

STRAW FROM THE HAYSTACK: UNIFORMLY SAMPLING SOLUTIONS FOR SWEETPEA DESIGNS

by
Benjamin Richard Draut

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Department of Computer Science
The University of Utah
April 2019

Copyright © Benjamin Richard Draut 2019
All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Benjamin Richard Draut**
has been approved by the following supervisory committee members:

Matthew Flatt, Chair(s) **TODO**
Date Approved

Vivek Srikumar, Member **TODO**
Date Approved

Jonathan Cohen, Member **TODO**
Date Approved

none, Member **TODO**
Date Approved

none, Member **TODO**
Date Approved

by **Ross Whitaker**, Chair/Dean of
the Department/College/School of **Computer Science**
and by **Richard Brown**, Dean of The Graduate School.

ABSTRACT

SweetPea, a language for experimental design, attempts to generate approximately uniformly sampled trial sequences by relying on the aid of the SAT-sampler `unigen` to provide uniformity guarantees. In practice, this approach does not scale to support realistic experimental designs. The current positional encoding scheme embeds every allowed permutation of trial sequences as a unique solution. Therefore, the solution space grows factorially in the sequence length. Solution spaces of such magnitude are beyond the capabilities of current SAT-samplers to handle. This thesis presents evidence demonstrating the need for an alternative and investigates an alternative encoding, based on a bijection from natural numbers to valid trial sequences.

TODO

CONTENTS

ABSTRACT	iii
LIST OF FIGURES	vii
LIST OF TABLES	viii
CHAPTERS	
1. MOTIVATION	1
1.1 Introduction	1
1.2 Current Implementation	2
1.2.1 Boolean Satisfiability	2
1.2.2 Example: Stroop Experiment	3
1.3 Problems	4
2. CURRENT BENCHMARKS	6
2.1 Tool Overview	6
2.1.1 CryptoMiniSat	6
2.1.2 Unigen	6
2.1.3 KUS	7
2.1.4 Spur	7
2.2 Experiments	7
2.3 Benchmarks	8
2.3.1 CryptoMiniSat	8
2.3.2 Unigen	8
2.3.3 KUS	8
2.3.4 Spur	8
2.4 Distribution of Samples	8
2.4.1 Unigen	11
2.4.2 KUS	11
2.4.3 Spur	11
2.4.4 CryptoMiniSat	11
2.5 Summary	11
3. A COMPLEXITY SPECTRUM FOR SWEETPEA DESIGNS	16
3.1 Factors and Windows	16
3.1.1 Factor Types	16
3.1.2 Window Construction	17
3.2 Complexity Gradation	18
3.2.1 Level 1	18
3.2.2 Level 2	18

3.2.3	Level 3.....	18
3.2.4	Level 4.....	19
3.2.5	Level 5.....	19
4.	SOLUTION COUNTING.....	20
4.1	Principles of Counting	20
4.2	Partition of an Experimental Design	21
4.3	Formula	22
4.4	Example.....	23
5.	SEQUENCE CONSTRUCTION	25
5.1	Enumerating Permutations and Combinations	25
5.1.1	Interval Mapping	25
5.1.2	Permutations	26
5.1.3	Combinations	26
5.2	Construction Algorithm	28
5.2.1	Interval Mapping for an Experimental Design	28
5.2.2	Mapping Selected Values to Levels	29
5.3	Example.....	30
5.4	Handling Constraints	30
5.5	Benchmarks.....	31
6.	FUTURE WORK	33
7.	CONCLUSION	35
	REFERENCES	36

LIST OF FIGURES

2.1	Distribution of Sequences Sampled by Unigen	12
2.2	Distribution of Sequences Sampled by KUS	13
2.3	Distribution of Sequences Sampled by Spur	14
4.1	Partition of an Experimental Design	24
5.1	Decomposing a Number into Constituent Ranges	27
5.2	Construction of a Permutation from its Inversion Sequence	27
5.3	Selecting Valid Source Combinations	32

LIST OF TABLES

2.1	Benchmark Experiments	9
2.2	Time to Solve for 1 Solution with CryptoMiniSat	9
2.3	Time to Generate 10 Samples with Unigen	9
2.4	d-DNNF Compilation for Sampling with KUS	10
2.5	Time to Generate 10 Samples with KUS	10
2.6	Time to Generate 10 Samples with Spur	10
5.1	Benchmark Experiments	32
5.2	Sampling Metrics	32

CHAPTER 1

MOTIVATION

SweetPea [3], a language for experimental design, attempts to generate approximately uniformly sampled trial sequences by relying on the aid of a SAT-sampler to provide uniformity guarantees. In practice, this approach does not scale to support realistic experimental designs. This thesis presents evidence demonstrating the need for an alternative and investigates other approaches to overcome this problem. These include a different encoding scheme as well as techniques for direct construction of a solution based on its numerical index in the sequence of all solutions. These improvements significantly improve SweetPea’s capability to guarantee a uniform sampling of trial sequences.

1.1 Introduction

For any valid design configuration, there exists some set of trial sequences that conform to the design. Typically this set is extremely large, as the size grows factorially in the trial sequence length of the design. The magnitude of the solution space for experimental designs has posed a significant challenge for scientists desiring to conduct unbiased research.

At present, scientists patch together different scripts and tools in various ways to generate experiment trial sequences. This method, in addition to being brittle, does not facilitate sharing, nor does it produce unbiased sequences, making it nearly impossible for peers to replicate results. SweetPea provides a high-level declarative language for describing all aspects of an experimental design, providing standard tooling where before there was none. In addition, with the aid of a SAT-sampler, it generates approximately uniformly sampled trial sequences from the space of all sequences conforming to the design. SweetPea is composed of a Python library, which provides the language constructs, and a server that generates and processes the design using a SAT-sampler.

At the highest level, an experimental design is composed of a *block*, whose properties may vary depending on its type. At present, only a single block type is provided by

SweetPea, which is the *FullyCrossBlock*. A *FullyCrossBlock* is composed of a set of *factors*, a set of *constraints*, and a *crossing*. A *factor* is a collection of *levels*, or values to which the factor may be set. In a given trial, each factor takes the value of one of its levels. Levels may be values determined by the designer, or they may be derived from the values of other factors in the design. A *constraint* constrains the experiment in some regard. For example, the designer may wish to prohibit a particular level from being repeated in a trial sequence. Lastly, the *crossing* identifies the factors whose levels should be combined (crossed) in all possible ways in each trial sequence. The output of a design is a set of trial sequences. A single level value from each factor comprises a *trial*.

1.2 Current Implementation

SweetPea works by encoding the experimental design in a boolean satisfiability (also known as SAT) formula. such that any satisfying assignment constitutes a trial sequence that conforms to the design. Currently, a simple positional encoding is used, in which every level of every factor of every trial is represented as a distinct variable.

1.2.1 Boolean Satisfiability

The Boolean satisfiability problem (abbreviated SAT) is the problem of determining whether or not a Boolean formula can be satisfied, or be caused to evaluate to true, by a particular true or false assignment to each of its variables. If a problem can be expressed as a Boolean formula, then techniques from satisfiability theory may be applied to determine whether or not a solution exists to the problem, or to locate a particular solution. Many different SAT solvers exist which accept a Boolean formula as input, and output either that the formula is *unsatisfiable*, or a solution to the formula.

Tools for interacting with SAT formulae typically require the input formula to be in *conjunctive normal form* (abbreviated CNF), which is a form of a Boolean formula with specific properties. A Boolean formula is in conjunctive normal form only when three conditions are met. First, the entire formula must be a conjunction of clauses. Second, each clause consists of an individual variable, or a disjunction of variables. Third, negation may only be applied to individual variables. Any Boolean formula may be expressed in conjunctive normal form, though it usually requires growth in some dimension. SweetPea

uses the Tseitin Transformation [8] to do this conversion, which has the tightest known bounds on formula length, at the expense of allocating additional variables. A SAT formula contains some number of boolean variables and propositional logic clauses. A selection of true or false for each variable in the formula constitutes an *assignment*. If a given assignment causes the entire formula to be true, then the formula is said to be *satisfied* by that assignment, or more generally, the formula is *satisfiable*. If no such assignment exists, the formula is said to be *unsatisfiable*.

1.2.2 Example: Stroop Experiment

The Stroop test [?] is a well-known design among experimental psychologists in which a subject's reaction time is influenced by perceived conflicts in the trial sequence. For example, a subject may be shown printed words representing colors and asked to read the word. However, some of the time the font color of the word will be different than the printed word itself, leading to a delayed reaction due to the conflict. This design is easily represented in SweetPea as follows:

```
color = Factor("color", ["red", "green", "blue"])
text  = Factor("text",  ["red", "green", "blue"])
```

A FullyCrossBlock with these two factors will yield a sequence of $3 \cdot 3 = 9$ trials. A research may want to constrain the design to ensure that there are never two consecutive trials in which the color and text are the same, or congruent. Such a constraint requires the addition of another factor:

```
congruent = Factor("congruent?", [
    DerivedLevel("yes", WithinTrial(operator.eq, [color, text])),
    DerivedLevel("no",  WithinTrial(operator.ne, [color, text]))
])
```

With the factor now in place, a constraint may be specified:

```
AtMostKInARow(1, ("congruent?", "yes"))
```

The current encoding scheme will allocate 72 variables to represent each level of each

factor in each trial. Additional variables will be allocated to enforce other constraints, such as ensuring that each factor takes only one level in each trial, and requiring that each crossing be present in the sequence. The first step of the encoding can be visualized easily:

Trial		color			text			congruent?		
#		red	green	blue	red	green	blue	yes	no	

	1	1	2	3	4	5	6	7	8	
	2	9	10	11	12	13	14	15	16	
	3	17	18	19	20	21	22	23	24	
	4	25	26	27	28	29	30	31	32	
	5	33	34	35	36	37	38	39	40	
	6	41	42	43	44	45	46	47	48	
	7	49	50	51	52	53	54	55	56	
	8	57	58	59	60	61	62	63	64	
	9	65	66	67	68	69	70	71	72	

For example, if variable 68 is true, that corresponds to the level red being selected for the factor text in trial number 9. The encoding will also require variables 69 and 70 to be false in this case, as well as require variable 71 to be equivalent to 65. Once a solution is found, it is then decoded back into the original language of the design as a trial sequence. For more details concerning SweetPea's syntax or the current encoding scheme, the reader is referred to the original paper.

1.3 Problems

SweetPea sounds panacean, but it has yet to fully realize its objectives. It relies *fully* on a SAT-sampler to approximately uniformly sample some number of solutions to the generated formula. In practice, this works wonderfully for *small* experimental designs, where *small* means designs in which the trial sequence length is less than ten. However, realistic experimental designs are significantly larger, often with trial sequence lengths in the tens or hundreds.

A solution to the formula represents a unique trial **sequence** in the design. Therefore every possible sequence permutation (so long as it upholds the design constraints) is also a distinct solution. Consequentially, the number of solutions grows factorially in the

sequence length. In the above example, the sequence length, denoted l , was 9. Assuming the congruence constraint were removed, there would be $9! = 362,880$ solutions, which is still reasonable. Consider an experiment however in which $l = 36$. The solution count grows to $36! \approx 2^{138}$, which is intractable.

Sequence Length (l)	#SAT ($l!$)
4	24
6	720
9	362880
12	479001600
16	20922789888000
20	2432902008176640000
25	15511210043330985984000000
30	265252859812191058636308480000000
36	371993326789901217467999448150835200000000

In practice, designs will be constrained, however the constraints only exclude specific patterns in the permutations, therefore the number of solutions remains dominated by $l!$. This growth rate very quickly overwhelms all tested SAT-samplers. Therefore, the current strategy to fully relying on a SAT-sampler to pick apart the entire solution space is flawed. Although the total number of variables and clauses in the formula is reasonable, the overall number of solutions is not.

CHAPTER 2

CURRENT BENCHMARKS

This chapter introduces several experiment designs and provides benchmarks for their performance as measured using the current approach to sampling. These results should convince the reader of the need for an alternative approach to the problem, as the current method cannot cope with the scale of the solution spaces in question.

2.1 Tool Overview

Multiple tools are being used or have been considered for different aspects of SweetPea with varying degrees of success. This section introduces each tool explaining briefly how it works and how it is being used in the current implementation.

2.1.1 CryptoMiniSat

CryptoMiniSat [5, 7] is the main SAT solver that SweetPea relies on for generating non-uniformly sampled trial sequences. SweetPea achieves this with CryptoMiniSat by using the naive approach of solving for any solution, and then constraining the original formula to disallow that particular solution from being produced a second time. Due to the difficulty encountered while attempting to use SAT samplers, it has proven quite valuable to provide researchers with this ability to quickly generate some number of trial sequences conforming to their design, even though they may not be uniformly distributed. CryptoMiniSat is also used internally by Unigen. Lastly, in addition to logical AND, OR, and NOT clauses, CryptoMiniSat also supports XOR clauses, although SweetPea has not yet put these to use. This is an opportunity for improvement.

2.1.2 Unigen

Unigen [2] is a recently developed SAT *sampler*. Given a boolean formula in conjunctive normal form (CNF), Unigen can produce nearly uniformly sampled satisfying

assignments. SweetPea relies directly on Unigen for uniformly sampling trial sequences for a given design. The SweetPea encoding is designed such that there is a one-to-one relationship between solutions to the formula, and unique trial sequences that conform to the design. Unigen is based on a hashing technique that partitions the solution space and then samples solutions from each partition. Unigen has worked well for some designs, but its capacities are quickly overwhelmed by even modest designs.

2.1.3 KUS

KUS [6] is another SAT sampling tool developed by some of the same team behind Unigen. However, it approaches the problem quite differently. KUS requires a compiled deterministic decomposable negation normal form (d-DNNF) of a formula in conjunctive normal form. There are known polynomial time techniques for many operations on the d-DNNF representation of a boolean formula, including model counting. KUS takes advantage of these techniques to uniformly sample solutions using this form. This technique offloads much of the complexity to the d-DNNF compiler, rather than the sampling phase. KUS, while able to sample quickly, is not used in SweetPea due to scaling problems in compiling the formula to d-DNNF. Though the number of variables and clauses in the CNF for a formula is typically modest, compiling the equivalent d-DNNF has proven intractable.

2.1.4 Spur

Spur [?] is a more recent SAT sampler that is based on reservoir sampling. In benchmarks, it performs better than Unigen in nearly all cases. However, it does require an exact model count (based on sharpSAT), while an approximation suffices for Unigen. SweetPea does not rely on Spur at present, as it was developed after SweetPea's inception. However it has been considered as it represents the latest research for SAT sampling.

2.2 Experiments

Each of these tools was used to solve or sample solutions to formulae representing several experimental designs of increasing complexity comparable to realistic experiments written by current researchers. Table 5.1 enumerates the characteristics of each design, including properties of its CNF encoding and number of solutions.

2.3 Benchmarks

Unless otherwise noted, all sampling benchmarks were configured to generate 10 samples. All benchmarks were done on this machine:

2.3.1 CryptoMiniSat

Table 2.2 denotes the the time required for CryptoMiniSat to compute a single solution to the SAT formula for each experiment.

2.3.2 Unigen

Table 2.3 denotes the time required for Unigen to generate 10 samples for each experiment.

2.3.3 KUS

Table 2.4 denotes the time required to compile the CNF for each experiment to d-DNNF using the d4 compiler, as well as the resulting d-DNNF file size compared to the original CNF file.

For the experiments for which a d-DNNF file could be successfully compiled, table 2.5 denotes the time required to sample solutions using KUS.

2.3.4 Spur

Table 2.6 denotes the time required to generate samples for each experiment using Spur.

2.4 Distribution of Samples

In order for the uniformity guarantees of sampling tools to tranlate back into the problem space of the original experiment design, it is critical that the encoding scheme maintain a one-to-one relationship between formula solutions and distinct trial sequences. Without this relationship, it would be possible to uniformly sample solutions to the SAT formula while still introducing bias in the generated trial sequences. For example, if three SAT solutions yielded trial sequence S_1 , while only one solution yielded trial sequence S_2 , uniformly distributed solutions to the SAT formula would produce S_1 more frequently

Table 2.1. Benchmark Experiments

Experiment Name	Sequence Length	Variables	Clauses	Total Solutions
stroop-2	4	392	1,559	12
stroop-3	9	1,965	8,531	151,200
stroop-4	16	3,848	16,451	$\approx 2^{44}$
stroop-5	25	10,326	45,991	$\approx 2^{83}$
stroop-congruency-balanced	17	5,570	24,215	$\approx 2^{115}$
stroop-response	33	16,242	72,679	$\approx 2^{220}$
padmala-pessoa	37	116,289	559,080	$\approx 2^{201}$
task-switching	32	14,246	62,700	$\approx 2^{275}$
task-switching-2	146	742,832	3,672,632	$\approx 2^{990}$
task-switching-cue-switching	1,025	-	-	$\approx 2^{8778}$

Table 2.2. Time to Solve for 1 Solution with CryptoMiniSat

Experiment Name	Time to Solve
stroop-2	4.9 ms
stroop-3	9.1 ms
stroop-4	18.9 ms
stroop-5	64.4 ms
stroop-congruency-balanced	23.9 ms
stroop-response	232 ms
padmala-pessoa	3.38 s
task-switching	140 ms
task-switching-2	200 s
task-switching-cue-switching	-

Table 2.3. Time to Generate 10 Samples with Unigen

Experiment Name	Time to Sample
stroop-2	-
stroop-3	8.11s
stroop-4	Timed Out
stroop-5	Timed Out
stroop-congruency-balanced	Timed Out
stroop-response	Timed Out
padmala-pessoa	Timed Out
task-switching	Timed Out
task-switching-2	Timed Out
task-switching-cue-switching	-

Table 2.4. d-DNNF Compilation for Sampling with KUS

Experiment Name	Time	d-DNNF File Size	Original CNF File Size
stroop-2	14.4ms	9.7K	24K
stroop-3	1.99s	6.3M	151K
stroop-4	Timed Out	-	303K
stroop-5	Timed Out	-	881K
stroop-congruency-balanced	48.6m	7.3G	447K
stroop-response	Timed Out	-	1.5M
padmala-pessoa	OOM	-	13M
task-switching	Timed Out	-	1.2M
task-switching-2	"Indeterminate"	-	92M
task-switching-cue-switching	-	-	-

Table 2.5. Time to Generate 10 Samples with KUS

Experiment Name	Time
stroop-2	92.7 ms
stroop-3	2.35 s
stroop-4	-
stroop-5	-
stroop-congruency-balanced	OOM
stroop-response	-
padmala-pessoa	-
task-switching	-
task-switching-2	-
task-switching-cue-switching	-

Table 2.6. Time to Generate 10 Samples with Spur

Experiment Name	Time to Sample
stroop-2	21.9 ms
stroop-3	3.72 s
stroop-4	Timed Out (10 hrs)
stroop-5	Timed Out
stroop-congruency-balanced	OOM
stroop-response	Error
padmala-pessoa	Timed Out
task-switching	Timed Out
task-switching-2	Error
task-switching-cue-switching	-

than S_2 .

Although it has not yet been proven that this relationship is maintained, practical experience indicates that it is so. In this section we'll examine the distribution of the trial sequences produced by Unigen, KUS, and Spur, providing empirical evidence that the resulting trial sequences are uniformly distributed.

We sampled 10,000 trial sequences for the stroop-3 experiment using Unigen, KUS, and Spur. The resulting samples appeared to be uniformly distributed, each sample only occurring once for the most part.

2.4.1 Unigen

Unigen sampled 10,010 trial sequences, 9,965 of which were unique. Numbering each unique trial sequence in the order that it appeared and generating a histogram gives a visual confirmation that the sequences are approximately uniform. There were 333 sequences that were selected twice, and only 6 that were selected three times. No sequences were selected more than three times.

2.4.2 KUS

KUS sampled 10,000 trials sequences, all of which were unique.

2.4.3 Spur

Spur sampled 10,000 trial sequences, 9,664 of which were unique. 332 sequences were selected twice, and 2 were selected three times. No sequences were selected more than three times.

2.4.4 CryptoMiniSat

In contrast to the previous tests, when a SAT solver is used to generate individual samples incrementally by adding negated solutions to the prior formula, the same samples are generated each time. We used CryptoMiniSat to generate 10,000 samples as well, in groups of 100. Exactly the same set of 100 samples was generated each time.

2.5 Summary

None of the current tools for sampling SAT formulae are capable of handling the scale of practical experimental designs. Although the number of variables and clauses in the



Figure 2.1. Distribution of Sequences Sampled by Unigen

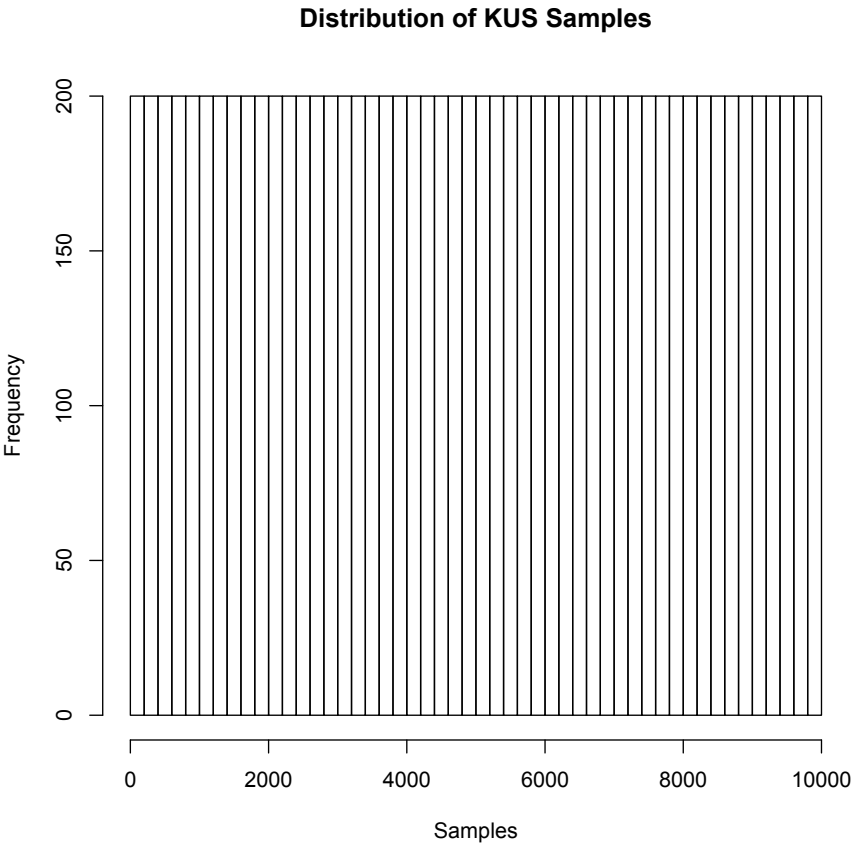


Figure 2.2. Distribution of Sequences Sampled by KUS

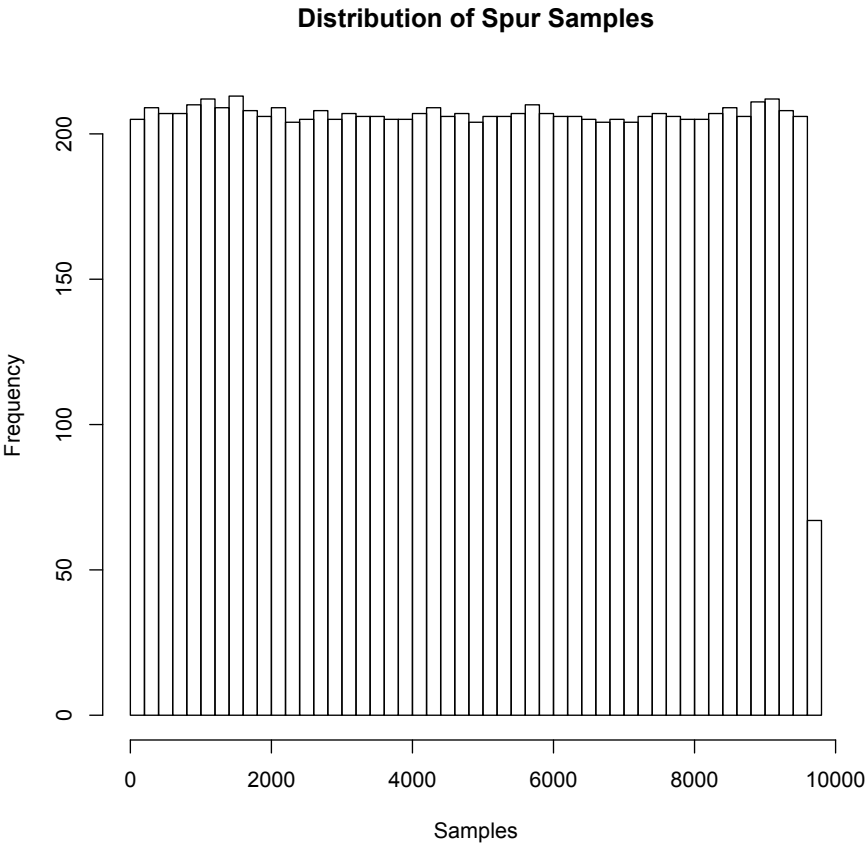


Figure 2.3. Distribution of Sequences Sampled by Spur

CNF for these experiments is reasonable, the factorial explosion in the number of solutions to the formulae overwhelmed each tool that we tested.

CHAPTER 3

A COMPLEXITY SPECTRUM FOR SWEETPEA DESIGNS

The complexity of SweetPea designs greatly varies depending on the user's requirements. At one end of the spectrum are simple designs, in which there are no structured relationships between trials in a sequence. At the other end are the most complex designs, in which multiple structured relationships exist between multiple trials in a sequence in varying intervals. This chapter will present a precised gradation of 6 complexity levels for experimental designs in order to facilitate future discussion. Recall that a complete experimental design is defined by a block, composed of a set of factors, a crossing of some subset of the factors, and a set of constraints. This section considers the complexity of a design based solely upon the factors present. Constraints on the overall design are not included in this gradation, as their complexity depends primarily on the complexity of the factors that they constrain.

3.1 Factors and Windows

3.1.1 Factor Types

Only two types of factors exist in SweetPea: *Basic* and *Derived*. The levels of a basic factor are literal values. Basic factors form the foundation of all designs. The following example declares a basic factor named `direction`, whose levels are the directions of the compass:

```
direction = Factor("Direction", ["North", "South", "East", "West"])
```

On the other hand, the levels of a derived factor depend upon values from other factors in the design. The congruency characteristic in the Stroop experiment is expressed as a derived factor:

```

congruency = Factor("Congruency", [
  DerivedLevel("Congruent", WithinTrial(lambda c, t: c == t, [color, text])),
  DerivedLevel("Incongruent", WithinTrial(lambda c, t: c != t, [color, text]))
])

```

The level of the congruency factor is determined by the current levels of the color and text factors, which will vary for each trial in a generated sequence. `WithinTrial` denotes the *window* of the derived level, which brings us to the next subject.

3.1.2 Window Construction

The construction of a derived level is based on the concept of a *window*, which slices the trial sequence into segments which are considered to determine the current level of a derived factor. A window's dimensions are defined by a *width* and *stride*. The *width* indicates how many preceding trials, including the current trial, are to be considered. The *stride* indicates how many trials to skip when moving on to the next trial for this factor.

The simplest type of window, for which `WithinTrial` is an alias, has a width and stride of one. This type of window does not span multiple trials. It only considers the values of basic factors within the same trial.

A slightly more complicated window has a width of two, but still a stride of one. (Aliased by the Transition window.) This type of window considers two adjacent trials for the purpose of inspecting the transition (or lack thereof) between them. For example, a Transition window would be appropriate when seeking to determine if a particular level was repeated in sequence.

```

direction_repeats = Factor("Direction Repetition", [
  DerivedLevel("Repeat", Transition(lambda directions: directions[0] == directions[1], [
  DerivedLevel("No Repeat", Transition(lambda directions: directions[0] != directions[1], [
])

```

The most complicated windows have arbitrary widths and strides. For example, a window with a width of one and a stride of four would consider the value of factors only at every fourth trial in the sequence. For example:

```

direction_is_north = Factor("Direction is North", [

```

```

DerivedLevel("Direction is North",      Window(lambda direction: direction == "North", [di
DerivedLevel("Direction is not North", Window(lambda direction: direction != "North", [di
])

```

Notice that windows with non-one width and stride will not have a value for every trial in the sequence. Generally speaking, a window with width w and stride s will first apply to trial w , and to every s th trial thereafter.

3.2 Complexity Gradation

Now understanding the types of factors and windows that can be used to define experimental designs, we can establish the complexity levels with which to categorize different designs.

3.2.1 Level 1

Level 1 designs consist of basic factors only. No derived factors are allowed, and all factors in the design are crossed. Every valid trial sequence is a permutation of the crossed factors.

```

direction = Factor("Direction", ["North", "South", "East", "West"])
time_of_day = Factor("Time of Day", ["Sunrise", "High Noon", "Sunset"])

```

3.2.2 Level 2

Level 2 designs still consist of only basic factors, but allow additional basic factors to be present which are uncrossed. Every valid trial sequence is a permutation of the crossed factors and random level selections for the uncrossed factors.

3.2.3 Level 3

Level 3 designs add derived factors with windows being limited to $width = 1$ and $stride = 1$. (WithinTrial) Every valid trial sequence is a permutation of the crossed factors and random level selections for the uncrossed basic factors. The values for uncrossed derived factors will depend upon the level selections of basic factors.

```

congruency = Factor("Congruency", [
  DerivedLevel("Congruent", WithinTrial(op.eq, [color, text])),

```

```

    DerivedLevel("Incongruent", WithinTrial(op.ne, [color, text]))
  ])

```

3.2.4 Level 4

Level 4 designs relax the window limits to *width* = 2 and *stride* = 1. (Transition)

```

color_repeats = Factor("Color Repeats", [
  DerivedLevel("Repeat",    Transition(lambda c: c[0] == c[1], [color])),
  DerivedLevel("No Repeat", Transition(lambda c: c[0] != c[1], [color]))
])

```

3.2.5 Level 5

Level 5 designs allow arbitrary values for the width and stride of windows.

```

unconnected_color_repeats = Factor("Color Repeats (Unconnected)", [
  DerivedLevel("Repeat",    Window(lambda c: c[0] == c[1], [color], 2, 2)),
  DerivedLevel("No Repeat", Window(lambda c: c[0] != c[1], [color], 2, 2))
])

```

CHAPTER 4

SOLUTION COUNTING

Currently, SweetPea offloads the entire sampling process to an external tool. While an attractive option from an implementation standpoint, this approach does not scale to the sizes needed. As demonstrated in chapter two, the problem spaces that SweetPea is trying to sample from are beyond the capacity of any existing tools. One of the shortcomings of this approach is that external tools cannot take advantage of knowledge of the problem to guide their decisions.

Experimental designs in SweetPea can be viewed as combinatorics problems, each with a countable set of solutions. If we could construct a formula for counting the solutions to an experimental design, we may be able to also discover a bijection between solutions to the design and the natural numbers. This would reduce the burden of guaranteeing uniformity to randomly sampling natural numbers from the uniform distribution.

As a first foray into solution counting for SweetPea designs, this section only considers level 1, 2, and 3 designs. We also leave the set of design constraints for later consideration.

4.1 Principles of Counting

Before describing a formula for counting solutions to experimental designs, it will be useful to review a few counting principles for future reference, beginning with counting permutations of a set. The standard formula for the number of permutations of r items taken from an n -element set is:

$$P(n, r) = \frac{n!}{(n - r)!}$$

See [1]. When $n = r$, this becomes simply $n!$.

When constructing a set S by combining individual elements from multiple other sets, P_1, P_2, \dots, P_n , the size of S is the product of the sizes of sets P_1, P_2, \dots, P_n :

$$|S| = \prod_{i=1}^n |P_i|$$

This is known as the *Multiplication Principle* [1].

4.2 Partition of an Experimental Design

An experimental design is composed of three elements: a set of factors, a subset of the factors to be combined with each other (crossed), and a set of constraints. The set of crossed factors fundamentally shape the generated trial sequences. The length of a trial sequence is governed by the number of combinations of level values for each crossed factor. For example, consider a design in which there are two crossed factors, each with three levels. By the multiplication principle, there are $3 * 3 = 9$ unique combinations of level values from each factor, hence trial sequences in this design will be nine trials long.

Let D be the set of all factors in the design. Let C be the set of crossed factors, and \bar{C} be the set of factors not included in the crossing. Therefore $C \subseteq D$ and $\bar{C} \subset D$. C and \bar{C} can both be further divided based on the type of factor: basic or derived. Let C_B be the set of basic factors in C , and C_D be the set of derived factors in C . We will define \bar{C}_B and \bar{C}_D similarly for \bar{C} .

Lastly, we further divide \bar{C}_B into two more sets. Although basic factors in \bar{C} are not included in the set of crossed factors, this does not imply that they are fully independent. It is possible that factors in C_D depend on factors in \bar{C}_B . We'll refer these factors in \bar{C}_B as *source* factors, as they are at least a partial source of the data controlling factors in C_D . We will group such factors in \bar{C}_{B_s} , while \bar{C}_{B_i} represents the truly independent basic factors in \bar{C} .

We could also define C_{B_s} and C_{B_i} , however there will be no benefit to doing so, as the relationship between factors in C_B and dependent derived factors is inverted. Each combination of levels in the crossed set governs the values for C_B , which will then in turn govern dependent derived factors.

The partition of D with which we are concerned is therefore: $\{C_B, C_D, \bar{C}_{B_s}, \bar{C}_{B_i}, \bar{C}_D\}$, as shown in figure 4.1.

Lastly, we define X to be a set of ordered pairs t_1, t_2, \dots, t_n , representing the crossing of factors in C . Each ordered pair t contains items i_1, i_2, \dots, i_j , where item i_j is one of the levels

from the j_{th} factor in C . Each ordered pair t must be unique, and there is an ordered pair for every possible combination of levels from factors in C . By the multiplication principle, there will be $n = C_1 \cdot C_2 \cdot \dots \cdot C_{|C|}$ such ordered pairs, and there are $P(n, n) = n!$ permutations of this set. We will also use l to refer to the trial sequence length, which is equivalent to $|X|$.

4.3 Formula

With the partition established, we are now able to develop the formula for counting the number of solutions, s , for an experimental design. For the simplest of designs (level 1), the ordered pairs in X fully represent all possible trials. $C_B \neq \emptyset$, while every other set in the partition is empty. Every permutation of X represents a unique trial sequence, therefore there are $s = l!$ solutions to a level 1 design.

Level 2 designs introduce basic factors that are not in C , and are thus less constrained. In otherwords, starting with level 2, $\overline{C}_{B_I} \neq \emptyset$. For a factor f , let $|f|$ denote the number of levels that f has. For each $f \in \overline{C}_{B_I}$, f is fully independent; therefore, any of the $|f|$ levels may be selected for each trial. In a sequence of l trials, we can again apply the multiplication principle to determine that there are $|f|^l$ possible combinations of the levels of f . If $|\overline{C}_{B_I}| > 1$, then we apply the multiplication principle repeatedly to determine the total number of combinations for all factors in \overline{C}_{B_I} . Applying the multiplication principle one more time, we see that the total number of solutions for a level 2 design is:

$$s = l! \cdot \prod_{i=1}^{|\overline{C}_{B_I}|} |f_i|^l$$

With level 3 designs come derived factors. It is now possible that all sets in the partition are non-empty. The only sets in the partition that we have yet to consider are C_D , \overline{C}_{B_S} , and \overline{C}_D . The presence of factors in C_D does not alter the formula, as level selection for factors in C_D is still controlled by X (the crossing). \overline{C}_D also requires no special treatment as the level selection for its factors depends entirely on the levels selected for basic factors. (Whether in C_B , \overline{C}_{B_S} , or \overline{C}_{B_I} .) This leaves only \overline{C}_{B_S} , which will be our next focus.

Derived factors allow the user to provide an arbitrary predicate for each level in the factor. We have not solved the halting problem, so we will need to apply these predicates to different argument combinations in order to determine which inputs are acceptable for

each trial. We define X_S to be the set of ordered pairs representing the crossing of factors in \overline{C}_{B_S} , also referred to as the *source crossing*. The next task is to determine, for every element in X , which elements of X_S are compatible using the user-defined predicates.

For every ordered pair $t \in X$, there is some number of items in t corresponding to derived factor levels, labeled $d_1, d_2, \dots, d_{|C_D|}$ comprising the set V_t . Each $d \in V$ is associated with a predicate¹, $pred(d)$. For every $t \in X$, every $b \in X_S$ is consulted to see if the predicates for all $d \in V_t$ are satisfied. If any one of them is not satisfied, then b is not a compatible choice for t , and must be discarded from consideration for t . Once this process is complete, we are left with a list of subsets of X_S , in which the n^{th} subset indicates which elements of X_S are compatible with the n^{th} crossing in X .

More formally, let S be a list of items $J_1, J_2, \dots, J_{|X|}$. The following statements are true:

1. $\forall J \in S \mid J \subseteq X_S$
2. $\forall t \in X, \forall d \in V_t \mid pred(d) \implies \top$

Once we have generated S , we can apply the multiplication principle one more time to complete the formula for level 1, 2, and 3 designs:

$$s = l! \cdot \prod_{i=1}^{|\overline{C}_{B_I}|} |f_i|^l \cdot \prod_{k=1}^{|S|} |J_k|$$

The total number of solutions s for an experimental design, is the product of the number of permutations of the crossed factors, the number of combinations of each independent factor, and the number of acceptable combinations of all source factors.

4.4 Example

TODO

¹These predicates, also called derivation functions, are expected to be deterministic. If they are non-deterministic, then inconsistent solutions may be produced as the predicates may be applied more than once to the same arguments. This could also be handled by ensuring that each predicate is only applied once to each argument combination and the results are saved.

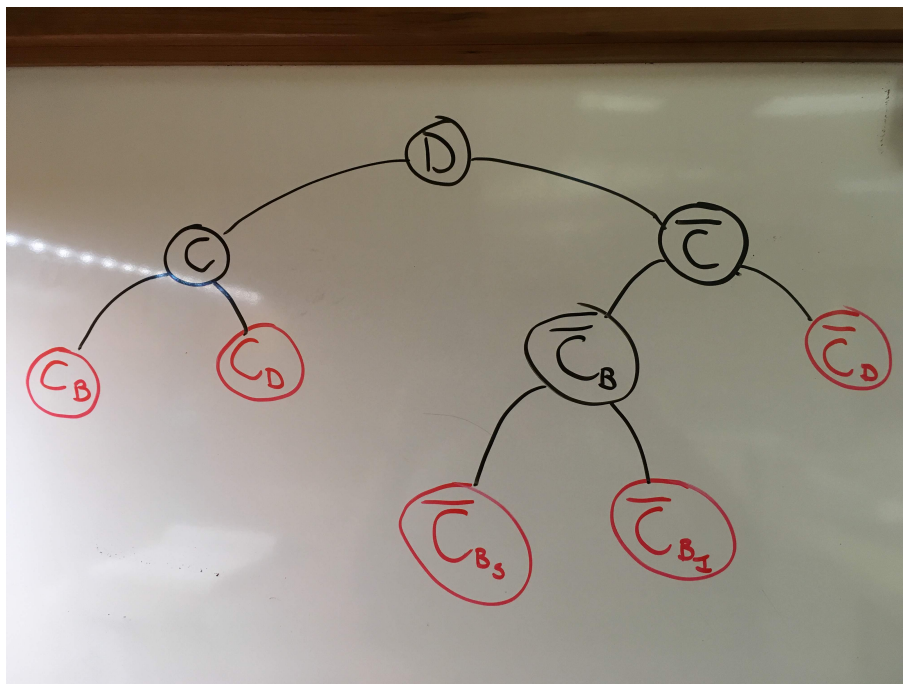


Figure 4.1. Partition of an Experimental Design

CHAPTER 5

SEQUENCE CONSTRUCTION

With an exact count of viable sequences for level 1, 2, and 3 designs, a mapping from natural numbers to valid trial sequences is now a possibility. Such a mapping would prove valuable for preserving uniformity guarantees, as well as generating solutions more efficiently. This chapter presents an algorithm that guarantees uniformly sampled solutions to level 1, 2, and 3 designs by randomly sampling natural numbers from the uniform distribution in the range of $[0, s)$. Each sampled number is then used to generate the unique trial sequence associated with that number.

5.1 Enumerating Permutations and Combinations

There are several critical components of constructing a solution based on a natural number. In this section, we will review known methods for generating the n^{th} permutation or combination of a set, as well as mapping individual natural numbers to a unique set of numbers that covers the same space using modular arithmetic. All of these techniques will play a major role in solution construction.

5.1.1 Interval Mapping

Suppose we have n numeric intervals, each from $[0, r_n)$. Revisiting the multiplication principle, we can see that there are $m = \prod_{i=1}^n r_i$ unique combinations of individual values from each interval. If j is a number selected from $[0, m)$, we can map j to a unique combination of interval value selections using the modulo and division operators. (This technique will still work even when $j \geq m$, but combinations will begin to repeat.)

We begin by computing $j \% r_1$, and saving the result as the selection for the first interval. We then recompute j to be $j = \lfloor \frac{j}{r_1} \rfloor$ before moving on to the next range. This process is repeated for r_2, r_3, \dots, r_n until the full combination is developed. Figure 5.1 gives a visual representation of this breakdown for 3 ranges, $r_1 = 3, r_2 = 4$, and $r_3 = 2$.

This technique is used repeatedly in the construction algorithm.

5.1.2 Permutations

Every permutation of a set can be identified by its *inversion* or *inversion sequence*, which loosely-defined, indicates how out-of-order the permutation is when compared with the original sequence. The concept of an inversion was first introduced by Hall [4], though we will use on Bruualdi's description [1] here.

Let S be the set of numbers $\{1, 2, \dots, n\}$. Suppose we have some permutation p of S , i_1, i_2, \dots, i_n . For each i , there exists some finite number of elements in S whose values are greater than i , yet precede i in p . This is called the inversion for i . An inversion sequence, a_1, a_2, \dots, a_n , is the sequence of inversions for a particular permutation of S .

For example, because the elements $\{1, 2, 3, 4, 5\}$ are in order, the corresponding inversion sequence for this arrangement is $\{0, 0, 0, 0, 0\}$. If we permute the sequence to obtain $\{2, 5, 4, 1, 3\}$, the associated inversion sequence is $\{3, 0, 2, 1, 0\}$. This can be interpreted to mean that 3 elements larger than 1 precede 1, 0 elements larger than 2 precede 2, 2 elements larger than 3 precede 3, 1 element larger than 4 precedes 4, and 0 elements larger than 5 precede 5. Notice that the final number in an inversion sequence must always be 0.

An inversion sequence can be used to generate the associated permutation by using the inversions to select where to place each element. Given an inversion sequence a_1, a_2, \dots, a_n , begin with n empty locations. Beginning with a_1 , skip a_i empty locations before inserting the i^{th} element of S . Using the previous example, this algorithm is demonstrated in Figure 5.2.

Finally, we can construct j^{th} inversion sequence of a set by applying the interval mapping technique introduced previously. The maximum value of each position in the inversion sequence is used to construct the intervals. To construct an inversion sequence of n inversions, the limits for the intervals are $\{n, n-1, n-2, \dots, 2, 1\}$. Lastly a zero is always appended as the final element in the sequence.

5.1.3 Combinations

We can enumerate combinations of a set of elements by again applying the interval mapping technique. For a combination of r elements, taken from a set of k elements, we know that there are k^r possible combinations by the multiplication principle. Applying

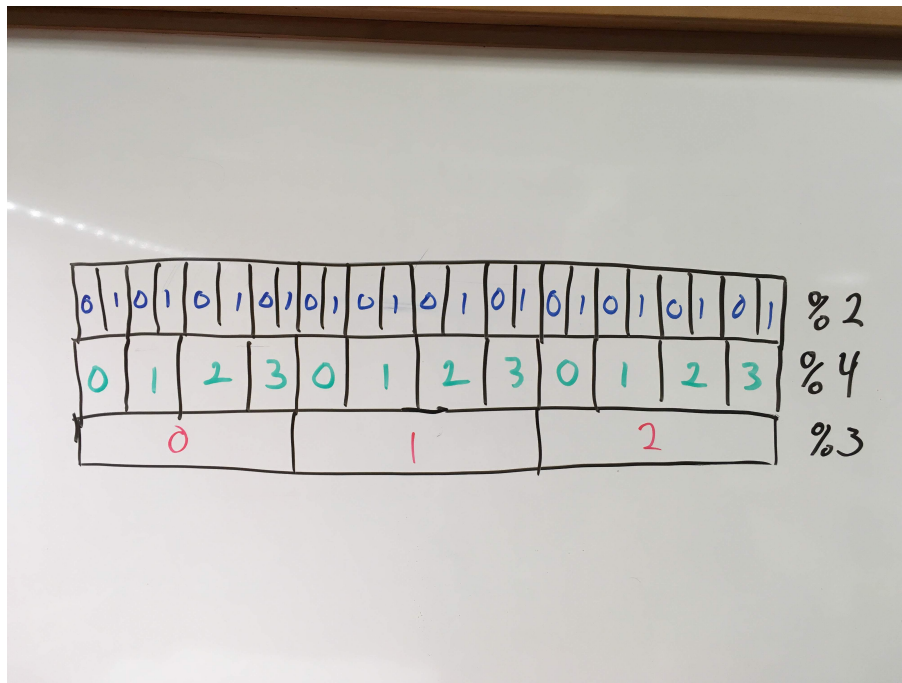


Figure 5.1. Decomposing a Number into Constituent Ranges

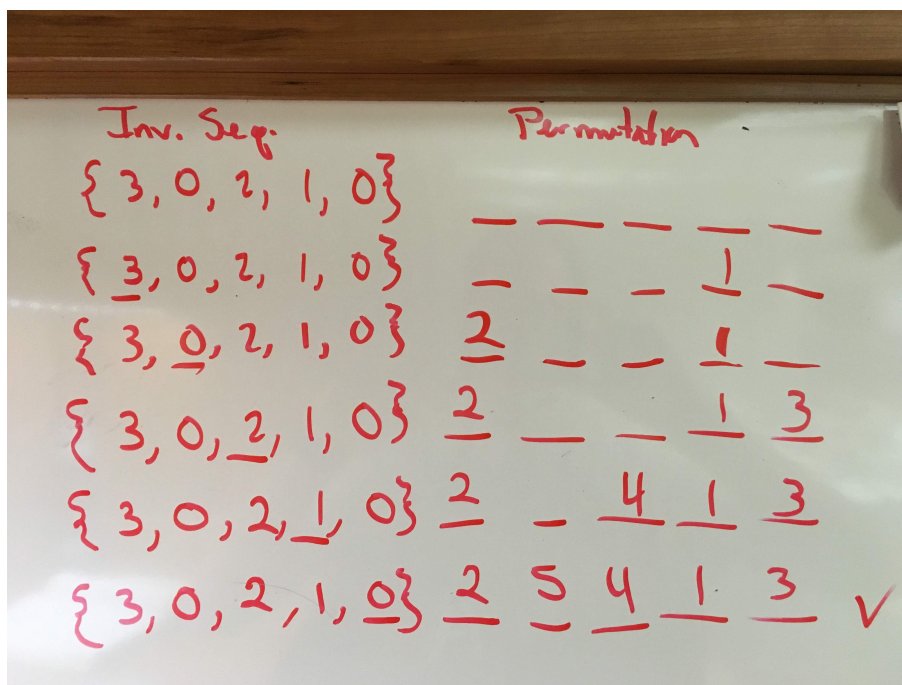


Figure 5.2. Construction of a Permutation from its Inversion Sequence

interval mapping with r intervals, each from $[0, k)$, suffices to enumerate each combination.

5.2 Construction Algorithm

With techniques established for counting solutions and constructing specific permutations and combinations, we are now ready to develop the full construction function. At a high level, the construction process has the following steps:

1. Determine the total number of solutions, s , for the design.
2. Randomly sample an integer, i , from the uniform distribution in the range $[0, s)$.
3. Use interval mapping to convert i into a unique combination of value selections, R , that identify each component of the final solution.
4. Convert each selected value in R to the associated factors and levels. Record them as part of the final solution.
5. Compute the selected values for factors in \overline{C}_D using the previously selected levels for basic factors.

Accomplishing step 1 was the subject of chapter 4. We will not revisit that here. Step 2 relies on the `randrange` standard library function in Python 3, which produces uniformly-distributed values from an arbitrarily large range. Step 3 is the complicated step, particularly selecting the ranges for interval mapping.

5.2.1 Interval Mapping for an Experimental Design

For level 1, 2, and 3 designs, as we saw in chapter 4, there is a partition of every design containing 5 sets: $\{C_B, C_D, \overline{C}_{B_S}, \overline{C}_{B_I}, \overline{C}_D\}$. In this section we will identify the upper bound of each interval to be used for interval mapping that cover all factors in the partition. We will use U to denote the set of these upper bounds. (The lower bounds are uninteresting, as they are always 0.)

We first consider C_B and C_D . Recall that together, these sets contain all factors that are crossed with each other to form the crossing, labeled X in chapter 4. Recall also that $|X|$ governs the length of a generated trial sequence: $l = |X|$, and that there are $l!$

permutations of the ordered pairs in X . Therefore, the first interval, which will determine which permutation of X to use, is $l!$. At this point, $U = \{l!\}$.

The next set in the partition is \overline{C}_{B_S} , which is the set of basic factors that are *not* included in C_B , but are sources of data for factors in C_D , and are therefore governed by the values associated with the chosen permutation for a given solution. Recall from chapter 4 that we defined X_S to represent the source crossing, and that there is a distinct subset of X_S , J_n , that represents the source crossings that are compatible with the n^{th} ordered pair in X . We also defined S to be the set of these subsets. We need a distinct interval for each of these subsets, so we add l values to U , one for the size of each subset in S . U will now contain $l + 1$ elements: $U = \{l!, |J_1|, |J_2|, \dots, |J_l|\}$.

Lastly, we consider \overline{C}_{B_I} . (\overline{C}_D also remains in the partition, but values for factors in this set are governed by other basic factors, and will be determined in step 5. No additional intervals need be computed.) Recall that \overline{C}_{B_I} represents the set of basic factors that are completely independent, meaning they are not governed by any factors in C_D . Because they are completely independent, every possible l -combination of levels for each factor in \overline{C}_{B_I} is valid. Therefore an interval must also be added for each factor f with upper bound $|f|^l$. (Recall that $|f|$ denotes the number of levels of factor f .) U is now complete:

$$U = \{l!, |J_1|, |J_2|, \dots, |J_l|, |f_1|^l, |f_2|^l, \dots, |f_i|^l\}$$

Where f_i represents the i^{th} factor in \overline{C}_{B_I} . Interestingly, the upper bounds in U are the same values which were multiplied together in chapter 4 to produce the total solution count.

5.2.2 Mapping Selected Values to Levels

Once a value has been produced for each interval, the next step is to produce the correct levels or level sequences for each factor associated with that interval. The selected permutation value is converted to a unique inversion sequence using interval mapping, which sequence is then used to construct the permutation. This permutation of integers is then converted into a sequence of level values by using each integer as an index into the ordered pairs in X .

The selected values for independent basic factors are converted in similar fashion. Each

selected value identifies a combination number, which is converted to a set of values using interval mapping. Each value is an index into the list of levels for the associated factor.

Lastly, the selected values for source combinations must be converted. These are slightly more complex, as the viable combinations vary based on the selected levels for each trial. It would be incorrect to select a source combination from the n^{th} set of valid source combinations for the n^{th} trial, because the permutation may have shifted the position of the original n^{th} trial in the sequence. A mult-level indexing scheme is employed to ensure that the selected value is used to map exclusively to valid combinations, in which the permuted values are used to index into the correct set of valid source combinations. This is shown in Figure 5.3.

For example, if the selected permutation places the original crossing value for trial number 2 in the zeroth position in the sequence (trial 0), then the source combination for trial 0 must also be selected from the valid source combinations originally generated for trial number 2.

5.3 Example

TODO

5.4 Handling Constraints

Recall that the complexity gradation did not account for the external constraints that may be applied to a design. Such constraints typically enforce some kind of rule regarding repetition of particular characteristics. For example, in the design for Stroop experiments, it is common to apply a constraint that requires that the generated sequence never contain two or more congruent trials in sequence. Constraints may be applied to the levels of derived factors as well, therefore the use may impose practically any condition upon the generated sequence.

In the future, the counting and construction algorithms could be enhanced to account for these constraints. This will be discussed further in Chapter 7. In the meantime however, basic rejection sampling is applied ensure that no generated sequences that violate constraints are ever returned to the user. All constraints are represented in the original SAT encoding, which allows us to invoke a SAT sampler with added clauses representing the

constructed sequence to verify that the constructed sequence is in fact valid. More details will be provided in the benchmarks section.

5.5 Benchmarks

Of the ten benchmarks designs that were introduced in chapter 2, 4 can be classified as level 1, 2, or 3 designs: stroop-2, stroop-3, stroop-4, stroop-5. We will repeat the benchmarks for those designs here using this construction approach to sampling. We will also introduce a few additional designs involving variations on the Stroop design to demonstrate the scalability of this approach, and the empirical performance of the rejection sampling phase. Table 5.1 reviews the basic experiment data for each benchmark design. Table 5.2 summarizes the time required to generate 10,000 samples for each design, as well as the average number of rejections per sample. (Rejections caused by constructing a trial sequence that violated external constraints.)

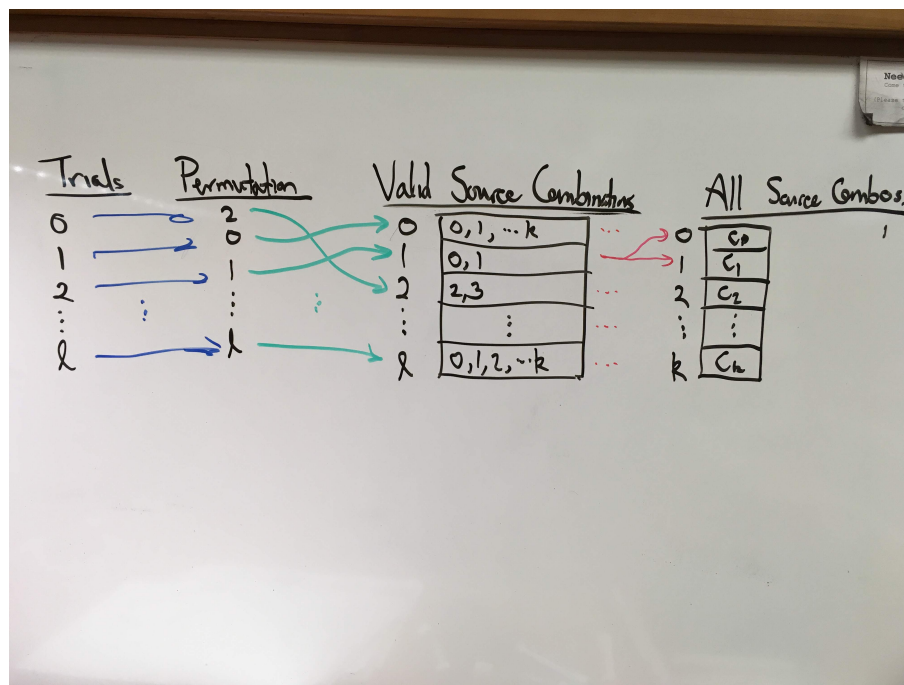


Figure 5.3. Selecting Valid Source Combinations

Table 5.1. Benchmark Experiments

Experiment Name	Sequence Length	Total Solutions	Generate Single Solution
stroop-2	4	12	
stroop-3	9	151,200	
stroop-4	16		
stroop-5	25		
stroop-10			
stroop-20			
stroop-20-constraints			

Table 5.2. Sampling Metrics

Experiment Name	10,000 Samples	Avg. Rejections/Sample
stroop-2		
stroop-3		
stroop-4		
stroop-5		
stroop-10		
stroop-20		
stroop-20-constraints		

CHAPTER 6

FUTURE WORK

This paper has presented an alternative approach to constructing trial sequences for a subset of experimental designs based on a bijection between the natural numbers and viable trial sequences. This approach is beneficial in that sampled sequences are guaranteed to be uniformly distributed, and sequences can be constructed nearly instantaneously. This is an improvement over sampling solutions to the current SAT encoding. However, there are many ways in which it could be improved, and other alternatives that should be researched.

The counting and construction methods presented here may be applied to level 1, 2, and 3 designs as defined in chapter 3, yet they do not account for external constraints that may be specified as part of the design. They only ensure that no internal constraints (between basic and derived factors) are violated. While rejecting sampling is sufficient in practice for some designs, one can construct design constraints in which valid solutions are so sparse as to render rejection sampling insufficient. Further research to incorporate constraints in the counting and construction process would be worthwhile. Combinatoric techniques for generating permutations with prohibited patterns would be likely be applicable to this process.

Additionally, the existing methods cannot yet construct sequences for designs beyond level 3 which include derived factors spanning multiple trials. (Complex windows) The approaches presented here could reasonably be extended to apply to level 4 designs (which allow Transition windows) with small adjustments. Increasing their capability to handle level 5 designs would also be worthwhile, though significantly more difficult due to potential overlap and staggering between different derived factors. It likely that rejection sampling could continue to handle constraints even for these advanced designs, if the construction methods could be discovered.

There are other approaches that could also be considered which do not rely wholly upon direct construction. Various hybrid approaches have been considered, but not thoroughly researched. For example, a numeric construction approach could be used to construct a partial trial sequence, which could then be completed using a SAT sampler, once the solution space had been sufficiently reduced. Difficulties may encountered in preserving uniformity guarantees in this approach, as a partially constructed sequence may overprune the remaning search space, leading to biased results.

Lastly, the SAT encoding itself may be altered. The naive encoding used currently is easy to understand and reason about, but is obviously inefficient. If we had more experience with other types of encoding schemes, an alternative may be discovered that could perform better. Such an encoding would still likely need to avoid encoding all permutations as distinct solutions to avoid the explosion. A hybrid approach, the inverse of the previous paragraph, would likely perform well. A SAT encoding could be developed to partially construct a sequence, including sampling the most complex aspects such as derived factors and constraints, while the remaining, simpler, values could be populated using a construction approach, thus reducing the burden on the SAT tooling.

CHAPTER 7

CONCLUSION

The current implementation of SweetPea, a language for experimental design, provides a useful standard for defining experimental designs, but has as yet failed to achieve its objective of providing uniformly sampled trial sequences for realistic experimental designs. This failure is due primarily to the inability of current SAT sampling tools to cope with massive solution spaces. While SAT solvers are still useful for generating non-uniform solutions to SweetPea design's, we have not yet been able to achieve the uniformity guarantees with the current SAT encoding.

This paper has presented the background information and benchmarks demonstrating these problems, and has introduced a complexity gradation for discussing experimental designs of varying complexity. It also introduced methods for enumerating arbitrary solutions to designs that can be classified below level 4. This allows efficient generation of samples which are guaranteed to be uniformly distributed for these classes of designs. Rejection sampling is used to ensure that no generated sequences that violate design constraints are returned to the user.

While not yet able to solve the problem for all experimental designs, this approach is extremely beneficial for simple designs. It also provides a framework and pattern upon which to build for more complicated designs. We expect that these methods can be extended to generate uniformly distributed samples for all experimental designs in efficient time and space.

REFERENCES

- [1] R. A. BRUALDI, *Introductory combinatorics*, Upper Saddle River, N.J.: Pearson/Prentice Hall, 2010.
- [2] S. CHAKRABORTY, K. S. MEEL, AND M. Y. VARDI, *A scalable and nearly uniform generator of sat witnesses*, in International Conference on Computer Aided Verification, Springer, 2013, pp. 608–623.
- [3] A. CHERKAEV, *Sweetpea: A language for experimental design*, Master’s thesis, The University of Utah, ProQuest, 12 2018.
- [4] M. HALL, JR., *Automorphisms of Steiner triple systems*, in Proc. Sympos. Pure Math., Vol. VI, American Mathematical Society, Providence, R.I., 1962, pp. 47–66.
- [5] O. KULLMANN, ed., *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, vol. 5584 of Lecture Notes in Computer Science, Springer, 2009.
- [6] S. SHARMA, R. GUPTA, S. ROY, AND K. S. MEEL, *Knowledge compilation meets uniform sampling*, in Proceedings of International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR), 11 2018.
- [7] M. SOOS, K. NOHL, AND C. CASTELLUCCIA, *Extending SAT solvers to cryptographic problems*, in Kullmann [5], pp. 244–257.
- [8] G. S. TSEITIN, *On the complexity of derivation in propositional calculus*, in Automation of reasoning, Springer, 1983, pp. 466–483.