EXPOSE: Inferring Worst-case Time Complexity by Automatic Empirical Study

Cody Kinneer ★ Gregory M. Kapfhammer ★ Chris Wright ☆ Phil McMinn ☆

★ Allegheny College ☆ University of Sheffield

Introduction to doubling. A useful understanding of an algorithm's efficiency, the worst-case time complexity gives an upper bound on how an increase in the size of the input, denoted n, increases the execution time of the algorithm, or $f(n)$. This relationship is often expressed in the "big-Oh" notation, where $f(n)$ is $O(g(n))$ means that the time increases by no more than on order of $g(n)$. Since the worst-case complexity of an algorithm is evident when n is large [1], one approach for determining the big-Oh complexity of an algorithm is to conduct a doubling experiment with increasingly bigger input sizes. By measuring the time needed to run the algorithm on inputs of size n and $2n$, the algorithm's order of growth can be determined [1].

The goal of a doubling experiment is to draw a conclusion regarding the efficiency of the algorithm from the ratio $f(2n)/f(n)$ that represents the factor of change in runtime from inputs of size n and $2n$. For instance, a ratio of 2 would indicate that doubling the input size resulted in the runtime's doubling, leading to the conclusion that the algorithm under study is $O(n)$ or $O(n \log n)$. Table 1 shows some common time complexities and corresponding ratios.

Ratio $f(2n)/f(n)$	Worst-Case Conclusion
1	constant or logarithmic
2	linear or linearithmic
4	quadratic
8	cubic

Table 1: Conclusions for worst-case time complexity.

Automatic doubling. EXPOSE [2, 3] is a tool to derive an "EXPerimental bigOh" for supporting "Scalability Evaluation". EXPOSE infers an algorithm's big-Oh order of growth by conducting a doubling experiment automatically. In order to evaluate an algorithm A, EXPOSE takes as input two functions. The first is a timing function $f(n)$ that runs an implementation of A on the provided input of size n and returns the runtime, and the second is a doubling function $d(n)$ that accepts an input for A and returns an input of size $2n$. After providing EXPOSE an initial input, the tool will output an inferred big-Oh order of growth for A.

EXPOSE derives the worst-case time complexity of A by repeatedly doubling the input until n is large enough that the worst-case time complexity of A is apparent. EXPOSE determines when n is large enough by monitoring the doubling ratio $\frac{f(2n)}{f(n)}$ for multiple iterations of doubling. Using a convergence algorithm, EXPOSE stops the doubling experiment when the doubling ratio reaches a stable value.

To test for convergence, for every time t, where t denotes the number of times the input has been doubled, we record the doubling ratio $r_t = \frac{f(2^t n)}{f(2^{t-1} n)}$. The current ratio r_c is compared to a previous ratio r_p where p is determined by a *lookback* value, such that $p = c - lookback$. The result of the comparison is a *difference* value, given by $difference = |r_c - r_p|$. This is then compared to a *tolerance* value, and the experiment is judged to have converged when $difference < tolerance$. The *lookback* and *tolerance* values are both configurable parameters.

Early use of the tool revealed that this converge checking rule was not enough, since a very small initial n may complete nearly instantaneously even for multiple rounds of doubling. For example, the time that it takes to sort a list of size $1, 2, 4, 8, \ldots, 128$ might not even be distinguishable. This would appear to converge to 1, which indicates constant time complexity. To prevent the experiment from incorrectly terminating given a small starting n, EXPOSE requires that a program under study display a ratio of 1 for a *minimum* number of times before judging that the ratio does in fact converge to 1. That is, if $r_c = 1, t > minimum$ must be true, in addition to the tolerance test, before the experiment is declared convergent. The *minimum* value is also a configurable parameter. Because a doubling ratio of 1 signifies constant or logarithmic time complexity, requiring these doubles does not significantly increase the experimentation time needed, all the while providing further assurance that a small ratio is not due to an insufficiently small n.

Implementation. EXPOSE is implemented as a package of classes in the Java programming language [3]. To use EXPOSE to evaluate a new algorithm A, you only need to extend the `DoublingExperiment` class to provide your own f and d functions. The f function should be implemented by providing a `double timedTest()` method, and d should be implemented by providing a `void doubleN()` method. Note that these methods do not accept any parameters, and only `timedTest()` returns a value. The programmer must ensure that `timedTest()` returns the runtime for the current input size, and that when `doubleN()` is called, the input size is doubled; initializing and storing this input should be handled by the user-provided implementation. The `runExperiment()` method can be called to conduct a doubling experiment and `printBigOh()` can be called to show the result. Figure 1 shows an complete Java class that conducts a doubling experiment on QuickSort; note the simplicity of the implementation when using EXPOSE.

```
public class QuickSortExp extends DoublingExperiment{
    private int size = 10;
    public static void main(String[] args){
        SortingExperiment exp = new SortingExperiment();
        exp.runExperiment(); exp.printBigOh(); }
    protected void doubleN(){ size *= 2; }
    protected double timedTest(){
        int[] n = createInput(size);
        long startTime = System.nanoTime();
        QuickSort.quickSort(n, n.length);
        long endTime = System.nanoTime();
        return (double) endTime - startTime; } }
```

Figure 1: A simple Java class that conducts a performance evaluation of the QuickSort algorithm.

Case study: Sorting Algorithms. Included with the EXPOSE tool is an example doubling experiment called `SortingExperiment`. This program provides a number of canonical sorting algorithms with well-known worst-case time complexities. Doubling experiments may be performed on these algorithms by running the command `java SortingExperiment algname`, replacing `algname` with the name of the desired sorting algorithm. Running the command without providing `algname` will show a list of options. When run 1000 times for each of the five provided sorting algorithms, EXPOSE achieves an accuracy of 98.84%.

Case study: *SchemaAnalyst*. In other work [2], we used EXPOSE to perform a comprehensive analysis of the search-based test data generation tool, *SchemaAnalyst*, that generates test suites for relational database schemas [4]. Since it is much more complicated than a sorting algorithm, performing doubling experiments on *SchemaAnalyst* requires more parameters than needed to study sorting [5]. To conduct these experiments, we developed a class called `SchemaExperiment` that extends `DoublingExperiment`. We developed `SchemaExperiment` to allow for conducting doubling experiments using a variety of *SchemaAnalyst* configurations, as well as accessing EXPOSE's parameters.

```
 Usage: <java SchemaExperiment> [options]
Options:
 --schema, -s          Select which schema to use
 --criterion           Select which criterion to use
 --datagenerator       Select which data generator to use
 --doubler             Select which schemaDoubler to use
 --convergence         Experiment convergent if diff < this
 --lookBack            Number of ratios to compare for convergence
 --tuningTries         Minimum number of times to doubles before O(1)
 --minDoubles          Minimum number of doubles to try
 --giveUp, --maxTime   Max time for a single trial in hours
 --help, --usage       Display command line options
 -o, --out, --csv      Desired csv filename for saving data
 --verbose, --debug    Display verbose output
```

Following the terminology from [2], a doubling experiment to evaluate *SchemaAnalyst* using the AICC criterion, a random data generator, the RiskIt database schema, and the number of `NOT NULL`s in the schema will run with this command: `java SchemaExperiment --criterion AICC --datagenerator random --schema RiskIt --doubler DoubleNotNullsSemantic`. Although less accurate than in the sorting case study, EXPOSE still successfully revealed meaningful trends in *SchemaAnalyst*'s performance [2].

Deploying on a High-Performance Cluster. Since the performance of *SchemaAnalyst* may depend on a number of factors (i.e., criterion, data generator, schema, and doubling strategy) a comprehensive survey of the parameter space may be conducted by performing a doubling experiment for each configuration. While computationally expensive, an experiment of this scale is possible by using a high-performance computing (HPC) cluster. Each doubling experiment can be run independently on a separate node of the cluster; EXPOSE can combine the resulting data for a later analysis. Data mining techniques can then be leveraged to interpret an algorithm's performance trade-offs.

Parameter Tuning. While EXPOSE greatly eases the process of conducting doubling experiments, its accuracy and performance is sensitive to the settings of the system's parameters. In particular, the $tolerance$ and $lookback$ values can result in a doubling experiment terminating prematurely or continuing indefinitely. To complicate the issue further, the parameters must be re-tuned based on hardware properties of the machine(s) being used and the performance characteristics of the implementation being studied.

The reliability of the tool and repeatability of its results would be further improved if EXPOSE could select good settings for these parameters automatically. A reasonable parameter tuning strategy could be to run EXPOSE on various algorithms of known worst-case time complexities, such as the sorting algorithms, and lower the *tolerance* threshold until EXPOSE reliably infers the big-Oh time complexities.

Future Work and Conclusion. While we recently used EXPOSE to study search-based test data generation in the domain of relational database schemas [2], the tool is general and can be applied to many other problem domains. Future work includes using EXPOSE to evaluate the efficiency of EVOSUITE's approach to test data generation for Java programs [6]. In conclusion, EXPOSE makes empirically evaluating the worst-case time complexity of algorithms very convenient. By automating the process of conducting these experiments, EXPOSE enables large-scale empirical studies that would otherwise be infeasible.

References

[1] C. C. McGeoch, *A Guide to Experimental Algorithmics*, 2012.

[2] C. Kinneer, G. M. Kapfhammer, C. J. Wright, and P. McMinn, "Automatically evaluating the efficiency of search-based test data generation for relational database schemas," in *Proc. of 27th SEKE*, 2015.

[3] C. Kinneer, "EXPOSE software tool," 2015. [Online]. Available: https://github.com/kinneerc/ExpOse/

[4] G. M. Kapfhammer, P. McMinn, and C. J. Wright, "Search-based testing of relational schema integrity constraints across multiple database management systems," in *Proc. of 6th ICST*, 2013.

[5] J. Kempka, P. McMinn, and D. Sudholt, "Design and analysis of different alternating variable searches for search-based software testing," *Theor. Comp. Sci.*, 2015, In Press.

[6] G. Fraser and A. Arcuri, "1600 faults in 100 projects: Automatically finding faults while achieving high coverage with EVOSUITE," *Empir. Softw. Engin.*, 2013.