to determine the time complexity of algorithm, the source code execution sequence depended on test cases from Pearson correlation. Learners are expected to directly formulate the source code and input set of algorithm to the tool to determine the time complexity, which does not enable the learner to comprehend the core structure of the algorithm. Furthermore, because Complexitor accepts codes from any programming language, it expects the learner to understand the technical details of each programming language, which may be difficult for a novice programmer. Over the years several visualization tools for algorithms have been researched as a solution to support learners to comprehend the workings of algorithms, BRIDGES, [12], Customizable Algorithm Visualization Tool for E-Learning, [13] VizAlgo and Algomaster platforms, [14] JSAV, [15]. For example, in BRIDGES visualization tool learners gain understanding of algorithms structure through visualization and exploration of the execution of the data structures of the algorithms [12], Customizable algorithm visualization tool [13], support learners to simulate different algorithms, calculate, and make comparisons of the steps involved in a particular algorithm and visualize any sorted algorithm in bar diagram. However, visualizing the behavior of the algorithm and comparing the steps is not adequate to gain an understanding of the time complexity of each step within the algorithm, as visualizing on bar diagram only sums-up the needed steps of the algorithm. One unique feature that makes BiGO tool novel is in the ability of the interpreter to ignore the block of codes that have no impact on the running time of the algorithm, laying emphasis on the blocks that matter, so that the student is aware of the components that actually determine the weight of the complexities. JCEL [9], a similar tool to Complexitor [11] that supports learning algorithm time complexity, by comparing BiG-O equation with real program execution. Although learners using JCEL do not need to have detailed practical knowledge of the programming language because JCEL automatically generate syntaxes that are language independent, however, several lines of codes that are not relevant to running time of the algorithm are evaluated, thereby, increasing cognitive work of the learner.

## 3. BiGO tool

### 3.1 Architecture

BiGO tool is built on top of BiGO interpreter that accepts BiGO script language. BiGO script is complete to context sensitive language [19] and hence, every $x \in L$ can be expressed as an expression where $L = (y^m \cdot z^n)$; every function, loop, recursive call and bracket can be nested and eventually requires to be closed. In total, the BiGO script contains 28 reserved keywords divided into categories of repetition (repeat
loops and recursion), arithmetics (basic operations such as divide, add, brackets and multiply), functions (mathematical functions such as logarithm, exponent, square root and factorial), casting (casts unary values to binary values, which avoids pseudo polynomial upper bounds) and variables, which are used for linear time or constant time operations. BiGO script also allows users to write one-line comments to the scripts by using #-tag before the comment. The interpreter will omit the text followed by the comment tag.

While allowing the users to write repetition structures in BiGO script, the syntax of BiGO script does not contain conditional statements. Hence, if an algorithm contains several branches of execution depending on some pre-defined conditions such as if ... then else -structures, the user has to write the conditions one by one with BiGO script and then compare the upper bounds and use the computationally most expensive branch of the execution. The conditions were not included to BiGO script because in the analysis of the algorithms, usually one uses computationally most expensive branch in the upper bounds and ignores the rest branches of the program flow in the analysis of the upper bounds.

The BiGO interpreter comes in three packages. First, the tokenizer creates tokens from the BiGO source. The tokenizer evaluates the given keywords and possible linear time operations as variables and ignores the EOL starting with the comment tag. After building the tokens from the source, the parser generates the parse tree from the tokens. Parser builds the parse tree recursively from the nested token blocks such as the nested repeat loops or nested functions. Finally, the generated parse tree is given to the evaluator and the evaluator calculates the time complexity from the parse tree ignoring constant time operations. The evaluator detects the level of the nesting in the nested operations and evaluates the number of the possible recursive calls by using Master’s theorem [20] as heuristics. The output of the evaluator is given in two different formats: TeX and standard Javascript function. Figure 1 illustrates the architecture of the tool.

![Figure 1: The architecture of the BiGO interpreter](image)

The given upper bounds of the evaluator are strict but the syntax of BiGO script only allows users to write primitive recursive functions, because it has been formally proven that there cannot exist a universal algorithm that would calculate the time complexity for every given function (Halting problem).

Complexitor derives the upper bounds to a given algorithm based on the evaluation of the running time of an algorithm in the inputs. This approach however has a downside that we have avoided in BiGO: if an algorithm with the actual time complexity of \( O(n^{\text{some very large constant}}) \) is given to be evaluated by Complexitor, Complexitor estimates the enormous running time of the algorithm to be exponential because there is a little difference between large polynomials and an exponential function in small inputs. While the algorithms with the upper bounds of exponential running time and \( O(n^{\text{some very large constant}}) \) in practice are very close to each other, the other one is polynomial time algorithm and the other one is exponential time algorithm. In theory, there is a fundamental difference between these two algorithms, which can only be detected with static analysis of the source code and empirical analysis of the execution time and of which every computer science student should be aware in the theoretical computer science courses.

In Figure 2a, it has been shown how a well known and quite simple sorting algorithm (the Bubble sort) has been written in Java, pseudocode and in Figure 2b, the BiGO script to illustrate how algorithms can be translated to BiGO script.

![Figure 2a: Java implementation and pseudocode of bubble sort](image)

```java
public class BubbleSort
{
    public static void bubbleSort(int arr[])
    {
        int n = arr.length;
        for (int i = 0; i < n-1; i++)
            for (int j = 0; j < n-i-1; j++)
                if (arr[j] > arr[j+1])
                    // swap arr[j+1] and arr[j]
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
    }
}
```

![Figure 2b: BiGO implementation of bubble sort](image)

```biogo
# BiGO implementation of bubble sort

startscript

# we loop from 0 to the end of the array
repeat n do
    # we loop from 0 to the end of the array
    repeat n do
        # we compare array[j] to array[j + 1] (the inner loop)
        if
            # and swap the elements if necessary
            c
        endrepeat
    endrepeat

# and return the sorted list
c

endscript
```
Writing a BiGO script needs student to identify three main elements from an algorithm: first, the student is required to distinguish operations other than repetition structures (loops or recursion) that require either a constant time (O(1)) or a time bounded by an asymptotic function (> O(1)). Second, the student is required to identify the repetition structures of the algorithm and define the length of the repetition structure. These are the very elements when analyzing analytically the upper bounds of an algorithm and apply also for BiGO script.

After these elements have been identified, the BiGO interpreter analytically calculates the upper bounds and highlights the pieces of the script that actually matter when deriving the upper bound for an algorithm.

3.2 GUI and Configuration system

GUI for the tool is written in HTML5 and Javascript, which is executed by Electron framework [21] in order to offer a desktop application look and feel. GUI contains a text editor with real-time syntax highlighting, a graph where the upper bounds of the user’s algorithm are compared to the well known asymptotic functions, the time complexity in TeX format and API for the BiGO script. GUI also offer extensive suite of examples of well known algorithms such as bubble sort, quicksort, in order traversal of a binary tree, depth first search of a graph, which are targeted for the beginners. The examples also include more complex algorithms such as DPLL algorithm for SAT and dynamic programming algorithm for sub-set sum.
problem that illustrates how pseudo-polynomial upper bounds are formed.

GUI of BiGO allows users to save the projects created in BiGO script to the local machine and can be used to import source code files from the different programming languages. BiGO, however, depends on the BiGO script and hence, the imported source code files from the different programming languages require the BiGO syntax to be commented to the source code with the // - one line comment or # - one line comment sign. The GUI of BiGO tool has been developed to be easy to use and powerful at the same time. The users can write multiple algorithm implementations and then compare the results among the written implementations. The users are also able view how the derived upper bounds are computed from the written algorithms: the GUI highlights the parts of the script in the text editor that have an influence in the derived upper bounds and the same highlighting is used in the asymptotic function rendered to the user. This way the user can follow the written algorithm in step by step manner and comprehend how the given upper bounds were actually calculated. The highlighting can support the students, for instance, to comprehend the recurrence relations. The GUI of BiGO tool can be seen in Figure 3 with an example for finding whether an input number is a prime number (note how BiGO avoid pseudo-polynomial upper bounds with a function `definebinary`).

4. Discussion and plans for future development of the BiGO tool

In the future, we test the BiGO in undergraduate and graduate level courses related to algorithm analysis such as Data structures and algorithms and Design and analysis of algorithms. We hope to gain deeper insight in how BiGO helps students to better comprehend the rigorous analysis of the upper bounds for algorithms in these courses. We are also interested in whether there are differences between the undergraduate and graduate level students when comparing the usefulness of the tool. Because BiGO can also be used to analyze more complex algorithms such as algorithms for NP-hard problems usually introduced in graduate level courses, the test cases of BiGO are not limited to the theoretical courses mentioned earlier.

The syntax of BiGO script is designed to resemble a pseudocode notation widely used in the theoretical computer science courses. However, there is no universal standard for pseudocode and hence, depending on the literature, the syntax used to describe an algorithm in a pseudocode level varies. We hope to develop the syntax of BiGO script further especially for function calls in arithmetic expressions to simplify the BiGO script even more.

The development of the tool continues and the next step in the development is to build a visual programming interface for the framework. This way the BiGO script can be written in both textual BiGO script and drag and drop -based visual programming language, which, under the hood, translates to BiGO script. As a visual programming language framework, our intention is to use Blockly [22].

Currently BiGO script accepts recurrence relations that are all equal within the BiGO script. The calculated upper bound for such recurrences is correct but not as tight as possible. Hence, in the future Akra-Bazzi method [23] solving such recurrences will be implemented as a part of the BiGO interpreter.

In this paper we have discussed mostly about primitive recursive functions. BiGO as a tool can be used to encode other algorithms as well. However, the loop invariant is immutable inside of the repeat loop block in BiGO script. Hence, if the user wishes to encode algorithms that are not primitive recursive, the variable representing the number of iterations in the repeat loop declaration should be defined accordingly and selecting such variable may require knowledge on how the loop body is executed. Also this leads to non-strict upper bounds. We hope to simplify these issues in the future and broaden the number of algorithms that can be encoded with BiGO script.

REFERENCES

satisfiability solver.” Proceedings of the 2001 IEEE/ACM international
[19] Zhang, Da-Qian, Kang Zhang, and Jiannong Cao. "A context-sensitive
graph grammar formalism for the specification of visual languages." The
[22] Culic, Ioana, Alexandra Radovici, and Laura Mihaela Vasilescu. "Auto-
generating Google Blockly visual programming elements for peripheral
hardware." 2015 14th RoEduNet International Conference-Networking in
generation and automated trip segmentation using location data." U.S.