Automatically Evaluating the Efficiency of Search-Based Test Data Generation for Relational Database Schemas

Cody Kinneer ★ Gregory M. Kapfhammer ★ Chris Wright ☆ Phil McMinn ☆
★ Allegheny College ☆ University of Sheffield

Abstract

The characterization of an algorithm's worst-case time complexity is useful because it succinctly captures how its runtime will grow as the input size becomes arbitrarily large. However, for certain algorithms—such as those performing search-based test data generation—a theoretical analysis to determine worst-case time complexity is difficult to generalize and thus not often reported in the literature. This paper introduces a framework that empirically determines an algorithm's worst-case time complexity by doubling the size of the input and observing the change in runtime. Since the relational database is a centerpiece of modern software and the database's schema is frequently untested, we apply the doubling technique to the domain of data generation for relational database schemas, a field where worst-case time complexities are often unknown. In addition to demonstrating the feasibility of suggesting the worst-case runtimes of the chosen algorithms and configurations, the results of our study reveal performance trade-offs in testing strategies for relational database schemas.

1 Introduction

Many disciplines, such as science, finance, and medicine, rely on relational databases to maintain large amounts of critical information [1]. The relational database schema defines the structure of a database and protects the integrity of the data. This makes testing the database schema necessary to avoid the corruption of data. Search-based algorithms, that use a fitness function to offer guidance to a desirable solution, have been applied to this challenging problem [2]. Although data generation for relational schemas may also be handled, albeit less effectively, with random generation techniques [3], the use of search-based approaches ensures that data creation methods can actively seek out test inputs that best fulfill testing goals [4].

Despite the effectiveness of search-based data generation methods, there is, to the best of our knowledge, little prior research that fully studies their efficiency and characterizes their worst-case time complexity. In part, we attribute this dearth of past work to the fact that these systems are complex, thus making a generalizable theoretical analysis hard.

In response to the lack of insight into the performance of search-based methods, this paper presents a fully automated performance evaluation framework that employs doubling experiments to suggest worst-case time complexities and conditional inference trees to identify efficiency trends. Applying this framework to the automated performance evaluation of search-based test data generation for database schemas, the results reveal trade-offs in efficiency with respect to the chosen testing goals, the structure of the relational schema, and the data generation strategy.

Since the presented approach is fully automated, it enabled a comprehensive study suggesting the worst-case time complexity of all the relevant data generator configurations. Although this paper focuses on automatically evaluating the efficiency of search-based test data generation for the database schema, the presented technique can be applied to a wide range of other methods using heuristic search. In summary, this paper's important contributions include:

1. A performance evaluation framework that automatically conducts and analyzes the results from doubling experiments with search-based methods.
2. Empirically derived suggestions for the worst-case time complexity of search-based test data generators.
3. With a systematic focus on a wide variety of configurations, an empirical study revealing trade-offs in search-based test data generation for relational schemas.

2 Background and Related Work

Testing Database Schemas. The relational database, a cornerstone of modern software, is protected by a schema that defines integrity constraints ensuring the coherence of data. These constraints defend the schema from manipulations that could violate requirements such as "user names must be unique" or "the host name cannot be missing or unknown". Prior work in this area proposed coverage criteria, derived from logic coverage criteria, that establish different levels of testing for the formulation of integrity constraints in a database schema [3]. These range from simple criteria that mandate the testing of successful and unsuccessful `INSERT` statements into tables to more advanced criteria that test the formulation of complex integrity constraints such as multi-column `PRIMARY KEY`s and arbitrary `CHECK` constraints. This family of criteria has been organized into a subsumption hierarchy, with criteria such as *Clause-Based Active Integrity Constraint Coverage* (ClauseAICC) emerging as a stringent testing strategy. Space constraints limit further commentary on testing methods for database schemas; prior work [3] provides additional details.

DOI reference number: 10.18293/SEKE2015-205

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.

Search-Based Test Data Generation. When testing a schema's integrity constraints for correctness, it is often necessary to provide input to the database and then observe and evaluate its execution [2]. Since the database's behavior is dependant on the input from `INSERT`s, the input space must be sufficiently explored to ensure thorough testing. Due to the fact that it is challenging to manually create input that supports high-quality testing, test data generation is used to automatically produce it according to a criterion, like ClauseAICC. A search-based test data generator is one that explores that input space using, among other components, a fitness function that rates the data's quality, thus allowing it to improve by repeatedly seeking better inputs [4].

Worst-Case Time Complexity. 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, $f(n)$. 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 [5], 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 an input of size n and the time needed to run with input of size $2n$, the algorithm's order of growth can be empirically determined [5, 6].

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 input sizes n to $2n$. For instance, a ratio of 2 would indicate that doubling the input size resulted in the runtime's doubling, thus 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 their corresponding ratios.

Related Work. Goldsmith et al. [7] developed a tool, called *Trend-Prof*, that empirically evaluates the computational complexity of a program by using code instrumentation to count the number of times each block of code is executed and then grouping these blocks by their behavior. *Trend-Prof* takes in a collection of workloads, user-specified features of the workloads, and the program to be studied. While this technique results in a more detailed analysis than the one presented in this paper, Goldsmith et al. did not address the issue of generating the workloads necessary to achieve a meaningful result, which this paper's technique can handle automatically. Our paper is also contrasted with this prior work because it describes experiments in a domain, search-based test data generation, where the method's worst-case time complexity is not always known.

Zhao et al. presented an empirical study of the performance of search-based test data generation for extended finite state machine (EFSM) models [8]. Although this paper focused on efficiency and made preliminary observations about the relationship between performance and the characteristics of an EFSM model, it did not, like our paper, use doubling experiments to suggest worst-case time complexities. Lakhotia et al. also reported on an experimental analysis of the efficiency and effectiveness of search-based test data generation for C programs [9]. While our paper looks at generator performance in a holistic manner, this prior work considered the number of fitness evaluations during data generation. Similar to our use of doublers that systematically increase the size of a schema, Mehrmand and Feldt empirically studied, with a focus on code coverage, search-based data generation as program size increased [10].

The empirical work presented in this paper is complemented by theoretical runtime analyses in prior research. For instance, Arcuri presented the first runtime analysis of a search-based test data generator called the alternating variable method (AVM) [11], which is also studied in this paper. Arcuri proved the worst-case time complexity of AVM when it generates data for a simple program called "triangle classification". More recently, Kempka et al. extended the work of Arcuri with a theoretical and empirical runtime analysis revealing that the use of certain local search techniques with AVM yields better performance than AVM alone [12]. While our paper's automated framework can easily be applied to new schemas—and even to other types of search-based test data generators—the results in these two aforementioned papers are more difficult to generalize.

3 Automated Doubling Experiments

Overview. The presented technique for performing automatic doubling experiments consists of two key components. The first is a method for systematically doubling the initially input relational schema, and the second is a rule for determining when a valid conclusion can be drawn from the experiment, thus allowing the doubling process to stop.

Doubling Schemas. Determining worst-case complexity by a doubling experiment requires that the size of the input be doubled. A relational database schema is a complex artifact with many features and interrelationships. This makes doubling rule implementation a non-trivial task.

A relational database schema contains tables and columns, and constraints that restrict the values allowed into these entities. Since the runtime of a schema testing technique may be affected by the number of any of these, it is desirable to have a strategy for doubling each one. Doubling the number of tables or columns in a schema is relatively easy. It is possible to double the number of tables

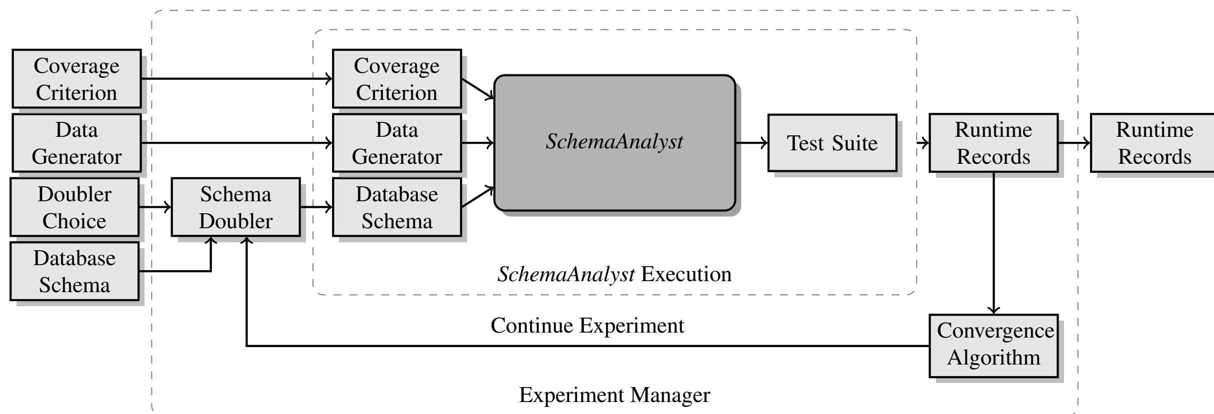


Figure 1: Technique for conducting automatic doubling experiments.

in a schema by following this rule: for every table present in the schema, create a new empty table. It is important that the new tables be empty to avoid changing more than one doubling parameter at once—if the new tables contained columns, for instance, then the number of tables and columns in the schema both would be increased, thus interfering with assessing table doubling's impact on performance. Additionally, doubling the number of columns can be accomplished by, for every table in the schema, and for every column, adding a new column to that table.

Doubling integrity constraints is more challenging. The `FOREIGN KEY` constraint, for instance, denotes a relationship between two tables, thus making it difficult to double without introducing extraneous database entities or cyclic dependencies. Since a `CHECK` constraint can express arbitrary conditions, it is also challenging to double if the meaning of each constraint must be considered to ensure satisfiability. Since a table can only contain one `PRIMARY KEY`, if a schema contains five tables, then at most it can have five `PRIMARY KEY` constraints, as adding more keys would require creating more tables, which should be avoided.

Because of these issues, and others like them, we focus our attention on constraints that can be doubled as follows: for every table and for every constraint, duplicate that constraint and re-add it to the table. Constraints such as `NOT NULL`, `UNIQUE`, and `CHECK` are amenable to doubling in this fashion. It is worth noting that, since they amount to a restatement of existing constraints, entities doubled this way would not have an impact on what data the schema would allow or disallow into a database, However, since the goal is to evaluate performance, the timing results should not be adversely affected as long as the test data generation technique must still process and consider these additional constraints.

Automatic Experimentation. To determine worst-case complexity, an input of size n is doubled until the ratio $f(2n)/f(n)$ converges to a stable value. To account for random error, every time n is doubled, $f(n)$ is computed ten times and the median time is used for calculating the ratios; we chose the median to minimize the effect of outliers. If the mean is used instead, then a single abnormally long run could have an outsized impact on the result. Figure 1 shows the overall structure of the experimentation framework.

Convergence checking is necessary because of the fact that worst-case time is only evident for large values of n. If too few doubles are tried, then the experiment may terminate before n reaches a value where the true worst-case time complexity is apparent. At the same time, for inefficient algorithms, each additional doubling run incurs a substantial time overhead. For the sake of efficiency, the doubling experiment should terminate as quickly as is possible.

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 $p = c - \mathit{lookback}$. The result of the comparison is a *difference* value, given by $\mathit{difference} = |r_c - r_p|$. This is then compared to a *tolerance* value, and the experiment is judged to have converged when $\mathit{difference} < \mathit{tolerance}$. The *lookback* and *tolerance* values are both configured before the experiment is run.

Another consequence of worst-case time only being apparent for large n is that a very small initial n may appear to converge to 1, which would indicate constant time complexity. To prevent the experiment from incorrectly terminating given a small starting n, our method 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 $r_c = 1$, $t > \mathit{minimum}$ must be true, in addition to the tolerance test, before the experiment is declared convergent. The *minimum* parameter is also configured before an experiment. 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.

Schema	Tables	Columns	Constraints
BioSQL	28	129	186
Cloc	2	10	0
iTrust	42	309	134
JWhoisServer	6	49	50
NistWeather	2	9	13
NistXTS748	1	3	3
NistXTS749	1	3	3
RiskIt	13	57	36
UnixUsage	8	32	24

Table 2: Database schemas used in the experiments.

4 Empirical Analysis

Experimental Design. To gain a full picture of the performance trade-offs, we conducted an experiment for every configuration of the parameter space (i.e., schema, coverage criterion, data generator, and doubling technique). Table 2 shows that the experiments focused on nine database schemas containing between 1 and 42 distinct tables, 3 to 309 columns, and up to 186 constraints. Including all of the test adequacy criteria proposed by McMinn et al. [3], the experiments study “weak” criteria (i.e., APC, NCC, ICC, and UCC), “moderately strong” ones (i.e., ANCC, AICC, and AUCC), and “strong” criteria (i.e., CondAICC and ClauseAICC). More details about each criterion, including its formal definition and relationship to the other criteria, are available in prior work [3]. We used all six test data generators provided by the *SchemaAnalyst* tool for automated test data generation [2], with four techniques employing a variant of random search and two based on Korel’s alternating variable method. After a restart of the search, each data generator could start with either default or random values.

In our study, we set $tolerance$ to 0.40 and $lookback$ to 4. These values were chosen by performing doubling experiments on various algorithms, with known worst-case time complexities, and observing that the ratio converged to the correct value with this configuration. After observing that *SchemaAnalyst* stopped displaying constant behavior after around five doubles, we set $minimum$ to be four times this number. Preliminary studies showed that, while experiments for “fast” configurations could be completed in less than an hour, “slower” configurations required days. Since there are over two thousand possible configurations, the study needed a substantial amount of computational resources. As a solution, we ran the experiments on a high-performance computing (HPC) cluster containing 195 worker nodes of various hardware configurations, ranging from 12 to 16 CPU cores and 24 to 256 GB of memory, and using a 64-bit GNU/Linux operating system.

Results. Our experiments reveal that, when doubling `UNIQUE`s, `NOT NULL`s, and `CHECK`s, *SchemaAnalyst* displays linear or linearithmic worst-case time complexity. Out of the 699 experiments performed to double these schema structures, 72% converged to linear or linearithmic. Another 8% failed to converge, and of these experiments, 80% failed because of memory limitations, 13% exceeded the maximum time limit, and 8% failed for reasons that could not be determined. The doubling ratios among these experiments were primarily linear or linearithmic at the time they were terminated, however there were 14 that were quadratic and 3 that were cubic. The experiments that failed to converge were primarily generating test data for complex schemas, such as iTrust and BioSQL, and the most stringent adequacy criteria, such as ANCC and AUCC. The remaining 20% of the 699 experiments converged on constant or logarithmic. Since there did not seem to be a pattern in which configurations converged this way compared to linear or linearithmic, it is likely that they terminated before the true worst-case time complexity was apparent.

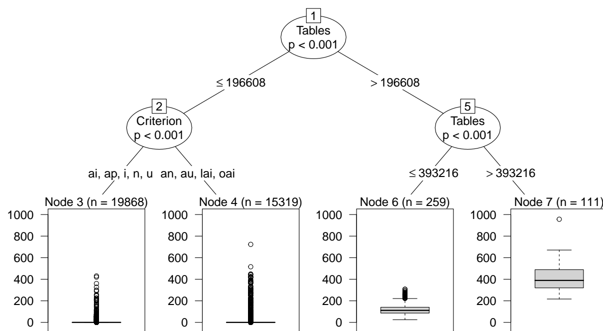
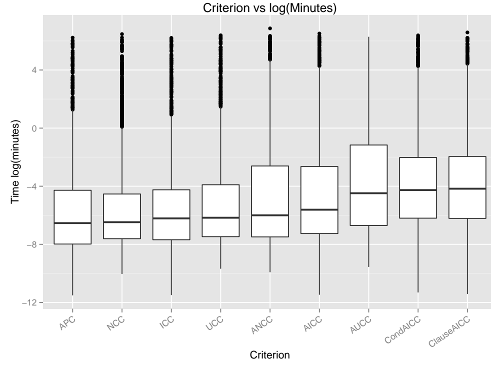


Figure 2: Tree model using all variables to predict runtime in minutes, demonstrating the important of the table count. Due to space constraints, criterion names are abreviated.

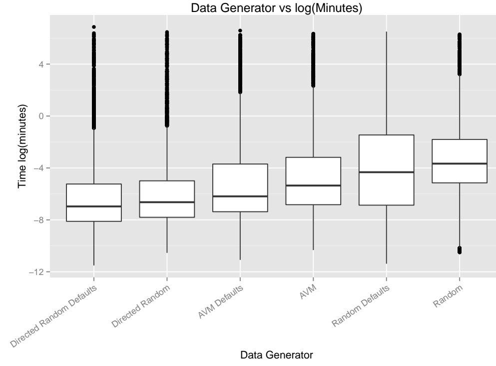
When doubling the tables and columns in the schemas, the results were less conclusive. Doubling the number of tables in the schema caused the runtime of *SchemaAnalyst* to increase much faster than it did for the other integrity constraints. As a result, 56% of the 467 experiments doubling this schema feature were terminated before convergence because they exceeded the time limit. Of the experiments that converged, 72 converged to quadratic and 10 converged to cubic. Of the experiments that terminated before they converged, the doubling ratios for 205 indicated quadratic, 18 suggested cubic, and 37 were worse than cubic.

Experiments on the number of columns were also inconclusive. We noted that 208 of the converged experiments showed linear or linearithmic time complexity, while 28 converged to quadratic and 2 cubic. Another 203 experiments failed to converge; however, unlike the experiments that doubled the number of tables, the experiments for doubling the number of columns most frequently failed by running out of memory rather than exceeding the time limit. The experiments that did not converge included 106 ratios indicating quadratic behavior, 73 cubic, and 3 worse.

To gain a more nuanced understanding of the results, our tool constructed a conditional inference tree model using



(a) Coverage criterion versus runtime in minutes.



(b) Data generator versus runtime in minutes.

Figure 3: Box and whisker plots for criterion and data generator.

the *ctree* package in the R language. These trees use the values of predictor variables (e.g., the adequacy criterion) to model the value of a response variable (e.g., *SchemaAnalyst*'s runtime); *ctree* accomplishes this by repeatedly splitting the data according to what predictor variable has the most influence on the response variable. Each tree node represents a choice of predictor variable, and the level of the node indicates its importance to the prediction, with higher nodes being more important to predicting generation time.

Using predictor variables for the number of tables, columns, `UNIQUE`s, `NOT NULL`s, and `CHECK`s; and the chosen criterion and data generator, *ctree* produced the tree model in Figure 2. In addition to confirming that the number of tables has the greatest impact on runtime, the tree also reveals that, when the number of tables in the schema is small, the choice of coverage criterion is most significant. While the number of tables had a large impact when over $197,000$, in practice schemas are unlikely to be this large. Another invocation of *ctree*, excluding tables from the list of predictors, provided insight into the behavior of *SchemaAnalyst* for more practical table sizes. In this tree, not shown due to space constraints, the coverage criterion emerged as the most important predictor for runtime, followed by the choice of data generator, and then the number of columns.

While the trees provide insight into the relative impact of each predictor, the box and whisker plots shown in the leaves of the trees do not furnish a detailed view of the choices within each predictor. To gain a finer-grained understanding, we created box and whisker plots of our own: Figure 3a shows the influence of coverage criterion on runtime, while Figure 3b shows the effect of data generator on runtime. Figure 3a shows that the strongest coverage criteria in the subsumption hierarchy (i.e., AUCC, ClauseAICC, and CondAICC) cause runtime to increase the most, followed by ANCC and AICC, and then the remaining criteria (i.e., APC through UCC). We anticipate that the stronger criteria always lead to higher time overheads because they force *SchemaAnalyst* to generate more tests. Also, criteria at the same level in the hierarchy engender similar runtimes.

Figure 3b reveals that, by a substantial margin, the Random and Random Defaults generators took the most time to generate data. This counterintuitive result suggests that less effective data generators actually take longer to create data than those that are known to be more effective [2]. A less pronounced difference between the remaining generators can be observed, with the use of default values consistently being faster than the use of random values at restart.

While the box and whisker plots show how choices between coverage criteria and data generators affect runtime, the question remains if these differences are statistically and practically significant. To answer this question, we employ the Wilcoxon rank-sum test and the $\hat{A}_{12}$ effect size [3].

The Wilcoxon rank-sum test is a non-parametric test for hypothesis testing. If the result of the test is greater than the significance level (0.05 is frequently used), then the configurations are indistinguishable. If, however, the result is less than the chosen level, then they are different. The $\hat{A}_{12}$ test is similar, but for drawing conclusions about the practical difference between two collections of data. A result of $\hat{A}_{12} = 0.50$ means that any difference is not practically significant, while $\hat{A}_{12} > 0.56$ or < 0.44 signifies a small difference, $\hat{A}_{12} > 0.64$ or < 0.36 denotes a medium difference, and $\hat{A}_{12} > 0.71$ or < 0.29 indicates a large difference.

Table 3 shows the statistical tests calculated for every pair of coverage criteria. The Wilcoxon rank-sum test reveals that changing the criterion results in statistically significant differences in runtimes, with the exception of changing between the four criteria at the top of the subsumption hierarchy and the two criteria at the bottom. The $\hat{A}_{12}$ results generally show a small to medium practical effect size when switching between criteria at the high or low end of the hierarchy, and small or no effect when switching between criteria at the same level of the hierarchy.

The statistical tests were calculated for all pairs of data generators, but the resulting table was omitted due to space constraints. All comparisons of data generators were statistically significant according to the Wilcoxon rank-sum test. The $\hat{A}_{12}$ values show that all choices of data gener-

	APC	ANCC	CondAICC	NCC	AUCC	AICC	ClauseAICC	ICC	UCC
APC	NA	0.425	0.337	0.484	0.334	0.413	0.329	0.481	0.449
ANCC	2.20E-16	NA	0.407	0.561	0.405	0.484	0.399	0.554	0.526
CondAICC	2.20E-16	2.20E-16	NA	0.671	0.503	0.581	0.492	0.656	0.634
NCC	1.20E-02	2.20E-16	2.20E-16	NA	0.335	0.417	0.322	0.491	0.461
AUCC	2.20E-16	2.20E-16	6.92E-01	2.20E-16	NA	0.577	0.490	0.651	0.628
AICC	2.20E-16	1.70E-02	2.20E-16	2.20E-16	2.20E-16	NA	0.412	0.571	0.547
ClauseAICC	2.20E-16	2.20E-16	2.72E-01	2.20E-16	1.40E-01	2.20E-16	NA	0.662	0.641
ICC	4.00E-03	2.20E-16	2.20E-16	1.83E-01	2.20E-16	2.20E-16	2.20E-16	NA	0.472
UCC	9.30E-16	3.83E-05	2.20E-16	7.36E-10	2.20E-16	5.73E-13	2.20E-16	9.29E-06	NA
	Rank-Sum:	significant	insignificant		$\hat{A}_{12}$:	none	small	medium	large

Table 3: For each pair of coverage criteria, lower left shows Wilcoxon Rank-Sum Test, upper right shows $\hat{A}_{12}$.

ator have at least a small practical impact, with the exception of choosing between random and random defaults, and directed random and directed random defaults. Changing between these data generators results in a large to medium effect size, and comparing either of the AVM-based generators to the other primarily resulted in a small difference.

Threats to Validity. Our technique for doubling the number of constraints in the schema is simply to duplicate the existing constraints. It is possible that *SchemaAnalyst* does less work processing these redundant constraints than it would given non-restated ones. However, doubling the constraints in this way is easy to implement and, as the results show, good at revealing performance trade-offs. Additionally, since worst-case time is only apparent for large n, it is possible that the experiments terminated too quickly. While we attempted to configure the parameters of our tool using algorithms with known worst-case complexities and conducting preliminary experiments with various settings and under manual supervision, it is possible that our configuration was not optimized for use on the HPC cluster.

5 Conclusions and Future Work

This paper presented an automated method for empirically suggesting the worst-case time complexity of search-based test data generation methods. Focusing on the domain of relational database schemas, our approach repeatedly doubles the size of the input schema and observes the commensurate change in runtime. Although some results are inconclusive, we find that, in many cases, data generation is linear or linearithmic and, in others, it is quadratic, cubic, or worse. Our automated method also revealed that, for all of the test adequacy criteria in the subsumption hierarchy presented by McMinn et al. [3], stronger criteria always necessitate more time for test data generation.

Since this paper's technique did not consider the doubling of constraints like `FOREIGN KEY`s, future work will focus on creating doublers for these unstudied constraints. Additionally, the current doubling mechanism avoids introducing semantically invalid constraints by restating existing constraints; in future work we plan to implement and evaluate more realistic ways to double relational schemas. Because certain experiments timed out before converging, we also want to re-run these configurations with longer time limits and more memory. Finally, we will investigate how automated parameter tuning, instead of manual tuning before experimentation in a new execution environment, can support choosing the convergence condition. Ultimately, the combination of the presented framework with the completed future work will yield an effective way to empirically understand the worst-case case time complexity of search-based test data generation for relational database schemas.

References

[1] G. M. Kapfhammer, "A comprehensive framework for testing database-centric applications," Ph.D. dissertation, University of Pittsburgh, 2007.

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

[3] P. McMinn, C. J. Wright, and G. M. Kapfhammer, "An analysis of the effectiveness of different coverage criteria for testing relational database schema integrity constraints," Department of Computer Science, University of Sheffield, Tech. Rep., 2015.

[4] P. McMinn, "Search-based software test data generation: A survey," *Soft. Test., Verif. and Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[5] C. C. McGeoch, *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

[6] R. Sedgewick and M. Schidlowsky, *Algorithms in Java: Fundamentals, Data Structures, Sorting, Searching*, 3rd ed. Addison-Wesley Longman Publishing Co., Inc., 1998.

[7] S. F. Goldsmith, A. S. Aiken, and D. S. Wilkerson, "Measuring empirical computational complexity," in *6th ESEC/FSE*, 2007.

[8] R. Zhao, M. Harman, and Z. Li, "Empirical study on the efficiency of search based test generation for EFSM models," in *3rd ICSTW*, 2010.

[9] K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: An open source tool for search based software testing of C programs," *Inf. Softw. Technol.*, vol. 55, no. 1, 2013.

[10] A. Mehrmand and R. Feldt, "A factorial experiment on scalability of search-based software testing," in *3rd AITSE*, 2010.

[11] A. Arcuri, "Full theoretical runtime analysis of alternating variable method on the triangle classification problem," in *1st SSBSE*, 2009.

[12] 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.