A Specialized Evolutionary Algorithm for Generating Tight Single-Change Covering Designs

Matt Zykan, M.S. Computer Science UMR

I. Abstract

A problem-specific evolutionary algorithm, a generic EA, and a non-genetic heuristic method for generating tight single-change covering designs are presented and compared. None of these methods succeed in outperforming existing algorithms, but their behavior does reveal properties of the problem's solution space.

II. Introduction

A. Problem Statement

A tight single-change covering design (TSCCD) is an ordered set of ordered sets of integers, or in other words, a sequence of "blocks". The size and complexity of a TSCCD are determined by two parameters: v, the total number of unique integers, or "symbols", and k, the number of symbols in each block. A valid TSCCD consists of a sequence of blocks which satisfies three properties. First, each successive block must have all symbols in common with the preceding block except for any single symbol which must differ. This is the "single-change" element. Second, all unique pairs of v symbols must be represented, or "covered", by the constituent symbols' appearance together in at least one block. Third and finally, for each single changed symbol from one block to the next, all pairs formed by the new symbol's introduction to the block must not be formed in any preceding block. This is the "tight" element of the problem. TSCCDs are deceptively complex constructions, and they are not easy to generate in non-trivial sizes.

B. Performance Goal

The key objective of this work is the development of an evolutionary algorithm which can generate TSCCDs with significantly higher efficiency than existing methods. The design will be considered successful if the EA is capable of generating TSCCDs with $(v = 20, k = 5)$ on common computer hardware in a run time of less than 24 hours. To date, solutions of this size have only been found using a deterministic algorithm running for several weeks.

III. Related Work

Straightforward though laborious techniques have been developed for generating TSCCDs by Preece et al. [1], [4] Their work includes in-depth analysis of the mathematical properties of the problem and its solutions, as well as several TSCCD solutions of various sizes. A stochastic algorithm of the type proposed here has little in common with the techniques used to generate these solutions, so the EA developed relies on none of these techniques. Philips [3] presents additional analyses of solution properties. The most important information taken from these papers is a table which appears in [1] listing values of v and k for which a complete TSCCD can be found, up to and including $(v = 21, k = 6)$.

Evolutionary computing has been previously applied to this problem in Johnson [2]. That approach is different from the presented work in two important respects: it used multi-objective algorithms, which will not be applied here, and the genome there includes every symbol of every block of the construction, whereas the presented work will store only the single-change for each block. Storing only the single-change dramatically reduces the search space by intrinsically satisfying the single-change property. At length, their algorithm was able to produce a correct sequence for $(v = 20, k = 5)$ up to 87% of the full solution length, with the remaining few blocks left undefined.

IV. Specialized Evolutionary Algorithm

The MOEA presented in [2] was not designed specifically for the TSCCD problem, that is, no problem-specific strategies were used to generate the TSCCD results. The goal of this design is to improve on the MOEA's performance by exploiting specific characteristics of the problem.

Here, the genome of an individual is a sequence of single changes, and this is accompanied by non-genetic information required to interpret the alleles. Individuals in the population may have genomes of various lengths, generally all shorter than what is needed to form a complete TSCCD. Each generation, the algorithm creates a full-length TSCCD by deterministically choosing individuals as needed and appending their alleles to the solution in progress as needed. Once complete, the TSCCD is evaluated and the fitnesses of contributing individuals are updated. Offspring are generated by taking random subsequences of the composite genome. Rather than search for a single individual with a full, correct solution in its genome, this algorithm searches for a set of individuals which can be assembled into a correct solution. This bears resemblance to a Co-EA, but there are no Co-EA mechanisms explicitly persued here.

A. Representation

The primary genome consists of an ordered set of single-change "moves", so-called because the construction of a TSCCD in-order can be considered a kind of puzzle game. A move is simply a transformation to be applied to a block, such that a TSCCD with n blocks can be represented completely as a single initial block and a sequence of $n-1$ moves. A move (gene) consists of a tuple (m_i, m_s), indicating the index into the block and the symbol to be placed there, respectively. To accompany the sequence of moves, each individual also stores one complete block of symbols, the "lead-in", indicating the context for which the genome is appropriate. The lead-in is not a part of the genome proper, as it does not have any genetic operations applied to it. The utility of this lead-in is detailed in section IV-B2. The population consists of individuals with genomes of various lengths, with all genomes being no longer than a complete solution. Individuals also carry their fitness value and "relevance", a value used for survivor selection as detailed in section IV-B5.

B. Iteration

An iteration of the EA begins by constructing a single full-length solution, first by generating a random initial block, and then by choosing individuals from the population to append as needed to build a full sequence of moves. Each move appended to the solution is assigned a "score" $(p_{new} - p_{dup} - p_{err})$, with the values p equal to the number of new pairs formed and covered, the number of pairs formed that were already covered, and the number of invalid pairs formed, respectively. The only type of invalid pair possible here is of the form (s_i, s_i), where a symbol is paired with itself. The score of the entire solution is the sum of the move scores plus the number of pairs in the initial block, $kC2$.

1) Choosing the Next Individual: Individuals are chosen from the population by tentatively applying their sequences of moves to the current construction and then choosing the individual which promises to contribute most positively to the total score. If several individuals promise to contribute the same best score, the one with highest fitness is chosen. A new individual must be chosen whenever the current individual being applied has exhausted its sequence of moves. The last individual to be chosen typically will not contribute its entire genome, as construction is halted when the solution reaches the known full length.

2) Applying an Individual's Moves: Whether contributing to the construction or being tentatively evaluated, an individual's alleles are transformed according to the individual's lead-in block before being evaluated. Most of the time, the trailing block of the construction is not identical to the prospective individual's lead-in. In these cases, a 1-to-1 mapping of the v symbols is captured which transforms the lead-in block into the construction's trailing block. This same mapping is then applied to the symbol m_s of each move before it is added to the construction. The individual's genome is not modified here, it is only transformed for application to the construction.

3) Updating Contributing Individuals: Once construction is complete, the individuals which contributed have their fitness and relevance updated. Fitness is based on a running average of the total score contributed by the individual to constructions for which it has been chosen. Relevance is simply an integer indicating for which construction the individual was last chosen. A tally is kept of how many constructions have been performed, and contributing individuals have their relevance value set to this tally. With this value, obsolete or irrelevant individuals can be identified.

4) Creating Offspring: New individuals are formed from random segments of the full solution. A random segment is chosen by taking two uniform random values over the indices into the full sequence of moves. The offspring's lead-in is set equal to the block which immediately precedes this range. For example, if an offspring's genome includes the very first move, its lead-in will be the random initial block which started the construction, furthermore, it is impossible for the last block in the construction to become an offspring's lead-in. The lead-in and moves are taken directly from the construction, where the contributing individuals' moves have been remapped. The offspring's fitness is set according to the scores for the moves which it includes, and its relevance is set to the current construction tally. Finally, mutation is applied. Each individual may be mutated in one of three ways: a random move is omitted or added, a random move has its row index randomly reset, or a random move has its symbol randomly reset. One problem with this method is that the initial fitness value of an individual is very unlikely to be accurate, as it is calculated before mutation. This is somewhat countered by the fact that the value will be corrected by averaging in subsequent generations. Also, this fitness value is not the most dominant selection criteria.

5) Survivor Selection: The number of individuals in the population is constant, as is the number of offspring generated from each construction. All offspring are added to the population, and the population is then truncated to its proper size by removing individuals with the smallest (oldest) relevance values. This means that all offspring are almost guaranteed to survive at least one generation, as long as the number of offspring generated is less than the population size.

C. Termination

The algorithm terminates as soon as it performs a construction which results in no errors or duplicate pairs.

V. Simple Evolutionary Algorithm

A simple EA was implemented for comparison to the more complex approach. This simple EA uses a genome design almost identical to the complex EA, with genes representing moves. The main difference here is that all individuals have full-length genomes, such that a complete TSCCD can be constructed from one individual. Fitness is defined simply as the number of pairs covered by the TSCCD generated from an individual's genome. Parent selection is fitness proportional, survival selection is done by 3-tournament, and population size and children produced per generation are fixed parameters. Mutation is done by uniform random resetting, and crossover is uniform random.

VI. Impractical Stochastic Algorithm: Collapser

The third and final approach is not an evolutionary algorithm, but is based on deterministically updating the weights in weighted sets of moves. Each feasible block has a set of all possible moves associated with it. The algorithm repeatedly forms full-length TSCCDs by randomly choosing moves in sequence from the weighted set of moves corresponding to the lastmost block of the in-progress solution. The error (number of repeated pairs) caused by each move is then used to negatively adjust the weight of that move in its respective set. This algorithm is impractical for solving hard TSCCDs because it must store a weight for every possible move for every possible block. Even using small data types to store the weights, it is not practical to apply this algorithm to a TSCCD larger than $(v = 20, k = 5)$ without using specialized computing hardware, as the memory requirement is simply too extreme. Even so, this algorithm reveals important characteristics of this problem's solution space.

A. *Stored Data*

The Collapser algorithm stores many weighted sets of all possible moves, one set for each feasible block. The number of stored weight values is then the number of feasible unique blocks times the number of possible moves, or $(kv^{(k+1)})$. For $(v = 20, k = 5)$ this means there are more than 300 million weights to be stored; this is very near the limits of a modern workstation, making this the maximum practical TSCCD size for this algorithm. An attempt was made to improve the memory requirement by storing weights only for moves which appeared at least once and thus had non-default weight values, but the algorithm explores the solution space so wildly that such data structures quickly inflate to their maximum size.

B. *Iteration*

Iteratively, the algorithm uses the weighted sets to randomly build full-length TSCCDs. An initial block is generated, then the first move is chosen from the set of moves corresponding to that particular block. This forms a second, usually different block, and the second move is then chosen from that block's corresponding set, and so on until a full length solution is built. The solution is evaluated, and if it is correct the algorithm terminates. Otherwise, for each move which caused errors the weight of that move in its member set is reduced proportional to the number of invalid pairs. Additionally, if the overall solution is superior to the best solution found so far, all moves involved, erroneous or not, have their weight updates positively biased. With weights updated a new random construction is attempted, and this process is repeated. This is not guaranteed to converge to a correct solution, so the implementation is set to reinitialize after a certain number of iterations without any improvement on the best-yet solution.

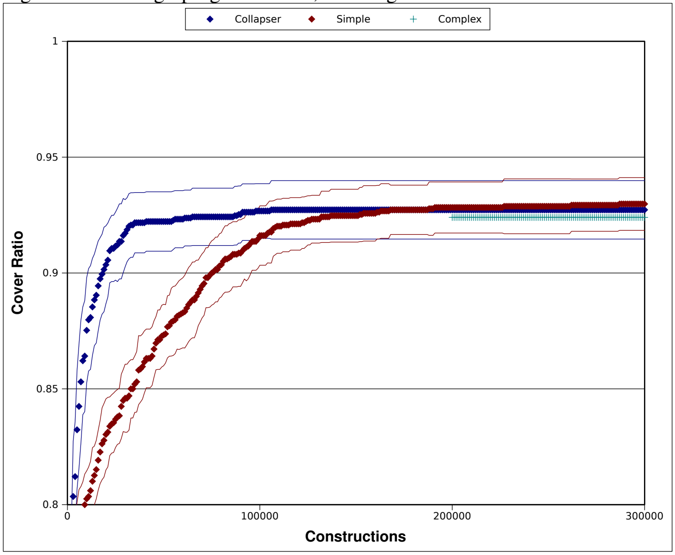
VII. Experimental Setup

Experimental results are limited as, unfortunately, this effort very clearly failed to produce an algorithm which can exceed the performance of existing methods. TSCCD parameters used for comparison are $(v = 12, k = 4)$. This problem is much harder than those with $k < 4$, but it is not excessively difficult.

VIII. Results

All three algorithms produced similarly suboptimal results, leaving efficiency as the only interesting performance measure. The following figure compares all three algorithms. The complex EA's typical final result is shown, but its trend is not shown due to difficulty in determining a fair "constructions" metric. The complex EA requires several times the run time of the two simpler approaches to achieve a similar result, so the small portion of data should be considered far-far-right of the shown horizontal range.

Figure 1. Average progress trends, showing one standard deviation



IX. Conclusions

None of these methods can efficiently generate non-trivial TSCCDs.

Ignoring the extreme memory requirement, the Collapser method is the best of the three, achieving suboptima of quality similar to the other two algorithms using far fewer calculations. Of course, the memory requirements make this approach impractical.

The primitive EA seems just as effective as the problem-tuned EA, but it requires far fewer calculations to achieve similar suboptimae. Precise comparison of efficiency is difficult because the problem-tuned EA uses much more fitness evaluation and uses it in a unique way, but it is clear that the simple EA is far more efficient.

The similarity of effectiveness across these three algorithms suggests that the TSCCD solution space is very complex indeed, having a great many suboptimal "traps". This is evident in the fact that the non-genetic algorithm can produce TSCCDs which are more than 90% correct ad nauseam, at the rate of several unique suboptimae per minute.

This suggests that any EA designed to generate TSCCDs will benefit from as much flexibility in the solution space as possible in order to facilitate escape from suboptimal traps. The results here do not prove that the type of representation used cannot be used to implement an effective TSCCD generator, but nor do they suggest any advantage to reducing the solution space in this way.

X. Relevant Publications

References

[1] D.A. Preece, Tight Single-Change Covering Designs. Utilitas Mathematica, 47: pp. 55-84 (1995)

[2] Matt David Johnson, The Stored Non-Domination Level Multi-Objective Evolutionary Algorithm. (2007)

[3] N.C.K. Phillips, D.A. Preece, Tight single-change covering designs with v=12, k=4. Discrete Mathematics, 197/198: pp. 657-670 (1999)

[4] N.C.K. Phillips, Finding tight single-change covering designs with v=20, k=5. Discrete Mathematics, 231: pp. 403-409 (2001)