Ruth Kissel and Ryan Dunham

COSC 480

Professor Dougherty

05/07/2020

Multiple Column IPO Algorithms

Introduction

For our project, we further explored the IPO strategy, or in-parameter-order algorithm. The IPO algorithm revolves around the idea of creating a larger covering array from a smaller one, by choosing a column that covers a maximal number of interactions. Rows are then added to the covering array if uncovered interactions exist. This augmentation is achieved by horizontal growth, which adds a new column to the covering array and chooses the value in each row of this new column strategically, maximizing uncovered interactions. However, in the traditional IPO algorithm, during the horizontal growth stage, only one column is added to the covering array at a time [1]. Additionally, horizontal growth is followed by vertical growth when some interactions remain uncovered by the array. Rows are added to the array to cover these uncovered interactions.

The IPO algorithm has seen great success in the field of combinatorial testing, due to its efficiency and ability to produce covering arrays of minimal size [2]. However, most work done on IPO algorithms has focused on the different strategies used to choose parameter values, such as greedy or random methods, and the order in which they are filled; but little work has been done exploring the effect of adding multiple columns at once.

For these reasons our research is centered around this topic; adding multiple columns at a time instead of one column during the horizontal growth stage. We aimed to produce results by first exploring whether adding two columns at a time produces smaller covering arrays, and then extending the IPO 2 algorithm for adding three, four, and five columns at a time. We attempt to understand the effect of adding multiple columns on the size of the covering array produced.

Related Work

Lei and Tai's paper titled "In-Parameter Order: A Test Generation Strategy for Pairwise Testing," provides an overview of the IPO strategy, created due to the NP-completeness of generating a minimum pairwise test [1]. This paper was applied to our research and helped us to generate our research topic going forward.

More specifically, previous work has also utilized various implementations, specifically the IPOG algorithm. This was introduced in a paper written by Lei et al. whose work focuses on a more generalized version of the IPO strategy [3]. In this particular research, they wanted to first develop a testing strategy that can be applied to general software applications. Because of this criteria the goal is to, ultimately, put no restrictions on the system configuration under test, which favors computation over an algebraic approach. The second motivation for this IPOG strategy is that generally *t*-way testing has a rigid demand on the time and space requirements than pairwise testing does; relating to the fact that the number of combinations grows exponentially as the strength of the coverage increases. This paper then provides us with the framework of the IPOG strategy which is as follows: for a system with *t* or more parameters, IPOG constructs a *t*-way test set for the first *t* parameters, extends the test set to construct a *t*-way test set for the first $t + 1$ parameters, and then continues to extend the test set until it constructs a *t*-way test set for all the parameters [3]. This extension of an existing *t*-way test set for an additional parameter is done using horizontal and vertical growth, which was discussed earlier.

Another paper written by Forbes et al. seeks to explore the horizontal growth stage, which is deemed as the more important of the two. The best approach for choosing values in horizontal growth seems to be a greedy method, outlining an algorithm that allows for greedy selection over both the row and value with which we extend the array [4]. Greedy methods have proven to produce minimal size covering arrays compared to using other methods such as random or density. Thus, being able to increase the optimality of the results, which in turn will decrease runtime.

What We Did

Using the IPOG algorithm as described in Lei et al. [3] as our baseline, we sought to extend this algorithm for multiple column horizontal growth. We began our research by implementing our own basic IPO algorithm that was based on the pseudo code given in the IPOG paper using Python [3]. The algorithm takes three inputs, the strength of the covering array t, the number of parameters k, and the number of values v. Given these inputs, the algorithm constructs an initial covering array from which the IPO algorithm builds on. This initial covering array is constructed exhaustively, with the number of columns equal to the strength of the covering array t. The remaining k-t columns are to be filled by the IPO algorithm. We then add a random shuffle to the rows of the initial covering array to add variation. This has proven to be effective in minimizing the overall size of the final covering array produced. The algorithm then iterates through the k-t columns, and augments them one at a time. In each new step we create a dictionary that stores the parameter interactions as keys and a list of all possible combinations of values as their value. This data structure allows us to track which interactions remain uncovered during the horizontal and vertical growth stage. During the horizontal growth stage, we begin by considering each row in order. We then generate new augmented candidate rows for each row, and choose the candidate that covers the greatest number of uncovered interactions. Once the best candidate row is chosen, the interactions it covers are removed from the interactions dictionary and the covering array is updated to contain this new column value. Once every row has been augmented, we consider vertical growth. If the interaction dictionary is empty, we do not need to proceed with vertical growth, as every combination of values exist for each interaction of t parameters. If it is not empty, uncovered interactions exist and rows must be added to the covering array to account for these values. The vertical growth algorithm considers each uncovered interaction. If there is a row that exists in which the parameters of the uncovered interaction have don't care values or the values specified in the uncovered interaction, then augment this row to fully account for the interaction, filling the don't care positions with the parameter values. If a row like this does not exist, then we create a row in which the parameters of the uncovered interaction are filled with their respective values, and the rest of the parameters are filled with don't care values. Once we have covered every interaction, the remaining don't care positions are filled randomly, and the algorithm considers the next row to be augmented. Once every row has been augmented, a covering array is produced.

After creating our baseline IPO algorithm that performed variable strength covering array generation, we sought to modify the horizontal growth stage by adding multiple columns at once instead of just one. The overall structure of the algorithm remains very similar to the baseline; the main differences lie in the horizontal growth stage and iterating over the columns. In IPO 2, we iterate over the k-t remaining columns, two at a time. If k-t is an odd number, we use IPO 2 until we get to the last column, and use our baseline IPO algorithm to fill the last column. IPO 3, 4 and 5 work in a similar fashion. We use their specific number of columns to augment the covering array as much as possible, but if k-t is not a multiple of the IPO variant, then lesser methods must be used to fill the remaining columns. For instance, if k-t is 19 and we are using the IPO 5 algorithm. We use IPO 5 to fill the first 15 of the k-t columns and then use IPO 4 to fill the remaining 4 columns. During the horizontal growth stage, we still consider rows in order, but generate candidate rows that have 2, 3, 4 or 5 columns appended. This increases the size of the search space of the horizontal growth algorithm, and leads to a slower runtime for the IPO variants. Once the horizontal growth stage augments all rows accordingly, the covering array is updated and we consider vertical growth.

We created a baseline IPO algorithm based off of IPOG [3], as well as several multiple column IPO variant algorithms. These variant algorithms include IPO 2, IPO 3, IPO 4, and IPO 5. Once we successfully implemented each algorithm, we began testing to see if the amount of columns added during the horizontal growth stage affected the overall size of the final covering array produced. The results and analysis from our testing are shown below.

Results

Figure 1 shows our initial results when we ran our IPO and IPO 2 algorithm for 1 iteration.

Figure 1. Here *k* = 10:50

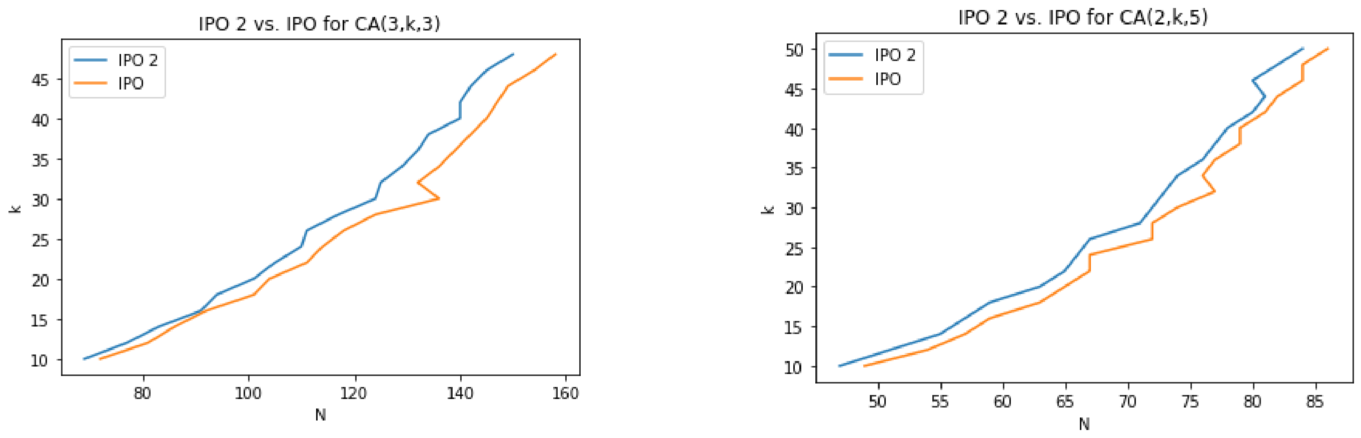

Table 2, below, then shows our results when the same algorithms ran, but this time for 10,000 iterations. We then extended our results to include IPO 3, 4, 5, and those results are shown in Figure 2.

IPO Algorithm	**IPO 2 Algorithm**
CA [3, 10, 3] Avg N. 71.547 Minimum 65 CA [2, 10, 3] Avg. N: 19.8707 Minimum N: 15	CA [3, 10, 3] Avg. 70.0987 Minimum N 65 CA [2, 10, 3] Avg. N: 18.8329 Minimum N: 15

Table 1. Algorithms ran for 10,000 iterations

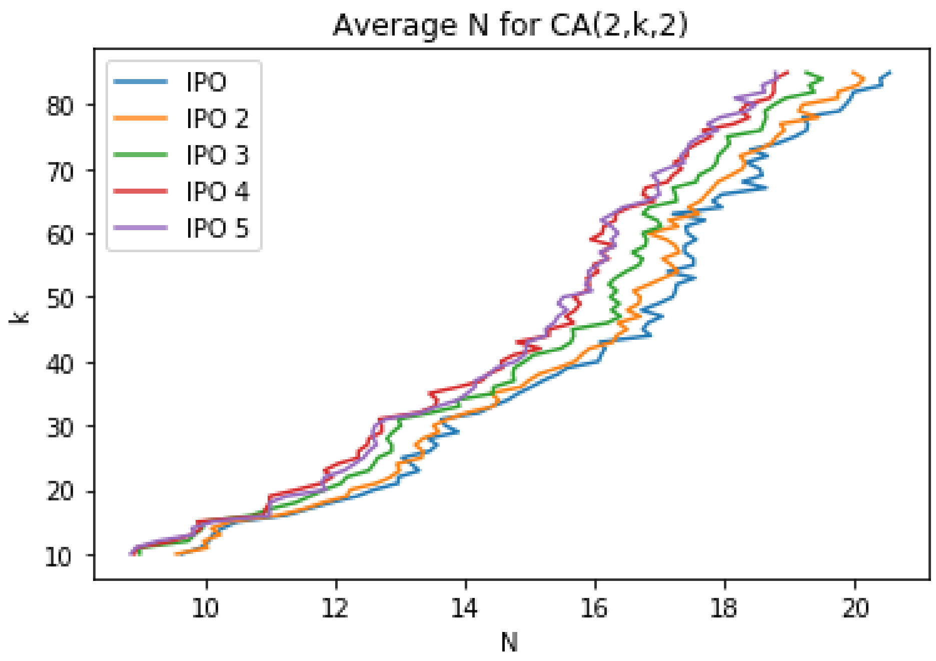

Figure 2. 50 iterations for $t = 2$, $k = 10{:}86$, $v = 2$

The results that we received from IPO 1, IPO 2, IPO 3, IPO 4, and IPO 5 are further shown in Table 2. Here we looked to gain more insights into the relationships between these varying algorithms, hoping to come to a more significant conclusion, as we will discuss later on. Table 2 shows the results from three covering arrays, with t=2, v=2 and differing *k* values.

CA(2, 10, 2)			**CA(2, 12, 2)**			**CA(2, 16, 2)**		
IPO	**Min N**	**Avg N**	**IPO**	**Min N**	**Avg N**	**IPO**	**Min N**	**Avg N**
1	9	9.673	1	9	10.0992	1	11	11.2384
2	9	9.6542	2	9	10.0306	2	11	11.1471
3	9	9.0	3	9	9.7499	3	10	10.7482
4	8	8.9369	4	9	9.4945	4	10	10.8972
5	8	8.8821	5	9	9.244	5	11	11.0

Table 2. Ran each IPO algorithm for 10,000 iterations

The last part of our results is regarding how IPO 1, 2, 3, 4, and 5 perform with larger N values. Table 3 shows the minimum N and average N produced.

IPO	**Min N**	**Avg N**
1	27	29.69
2	27	29.08
3	26	28.19
4	26	27.56
5	25	27.18

Table 3. For CA(2, 30, 3)

IPO	**Min N**	**Avg N**
1	12	13.49
2	12	13.15
3	12	12.81
4	11	12.38
6	11	12.52
8	12	13.26
12	16	16.04

Table 4. 100 iterations for CA(2, 26, 2)

Discussion of Results

The results of our testing and analysis point to several significant conclusions. In our first figure, it shows the relationship between IPO 2 and IPO for just one iteration, for $t = 3$, $k = 10:50$, $v = 3$ and $t = 2$, $k = 10:50$, $v = 5$. It appears that IPO 2 is consistently producing smaller size covering arrays than IPO for the same t, k, and v. Although this result appears to be very promising, IPO algorithms require several runs to find the smallest covering array, as there is a significant amount of random variation in the algorithm. We were not sure whether IPO 2 was producing smaller covering arrays or was just more efficient at finding smaller covering arrays. Table 1, thus shows our results from 10,000 iterations of our IPO 2 and IPO algorithms. These results allowed us to come to more concrete solutions regarding the two. From these results it is evident that IPO 2, on average, produces lower N values or smaller covering arrays. While IPO 2 is more efficient at finding smaller covering arrays, it does not find a minimal covering array. After 10,000 iterations, both IPO and IPO 2 found the same minimum sized covering array for the respective values of t, k and v. Although IPO will find the same exact absolute minimum sized covering array, it requires far more iterations to do so. This becomes even more pronounced for covering arrays with a large amount of columns and a higher strength. The IPO algorithm may require millions of iterations to find the absolute minimum, whereas the IPO 2 may require only hundreds or thousands. The increased efficiency provided by the IPO 2 algorithm is thus important and substantial.

Motivated by these results we extended our testing to include the IPO 3, 4, and 5 algorithms, where three, four, or five columns are added to the covering array at a time during the horizontal growth stage. These results are shown in Figure 2. This figure shows the average N for IPO 1, 2, 3, 4, and 5 for 50 iterations. Here $t = 2$, $k = 10{:}86$, and $v = 2$. For IPO 1, 2, and 3, as the k value increases the N does too. But with each successive IPO algorithm the average size covering array starts to get smaller, with IPO 1 producing the largest average size covering array. At first, all 5 IPO algorithms produce a similar average sized covering array, as there is significant overlap. As k increases, the gap between the algorithms begins to widen. The higher order IPO algorithms outperform the lower order IPO algorithms. Although IPO 5 seems to outperform IPO 4 on most values of k, there is a significant amount of overlap. We will explore this finding further.

The results produced in Figure 2 are clarified when we run these algorithms for 10,000 iterations, as shown in Table 2. In Table 2, you can see that on a CA(2, 10, 2), IPO 4 and IPO 5 do produce smaller covering arrays after 10,000 iterations, which is very significant. With the minimum N for IPO 4 and IPO 5 being 8, compared to 9 for IPO 1 through 3. As you add more columns, the average N also decreases. But this may not always be the case. In the CA(2, 16, 2), it appears that IPO 4 and IPO 5 do not perform as well as IPO 3. This result was very interesting because it differed from our previous results and initial thoughts.

Through further testing, we realized that some IPO algorithms work better for certain sized covering arrays. We found that if the difference between *k* and *t* was a multiple of four or five, then IPO 4 or IPO 5 would work better. For example, if we have CA(2,17, 2), since 17 minus 2 is 15, and 5 is a multiple of 15, then IPO 5 would be the most optimal. Compared to CA(2, 16, 2), in Table 2, where 16 minus 2 is 14, and 5 is not a multiple of 14. That is why we see here that IPO 3 performs better. As *k* gets larger, though, this tends to disappear, but for these smaller sized covering arrays where *k* is between 5 and 20, we found this to be the case. This ultimately led us to realize that our method, IPO 4 and IPO 5, really works the best when *k* is very large.

To explore the effect of covering array size on the effectiveness of IPO variant algorithms, we tested a covering array whose *t-k* value had many factors. We tested IPO variants of size 1, 2, 3, 4, 6, 8, and 12 to see which performed best on CA(2,26,2). Since all of these IPO variants are factors of t-k or 24, we thought it would be interesting to see which performed best. After 100 iterations of each algorithm, IPO 4 performed best, as shown in Table 4. While it is important that the IPO variant is a factor of *t-k*, the largest factor does not always produce the smallest average sized covering array. There seems to be some ratio based on the number of columns that determines which IPO variant will prove most effective. This should be explored through further work.

This brings us to Table 3, where we have results from a large N, CA(2, 30, 3). Here we can identify that the gaps between the means are getting greater, showing us that as we get to these larger covering arrays, adding more columns at a time is more optimal. It produces smaller covering arrays, more efficiently.

Generally, these multiple column IPO algorithms are more efficient than the standard IPOG algorithm for finding a minimum sized covering array. Additionally, as the size of the covering array increases, IPO variants also become more effective. However, there does seem to be a limit on the effectiveness of these algorithms. As apparent throughout our results, IPO 5 is not always more efficient than IPO 3 and IPO 4. It really depends on the values of *t*, *k*, and *v*, so for smaller covering arrays sometimes IPO 1 or IPO 2 would work better than IPO 5.

Future Work

Going forward, we acknowledge that further experimentation is needed to be able to fully come to a more conclusive understanding, specifically more efficient algorithms. If we were to continue our work, we would wish to do so in the following ways. First we would like to further optimize our IPO algorithm and observe its continued effect on the covering array size. This would be achieved by implementing a better method for filling “don’t care” positions in the vertical growth stage. Our algorithm fills “don’t care” positions randomly, which does not always result in the best choice of values for a covering array of minimal size. In some cases a better choice of “don’t care” values could result in fewer rows being added in the next round of vertical growth.

Our IPO algorithm could also be optimized by greedily choosing the order in which to augment the rows during horizontal growth. It was clear after reviewing the literature on IPO algorithms that the method for choosing the order in which the rows are augmented can be very effective at producing minimal size covering arrays. Currently during horizontal growth, our algorithm considers the rows to augment in order. We begin with the first row and add an optimal column, then the second row, and so forth. A better method is to choose these rows using a greedy method, where whichever augmented row covers the most uncovered interactions is updated first. This approach has proven to be effective in producing a covering array with fewer rows [4]. Although this method for considering rows produces minimal sized covering arrays, it does significantly impact runtime. The search space of the horizontal growth algorithm grows exponentially using this method, leading to a slower, less efficient algorithm. The increased runtime could be a detriment to testing, especially when using a slower language like Python.

Given these improvements that allow us to produce more optimal sized covering arrays, it would be interesting to see if the effect of multiple column growth still remains. Would the increase in efficiency provided by multiple column growth be diminished by the increased efficiency provided by the strategic filling of "don't care" positions or using a greedy method to consider the order of rows? Further research is needed to answer these questions.

Conclusion

In this paper, we have shown the varied effects of multiple column IPO variants. Stemming from our initial research into the IPO algorithm and the baseline for our implementation, we aimed to explore the effects of adding multiple columns to a covering array at a time. Once we found significant results from two columns at a time, we then extended our algorithm for three, four, or five columns. This varied from the standard IPO algorithm where only one column is added during the horizontal growth stage. This work on multiple column variants proved to provide us with compelling results. Our multiple column IPO algorithms appear to be more efficient at finding minimal sized covering arrays than the baseline IPOG algorithm. We have shown that as covering array size increases, particularly the number of columns k, our IPO variants become more effective. However the effectiveness of the IPO variant depends on the specific *t*, *k* and *v* values.

References

[1] Y. Lei and K.-C. Tai, “In-Parameter-Order: a Test Generation Strategy for Pairwise Testing,” *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No.98EX231)*, Washington, DC, USA, pp. 254–261, Nov. 1998.

[2] S.-W. Gao, J.-H. Lv, B.-L. Du, C. J. Colbourn, and S.-L. Ma, “Balancing Frequencies and Fault Detection in the In-Parameter-Order Algorithm,” *Journal of Computer Science and Technology*, vol. 30, no. 5, pp. 957–968, 2015.

[3] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, “IPOG: A General Strategy for T-Way Software Testing,” *14th Annual IEEE International Conference and Workshops on The Engineering of Computer-Based Systems (ECBS07)*, pp. 549–556, 2007.

[4] 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, pp. 287–297, 2008.

[5] S. Gao, J. Lv, B. Du, Y. Jiang and S. Ma, "General Optimization Strategies for Refining the In-Parameter-Order Algorithm," *2014 14th International Conference on Quality Software*, Dallas, TX, 2014, pp. 21-26.