An Approach for Choosing the Best Covering Array Constructor to Use

Hanefi Mercan, Cemal Yilmaz, and Kamer Kaya
Faculty of Engineering and Natural Sciences, Sabanci University, Istanbul
{*hanefimercan,cyilmaz,kaya*}*@sabanciuniv.edu*

***Abstract*—Covering arrays have been extensively used for software testing. Therefore, many covering array constructors have been developed. However, each constructor comes with its own pros and cons. That is, the best constructor to use typically depends on the specific application scenario at hand. To improve both the efficiency and effectiveness of covering arrays, we, in this work, present a classification-based approach to predict the "best" covering array constructor to use for a given configuration space model, coverage strength, and optimization criterion, i.e., minimizing the construction time or the covering array size. We also empirically evaluate the proposed approach by using a relatively small, yet quite realistic space of application scenarios. The approach predicted the best constructors for reducing the construction times with an accuracy of 86% and the best constructors for reducing the covering array sizes with an accuracy 90%. When two predictions were made, rather than one, the accuracy of correctly predicting the best constructors increased to 94% and 98%, respectively.**

***Index Terms*—Combinatorial interaction testing, covering arrays, covering array constructors, predictions**

I. Introduction

Covering arrays systematically sample a given configuration (i.e., input or variability) space and test only the selected configurations. Given a *configuration space model*, which implicitly defines a valid configuration space for testing by expressing configuration options and their settings as well as the inter-option constraints (if any) that invalidate certain combinations of option settings, a t-way covering array is a set of valid configurations, in which each valid t-tuple appears at least once, where a t-tuple is an ordered set of option-setting pairs for a combination of t distinct options [1] [2]. In this definition, t is often referred to as the coverage strength.

Covering arrays have been successfully used in many domains, including systematic testing of input parameters [3], software configurations [4], software product lines [5], graphical user interfaces [6], multi- threaded applications [7], and network protocols [8]. Therefore, approaches for computing covering arrays in an efficient and effective manner are of great practical importance, which is also evident from more than 50 papers published on the subject [2].

In a number of studies we conducted on existing covering array constructors, we observe that each constructor typically comes with its own pros and cons. Some constructors are fast and scalable, but typically generate large covering arrays. Others generate small covering arrays, but typically are slow and does not scale up well. Furthermore, the efficiency (i.e., generating covering arrays faster) and the effectiveness (i.e., generating smaller covering arrays) of the constructors often depend on some important problem parameters, such as the coverage strength, the number of configuration options, the number of constraints, and the "length" of these constraints. Consequently, the best covering array constructor to use often typically depends on the specific application scenario at hand.

In this work we present a classification-based approach to predict the "best" covering array constructor to use for a given configuration space model, coverage strength, and optimization criterion, i.e., reducing the construction time or the covering array size. In the experiments we carried out, the proposed approach predicted the best constructors for

reducing the construction times with an accuracy of 86% and the best constructors for reducing the covering array sizes with an accuracy of 90%, where the accuracy was computed as the percentage of the correctly predicted best constructors. When two predictions were made instead one, the accuracy of having the best constructor among the predicted ones, was 94% and 98%, respectively.

The remainder of the paper is structured as follows: Section II provides background information about the existing covering array constructors used in this paper; Section III introduces the proposed approach; Section IV presents the experiments we carry out to evaluate the proposed approach; Section V discusses threats to validity; Section VI presents related work; and Section VII presents concluding remarks and possible directions for future work.

II. Background

Covering array constructors take as input a configuration space model $M =< O, V, Q >$. This model includes a set of configuration options $O = \{o_1, o_2, \ldots, o_k\}$, a set of settings for these options $V = \{V_1, V_2, \cdots, V_k\}$, where V_i is a discrete domain from which option o_i takes on a setting, and a system of inter-option constraints Q, which invalidates certain combinations of option settings. For this work, we express the constraints as a set of forbidden tuples, i.e., combinations of option settings that are not allowed to appear in any configuration, which is one of the well-known approaches for expressing constraints in combinatorial testing [9].

In the empirical studies presented in Section IV, we use five well-known covering array constructors, namely `CASA` [10] `ACTS` [11], `Jenny` [12], `PICT` [13], and `TCA` [14].

`CASA` uses simulated annealing [15] to compute covering arrays [10]. The standard simulated annealing loop is enhanced with an additional outer loop to determine the "minimum" size of the array. Note that the minimum size of a covering array cannot be determined with certainty in the general case. Therefore, constructors, especially the ones based on metaheuristic search, should locate a good approximation to the minimum size. The approach proposed in [10] judiciously chooses a new size at each iteration and feeds it to simulated annealing. Starting with an initial state of the given size, which is represented as a set of configurations, the annealing process repeatedly applies a series of alterations to the current state until the current state constitutes a t-way covering array. During the iterations, superior states (i.e., the ones with fewer number of missing t-tuples) are always accepted, whereas the the inferior states (i.e., the ones with more number of missing t-tuples) are probabilistically accepted to avoid getting stuck with a local optima. Furthermore, the probability of accepting an inferior state gradually decreases as the search proceeds.

`ACTS` uses the standard IPOG algorithm [16] as well as a number of variations of this algorithm [17] to construct covering arrays. The IPOG algorithm starts with a t-way covering array for the first t options and then repeatedly carries out a *horizontal growth* followed by a *vertical growth* in a loop until the array constitutes a t-way covering array. In a horizontal growth, a new option is added to array (i.e., the array is extended by a column) and the settings in the newly added column are decided in a greedy manner. After populating the new column, if there are still some missing t-tuples involving the options so far included in the array, a vertical growth is carried out, which adds additional rows to the array in a greedy manner until the current set of missing t-tuples are covered.

`Jenny` is another well-known covering array constructor. It uses a greedy approach [12]. In each iteration, a set of configurations are randomly generated and then the settings of the options present in the configurations are fine tuned in a greedy manner to reduce the number of missing t-tuples. Finally, among all the configurations generated, the one that covers the maximum number of previously uncovered t-tuples is included in the array. The iterations end when no t-tuples left uncovered.

`PICT` is developed with three main design decisions: 1) computing covering arrays faster, (2) ease of use, and (3) extensibility [13]. It uses a greedy algorithm. In each iteration, a configuration

that covers the "maximum" number of missing t-tuples is generated and added to the array of selections. PICT differs from Jenny in that it is a deterministic constructor and a single configuration is generated at each iteration, rather then multiple ones.

TCA is a covering array constructor that operates in two modes: *greedy mode* and *random mode* [14]. In the greedy mode, TCA employs an influential tabu search [18] in an attempt to optimize the objective function, whereas in the random mode, it adapts the random walk heuristic in an attempt to better explore the search space and diversify the search.

We opted to use these covering array constructors because 1) they are publicly available, well-known constructors, 2) they use a spectrum of different approaches to construct covering arrays, and 3) they all support configuration space models with inter-option constraints and varying numbers of settings for the configuration options.

III. Approach

In this paper we develop a classification-based approach to predict the "best" covering array constructor to use for a given application scenario. In particular, we take as input a configuration space model, a coverage strength, and an optimization criterion. The optimization criterion can either be to reduce the construction time or to reduce the covering array size. The former criterion is important when the cost of running a covering array is negligible compared to that of constructing the array. And the latter criterion is important when the cost of constructing a covering array is negligible compared to that of running the array.

A. Features

To train a prediction model and later to use this model to predict the best constructor to use, we extract four features: the coverage strength t, the number of configuration options k, the level of constraints L, and the constraint length $\bar{Q}$, which is indeed the number of unique options involved in a forbidden tuple.

All of these features, except for the first one, are extracted from the given configuration space model. We use the former two features, i.e., t and k, because the number of t-tuples to be covered grows exponentially with t and polynomially with k. Therefore, these parameters typically affect both the construction times and covering array sizes in a significant manner. The latter two features are used because in a number of studies we carried out, we observed that the efficiency and the effectiveness of the existing constructors often depend on these parameters at different levels and in different ways.

Furthermore, rather than using the actual number of constraints $|Q|$, we opt to use the constraint level L. This is because the effect of $|Q|$ depends on the configuration space under test, which necessitates that $|Q|$ needs to be normalized. The relationship between $|Q|$ and L is given below:

$$|Q| = \lfloor 2L\sqrt{k} \rfloor \tag{1}$$

We do this to train the prediction models in an unbiased manner. Since the number of valid configurations tends to decrease exponentially with $|Q|$, by making $|Q|$ proportional to the square root of k, we attempt to keep the decrease in the configuration space linear. That is, given the number of forbidden tuples $|Q|$, we compute L as described in equation (1) and use it, rather than $|Q|$, for predictions. Given a set of forbidden tuples, one can trivially determine the implicit forbidden tuples (if any) by using the algorithm presented in [19].

B. Training Phase

In this work we envision a service (e.g., a web service), which, given an application scenario, predicts the best covering array constructor to use. That is, the prediction model (i.e., the classification model) is trained by the service provider and once trained, the same model is used by all practitioners; practitioners are not required to train their own prediction models.

To train the prediction model, we opted to use the *full-factorial designs* [20]. The proposed approach operates as follows: 1) for each of the four features discussed in Section III-A, a minimum and a maximum value to be used for training are determined; 2) the range of values for each attribute is then discretized, such that the data required for training

the prediction models can be collected given the resources available; 3) each constructor is executed in every possible combinations of the discretized levels across all the features, and both the construction times and the covering array sizes are recorded; and 4) the data obtained is then used to train a separate classification model for the construction time and the covering array size, where for every experimental setup the best constructors are used as class labels.

The ultimate goal of this approach is to make reliable predictions within the space used for training, the border of which is determined by the minimum and maximum values of each feature. Since a single training model needs to be trained for all the practitioners and the data collection process can trivially be parallelized, we believe that such a prediction model can easily be trained for a large enough space to be practical. Furthermore, the amount of resources required for collecting the data can significantly be reduced, if *screening designs* [20], rather than full-factorial designs, are used. Screening designs are a class of highly economical experimental designs that can be used to identify the important features, i.e., combinations of feature settings that affect the construction times and/or the covering array sizes most, and then only these features can be used to train the prediction models.

The prediction model trained as described above suggests only one covering array constructor. Consequently, we call this model *best-constructor* model. To further experiment with the proposed approach and to provide developers with more options, we have also developed an approach to predict the second best constructors, which can indeed be trivially generalized to predict the n^{th} best constructors. We call these models *second-best-constructor* models.

The second-best-constructor models are trained as follows: For each constructor C, the constructor is completely removed from the training set. That is, the training set is updated as if nothing was known about C. For example, the experimental setups for which C was the best constructor are updated with the second best constructors for them. Thus, we end up having 5 new different training sets. The updated training sets are then used to create the second-best-constructor models for C.

These models are used for the experimental setups for which C was predicted to be the best constructor by the best-constructor model. Note that by removing C from the training set, C is prevented from being both the best and the second best constructor. Consequently, the number of second-best-constructor models we train is the same as the number of constructors used one model for each constructor.

C. Deployment Phase

In the deployment phase, the prediction models created in the training phase are used to predict the best and the second best constructors. To predict the best constructor, the features extracted from the given scenario are fed to the best-constructor model. To predict the second-best predictor, the second-best-constructor model trained in the absence of the best constructor predicted is used with the same features.

IV. Experiments

We carried out a series of experiments to evaluate the proposed approach.

A. Training Set

In these experiments we, as a training set, used all combinations of the settings over the four features described below:

- **Coverage strength (t):** We experimented with $t = \{2, 3\}$.
- **Number of configuration options (k):** We experimented with $k = \{20, 80, 140, 200\}$. The number of values that each option can take on was set to 2, 3, or 4 with an equal probability.
- **Constraint level (L):** We experimented with $L = \{0, 1, \cdots, 9\}$. Note that given L, the actual number of forbidden tuples is determined by using the equation (1).
- **Constraint length ($\bar{Q}$):** We experimented with $\bar{Q} = \{2, 3, 4, 5\}$.

We chose these settings, because we believe that they represent a space of quite practical scenarios,

the size of which is small enough, so that the experiments can be carried out in a timely manner. In particular, we randomly generated 5 different configuration space models for each of the 320 experimental setups (2 values of t $\times$ 4 values of k $\times$ 10 values of L $\times$ 4 values of $\bar{Q}$=320), executed all of the 5 covering array constructors described in Section II in each of these setups, and computed the average construction times and covering array sizes for every constructor. That is, in total, we executed the constructors 8000 times ($320 \times 5 \times 5$) and for every experimental setup, sorted the constructors by the increasing order of their construction times and covering array sizes.

B. Test Set

We used the training set described above to create the best-constructor and second-best-constructor models. To evaluate the proposed approach, on the other hand, we created a test set by using 50 different randomly chosen k values between 20 and 200. Note that the k values selected for the test set are not necessarily the same as the ones used in the training set. For each k, the coverage strength t, the constraint length $\bar{Q}$, and the level of constraints L are also determined randomly, such that they stay between the minimum and the maximum values of the features used in training. That is, the evaluation of the proposed approach was performed by using previously unseen data. We created 5 different configuration space models for each of these setups.

For each experimental setup in the test set, we used all the constructors to generate covering arrays for 5 configuration space models and computed the average of recorded the actual construction times and the actual covering array sizes. For evaluations, we compared the actual best constructors with the predicted best constructors (Section IV-D). The training and test sets used in this work can be found at TODO.

C. Operational Model

Not all the differences between the covering array constructors were meaningful in practice. Therefore, we considered two construction times as equal when they were within three seconds of each other. Similarly, we considered two covering array sizes as equal when they were within one configuration of each other. Furthermore, not all the constructors scaled up to the whole experimental design space. Thus, we put a 24 hours threshold. That is, if a constructor failed to create a covering array for a given experimental setup within 24 hours, we killed the constructor and marked the setup.

TABLE I
ACCURACY OF PREDICTING THE BEST CONSTRUCTORS.

Classifier	Accuracy for Best Time (%)	Accuracy for Best Size (%)
Decision Tree	86	76
SVM	74	64
Multilayer Perceptron	84	84
Naive Bayesian	84	74
Random Forest	86	90

To train the prediction models, we experimented with five different classification algorithms, namely Decision Trees [21], Support Vector Machines (SVMs) [22], Multilayer Perceptrons [23], Naive Bayesian [24], and Random Forests [25]. In particular, we used WEKA 3.8.1 [26] to create the prediction models.

All the experiments were carried out on a 20-core (2 sockets) Intel Xeon E5-2680 v2 2.80 GHz machine with 256 GB of RAM, running 64-bit CentOS 6.5 operating system.

D. Evaluation Framework

To evaluate the success of the proposed approach, we computed the accuracy of correctly predicting the best constructor for reducing the construction time and for reducing the covering array size, separately, for all the setups in the test set. We also repeated the the same analysis by using the second-best-constructor models. That is, for each setup in the test set, we predicted the best and the second best constructors and computed the accuracy of the actual best constructor being one of these two predictions.

TABLE II
STATISTICS ABOUT THE DIFFERENCES BETWEEN THE ACTUAL AND THE PREDICTED VALUES BY THE BEST-CONSTRUCTOR MODEL.

	Avg. Difference	Max. Difference
Time (in secs.)	22.7	678
Size (in configs.)	4.43	90

TABLE III
STATISTICS ABOUT THE BEST-CONSTRUCTOR AND THE SECOND-TO-BEST CONSTRUCTOR MODELS.

	Accuracy (%)	Avg. Difference	Max. Difference
Time	94	6.58	307.1
Size	98	0.16	7.8

E. Data and Analysis

Table I presents the accuracy of correctly predicting best constructors for reducing construction times (second column) and covering array sizes (third column) by using the best-constructor models obtained from different classification algorithms (first column). Although, almost all the classification algorithms performed fairly well, the best performer was Random Forest with the an accuracy of 86% (a weighted F-measure of 0.78) and 90% (a weighted F-measure of 0.90) for predicting the best constructor for reducing the construction times and the covering array sizes, respectively. The F-measures were computed by giving equal importance to precision and recall. Consequently, we use only Random Forest-based prediction models in the remainder of the analysis.

Table II, furthermore, presents the average and the maximum differences in the construction times (in seconds) and in the covering array sizes (in number of configurations) between the predicted and the actual best constructors. The lower the differences, the better the predictions are. They were 22.7 and 678 seconds for the construction times and 4.43 and 90 configurations for the covering array sizes, respectively.

To demonstrate that these results are not by chance, we generated 200 random predictions for each setup in the test set and computed the same metrics. The average and the maximum differences turned out to be 639.7 and 14569.4 seconds for the construction times and 18.4 and 112 configurations for the covering array sizes, respectively. Furthermore, the worst-case average and maximum differences were 2084.4 and 26137.9 seconds for the construction times and 39.4 and 130.3 configurations for the covering array sizes, respectively.

We then used both the best-constructor and the second-best-constructor models to make two predictions, rather than one, as described in Sections III-B and III-C. Table III presents the results we obtained. The accuracies were improved from 86% to 94% (a weighted F-measure of 0.88) for predicting the best constructors for reducing the construction times and from 90% to 98% (a weighted F-measure of 0.98) for predicting the best constructors for reducing the covering array sizes. Furthermore, the average and the maximum differences were dropped to 6.58 and 307.1 seconds and to 0.16 and 7.8 configurations, respectively, further emphasizing the success of the proposed approach.

V. THREATS TO VALIDITY

All empirical studies suffer from the threats to their internal and external validity. For this work, we were primarily concerned with threats to external validity since they limit our ability to generalize the results of our experiment to industrial practice.

A potential threat is the representativeness of the configuration space models used in the experiments. We plan to repeat the experiments by using a larger range of values for t, k, $|Q|$ (thus L), and $\bar{Q}$. However, we believe that the training set we used in this work, represents a space of quite practical scenarios, which is based on our experience with highly configurable systems. We, furthermore, systematically varied the coverage strength, the number of configuration options, the level of constraints, and the length of the constraints.

A related concern is that we used a full-factorial design as our training set. As a future work, we plan to use screening designs [20] to improve the efficiency of the proposed approach. This highly economical class of designs can be used to determine a small number of factors that affect the

construction times and/or the covering array sizes the most and then the prediction models can be constructed by using these factors.

Another potential threat is that we used only five covering array constructors. However, they are all well-known and frequently-used covering array constructors. Further justifications for using them can be found in Section II.

VI. Related Work

The covering array construction algorithms can be classified into four main groups: greedy algorithms, metaheuristic search-based algorithms, mathematical methods, and random search-based methods [2].

Greedy algorithms [27], [28] construct covering arrays by iteratively choosing the "best" configuration at every iteration, which covers the "maximum" number of previously uncovered t-tuples. Metaheuristic search-based approaches, such as hill climbing [29], great flood [30], and simulated annealing [31], [32] start from an array of configurations and iteratively apply a set of alterations to the array until all the valid t-tuples are covered. One good thing about these approaches is that they employ mechanisms to avoid getting stuck with local optima as much as possible. Random search has also been used for constructing covering arrays [33]. These techniques randomly select valid configurations from the configuration space until the selected configurations constitutes a covering array. In addition, several mathematical approaches [34], [35] have been proposed for computing covering arrays.

Constraint handling is a problem extensively studied in combinatorial interaction testing. Bryce et al. [36] present an approach for handling "soft constraints." Hinc et al. [37] propose a technique for handling "hard constraints". Cohen et al. [38] demonstrates that not handling constraints often leads to wasted testing resources.

VII. Concluding Remarks and Future Work

Many covering array constructors exist. However, each constructor typically comes with its own pros and cons. That is, the best constructor to use generally depends on the specific application scenario. To improve both the efficiency and effectiveness of covering arrays, we, in this work, presented a classification-based approach to predict the "best" covering array constructor to use for a given configuration space model, coverage strength, and optimization criterion. We then empirically evaluated the proposed approach by using a relatively small, yet quite realistic space of application scenarios. The approach predicted the best constructors for reducing the construction times with an accuracy of 86% and the best constructors for reducing the covering array sizes with an accuracy 90%. When two predictions were made, rather than one, the accuracies increased to 94% and 98%, respectively.

As future work, we plan to expand our covering array constructor database by adding new covering array constructors [39] and train the prediction models by using a larger training set. We also plan to make the prediction models publicly available.

VIII. Acknowledgments

This research was supported by the Scientific and Technological Research Council of Turkey (113E546).

References

[1] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The aetg system: An approach to testing based on combinatorial design," *Software Engineering, IEEE Transactions on*, vol. 23, no. 7, pp. 437–444, 1997.

[2] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[3] G. Rothermel, S. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 13, no. 3, pp. 277–331, 2004.

[4] C. Yilmaz, M. B. Cohen, A. Porter *et al.*, "Covering arrays for efficient fault characterization in complex configuration spaces," *Software Engineering, IEEE Transactions on*, vol. 32, no. 1, pp. 20–34, 2006.

[5] M. F. Johansen, Ø. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference-Volume 1*. ACM, 2012, pp. 46–55.

[6] X. Yuan, M. B. Cohen, and A. M. Memon, "Gui interaction testing: Incorporating event context," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 559–574, 2011.

[7] Y. Lei, R. H. Carver, R. Kacker, and D. Kung, "A combinatorial testing strategy for concurrent programs," *Software Testing, Verification and Reliability*, vol. 17, no. 4, pp. 207–225, 2007.

[8] B. Stevens and E. Mendelsohn, "Efficient software testing protocols," in *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1998, p. 22.

[9] D. R. Kuhn, R. Bryce, F. Duan, L. S. Ghandehari, Y. Lei, and R. N. Kacker, "Chapter one-combinatorial testing: Theory and practice," *Advances in Computers*, vol. 99, pp. 1–66, 2015.

[10] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *Search Based Software Engineering, 2009 1st International Symposium on*. IEEE, 2009, pp. 13–22.

[11] L. Yu, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Acts: A combinatorial test generation tool," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 370–375.

[12] B. Jenkins, "jenny: A pairwise testing tool," http://www.burtleburtle.net/bob/index.html, 2005.

[13] J. Czerwonka. Pairwise testing in the real world: Practical extensions to test-case scenarios. [Online]. Available: http://msdn.microsoft.com/en-us/library/cc150619.aspx

[14] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "Tca: An efficient two-mode meta-heuristic algorithm for combinatorial test generation (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 494–505.

[15] B. Stevens, "Transversal covers and packings," Ph.D. dissertation, University of Toronto, 1998.

[16] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog: A general strategy for t-way software testing," in *14th Annual IEEE Int. Conf. and Workshops on the Eng. of Computer-Based Systems (ECBS'07)*. IEEE, 2007, pp. 549–556.

[17] M. Forbes, J. Lawrence, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Refining the in-parameter-order strategy for constructing covering arrays," *Journal of Research of the National Institute of Standards and Technology*, vol. 113, no. 5, p. 287, 2008.

[18] K. J. Nurmela, "Upper bounds for covering arrays by tabu search," *Discrete applied mathematics*, vol. 138, no. 1, pp. 143–152, 2004.

[19] L. Yu, F. Duan, Y. Lei, R. N. Kacker, and D. R. Kuhn, "Combinatorial test generation for software product lines using minimum invalid tuples," in *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*. IEEE, 2014, pp. 65–72.

[20] L. Eriksson, E. Johansson, N. Kettaneh-Wold, C. Wikström, and S. Wold, "Design of experiments," *Principles and applications*, pp. 172–174, 2000.

[21] S. L. Salzberg, "C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993," *Machine Learning*, vol. 16, no. 3, pp. 235–240, 1994.

[22] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[23] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," DTIC Document, Tech. Rep., 1985.

[24] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752. Citeseer, 1998, pp. 41–48.

[25] T. K. Ho, "The random subspace method for constructing decision forests," *IEEE transactions on pattern analysis and machine intelligence*, vol. 20, no. 8, pp. 832–844, 1998.

[26] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[27] J. Czerwonka, "Pairwise testing in the real world: Practical extensions to test-case scenarios," in *Proceedings of 24th Pacific Northwest Software Quality Conference, Citeseer*, 2006, pp. 419–430.

[28] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog/ipog-d: efficient test generation for multi-way combinatorial testing," *Software Testing, Verification and Reliability*, vol. 18, no. 3, pp. 125–148, 2008.

[29] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 38–48.

[30] R. C. Bryce and C. J. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1082–1089.

[31] M. B. Cohen, C. J. Colbourn, and A. C. Ling, "Augmenting simulated annealing to build interaction test suites," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. IEEE, 2003, pp. 394–405.

[32] J. Torres-Jimenez and E. Rodriguez-Tello, "New bounds for binary covering arrays using simulated annealing," *Information Sciences*, vol. 185, no. 1, pp. 137–152, 2012.

[33] M. Harman, "The current state and future of search based software engineering," in *2007 Future of Software Engineering*. IEEE Computer Society, 2007, pp. 342–357.

[34] A. Hartman, "Software and hardware testing using combinatorial covering suites," in *Graph Theory, Combinatorics and Algorithms*. Springer, 2005, pp. 237–266.

[35] A. W. Williams and R. L. Probert, "Formulation of the interaction test coverage problem as an integer program," in *Testing of Communicating Systems XIV*. Springer, 2002, pp. 283–298.

[36] R. C. Bryce and C. J. Colbourn, "Prioritized interaction testing for pair-wise coverage with seeding and constraints," *Information and Software Technology*, vol. 48, no. 10, pp. 960–970, 2006.

[37] B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith, "Constraint models for the covering test problem," *Constraints*, vol. 11, no. 2-3, pp. 199–219, 2006.

[38] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *Software Engineering, IEEE Transactions on*, vol. 34, no. 5, pp. 633–650, 2008.

[39] J. Czerwonka, "Pairwise testing: Available tools," http://www.pairwise.org/tools.asp.