Genetic Algorithms for Affine Transformations to Existential t-Restrictions

Ryan E. Dougherty*
Arizona State University
Tempe, Arizona
ryan.dougherty@asu.edu

Erin Lanus
Arizona State University
Tempe, Arizona
erin.lanus@asu.edu

Charles J. Colbourn
Arizona State University
Tempe, Arizona
Charles.Colbourn@asu.edu

Stephanie Forrest
Arizona State University
Tempe, Arizona
stephanie.forrest@asu.edu

ABSTRACT

The subject of t-restrictions has garnered considerable interest recently as it encompasses many different types of combinatorial objects, all of which have unique and important applications. One of the most popular of these is an ingredient in the generation of covering arrays, which are used for discovering faulty interactions among software components. We focus on existential t-restrictions, which have a structure that can be exploited by genetic algorithms. In particular, recent work on such restrictions considers affine transformations while maximizing the corresponding "score" of the formed restriction. We propose to use genetic algorithms for existential t-restrictions by providing a general framework that can be applied to all such objects.

CCS CONCEPTS

• **Mathematics of computing → Discrete mathematics**; **Combinatorics**;

KEYWORDS

covering array, covering perfect hash family, t-restriction

ACM Reference Format:
Ryan E. Dougherty, Erin Lanus, Charles J. Colbourn, and Stephanie Forrest. 2019. Genetic Algorithms for Affine Transformations to Existential t-Restrictions. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3319619.3326823

1 INTRODUCTION

Imagine that some input to a software system produces incorrect or unexpected behavior. Efficient methods are needed to find faults of this type. Suppose that there are k components of the system, and each component i has a set of v_i possible inputs to the component. Overall, there are potentially $\prod_{i=1}^{k} v_i$ possible tests that can be run on this system; this is called a *full factorial design*. Such a design guarantees to test every possible set of inputs to the system; if any one of them produces an incorrect output, then we have determined the existence of at least one fault in the system. However, if k or any of the v_i are large, this would be a very prohibitive strategy. If one makes the assumption that at most a small number of components can cause a fault, then one can achieve a much better bound in general; this is known as *combinatorial interaction testing* (CIT). Even under this assumption, forming a test suite with few tests is challenging.

Table 1 gives a real-world example of such a CIT test suite, which appears in [2]. Suppose we have a system that contains any of three browsers, three operating systems, three types of connections, and three types of connections to a printer; therefore, the system has four components, each of which can take on three values. In addition, suppose we know that a fault will not arise from at least three different components interacting, but there may be such a fault from two components. It is possible then to test this system using 9 tests, which is optimal.

Here, we consider a family of objects with a specific structure, with many applications including being ingredients to generate a desired test suite. The number of tests generated is often very competitive with existing methods, and the size of the objects in this family is often much smaller, allowing for much faster generation. Furthermore, we employ a strategy that is even faster, while also limiting the additional rows needed for the test suite. The domain for such a strategy is very large, so we propose using genetic algorithms (GAs) to search the space more effectively than other approaches. A simple implementation of a GA using standard operators and representation improved fitness from 59% of the theoretically optimal fitness to over 78% of this optimum (and the improvement for the estimated true optimal fitness is over 30%).

2 THE MODEL

For simplicity, we describe the model at a high level; for a formal treatment, see [5]. A *t-restriction* is an $N \times k$ array, where each element belongs to some domain. Furthermore, there is a set of *demands* provided of the form (X, Q), where X is a set of t-tuples of symbols, and Q is either $\exists$ or $\forall$. For every choice of t columns of the array, these columns *satisfy* all demands that involve them;

*Corresponding Author

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326823

Table 1: A test suite to cover all pairs of interactions of browser, OS, connection, and printer.

Test	Browser	OS	Connection	Printer
1	NetScape	Windows	LAN	Local
2	NetScape	Linux	ISDN	Networked
3	NetScape	macOS	PPP	Screen
4	IE	Windows	ISDN	Screen
5	IE	macOS	LAN	Networked
6	IE	Linux	PPP	Local
7	Chrome	Windows	PPP	Networked
8	Chrome	Linux	LAN	Screen
9	Chrome	macOS	ISDN	Local

by this, we mean that if (X, Q) is such a demand, and $Q = \exists$, then some element of X appears in some row restricted to these columns; otherwise if $Q = \forall$, then for any element x in X, some row equals x. A t-restriction is *existential* if $Q = \exists$ for all demands, and is *universal* if $Q = \forall$ for all demands. In a GA context, one example for the fitness of an individual, represented as the array, is the number of demands it satisfies.

Some notable examples of t-restrictions are covering arrays (CAs) [2] and perfect hash families (PHFs) [9]. The demand for a CA is that every choice of t symbols appears in some row, and a PHF requires some row having all distinct symbols. A CA is an analogue of the testing strategy described above, and a PHF is a common ingredient in the construction of CAs. To generate a desired CA with k components, a PHF with k columns is needed, along with another CA with fewer columns. Many generalizations of PHFs exist [7, 11], but we focus here on *Sherwood covering perfect hash families* (SCPHFs) [10]. Each entry in a SCPHF is a list of values, called a *permutation vector*, that expands into a (partial) column of a CA directly.

Much effort has been invested to determine the smallest size of these restrictions; see [2] for a survey on CAs, and [3, 8] for tables of best-known CA and PHF sizes. We only discuss PHFs and SCPHFs from here on, but the framework is general enough to allow application to any existential t-restriction. To obtain a solution with many columns, one can horizontally replicate the array multiple times; any t choices of columns that involve an original column at least twice do not satisfy the demand (along with other constraints for SCPHFs), so additional rows are needed to satisfy them; some techniques bound how many additional rows are needed [1, 4]. However, by changing the duplicates in a controlled way, we can dramatically reduce the number of unsatisfied demands.

3 AFFINE TRANSFORMATIONS

Colbourn and Lanus [6] consider *affine transformations* on SCPHFs in an attempt to reduce the number of such sets. An affine transformation (AT) is a *multiplier* μ and an *adder* α, such that if c is an observed value, then the AT maps c to $\mu c + \alpha$, with arithmetic done over the finite field. As an example, if there are 3 symbols, and if the AT has $\mu = 2, \alpha = 1$, then the value 2 under this AT will be $(2 \times 2 + 1)\%3$, which is 2.

Because the sample space is so large, they only use a greedy algorithm for finding appropriate transformations. Naturally, a different AT for each row of any existential t-restriction can be applied, because whether or not a demand is satisfied is only determined on a per-row basis (since each demand has $Q = \exists$); the set of demands for each duplicate will always be satisfied. One benefit of ATs is that they reduce the number of duplicate columns. What makes SCPHFs interesting is that ATs may not just be applied to each row, but independently for *each index in the permutation vectors*; so, we can associate each row and each index of the permutation vectors its own multiplier and adder. This dramatically increases the number of possible transformations. Unpublished work by the first author shows that a standard GA representation without use of affine transformations and a small population (50) can improve the number of columns for SCPHFs (see Table 2). Additionally, the same GA model against PHFs, with ATs, has an over 30% improvement in fitness with one minute of computation. We believe that by exploiting ATs, GAs can find more competitive existential restrictions than other methods can.

Table 2: Improvements to k for SCPHFs with $t = 7$ and 3 symbols.

# rows	2	3	4	5	6	7	8	9	10
$k_{previous}$	9	10	12	13	13	15	17	19	21
k_{new}	10	11	13	14	15	17	18	20	22

4 CONCLUSION

Testing interactions between components in software will always remain of high importance, and one of the main bottlenecks currently in generating competitively small test suites is the amount of computation needed. We expect that a collaboration between the Genetic Improvement and the interaction testing communities to investigate how a simple improvement in a small existential t-restriction can lead to large improvements in the test suites generated.

REFERENCES

[1] Renée C. Bryce and Charles J. Colbourn. 2009. A density-based greedy algorithm for higher strength covering arrays. *Software Testing, Verification and Reliability* 19, 1 (2009), 37–53.

[2] Charles J. Colbourn. 2004. Combinatorial aspects of covering arrays. *Le Matematiche (Catania)* 58, 1,2 (2004), 125–172.

[3] Charles J. Colbourn. 2005–2018. Covering Array Tables for t=2, 3, 4, 5, 6. (2005–2018). http://www.public.asu.edu/ ccolbou/src/tabby/catable.html.

[4] Charles J. Colbourn. 2008. Constructing perfect hash families using a greedy algorithm. *Coding and Cryptology* (2008), 109–118.

[5] Charles J. Colbourn. 2011. Covering arrays and hash families. *NATO Science for Peace and Security Series, D: Information and Communication Security* 29, Information Security, Coding Theory and Related Combinatorics (2011), 99–135.

[6] Charles J. Colbourn and Erin Lanus. 2018. Subspace Restrictions for Covering Perfect Hash Families. *Art of Discrete and Applied Mathematics* 1 (2018). Issue 2. #P02.03.

[7] Charles J. Colbourn and Jose Torres-Jimenez. 2010. Heterogeneous hash families and covering arrays. *Contemp. Math.* (2010), 3–15.

[8] Ryan E. Dougherty. 2017. Perfect Hash Family Tables for t=3 to 11. (2017). http://www.public.asu.edu/~redoughe/phf_pages/phf_tables.html

[9] Kurt Mehlhorn. 1982. On the program size of perfect and universal hash functions. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*. IEEE, 170–175.

[10] George B. Sherwood, Sosina S. Martirosyan, and Charles J. Colbourn. 2006. Covering arrays of higher strength from permutation vectors. *Journal of Combinatorial Designs* 14, 3 (2006), 202–213.

[11] D. R. Stinson, R. Wei, and L. Zhu. 2000. New constructions for perfect hash families and related structures using combinatorial designs and codes. *Journal of Combinatorial Designs* 8, 3 (2000), 189–200.