STRAW FROM THE HAYSTACK: UNIFORMLY SAMPLING LARGE SOLUTION SPACES

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 Mathematics

The University of Utah

April 2019

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:

Cornelius Lánczos ,	Chair(s)	**17 Feb 2016** Date Approved
Hans Bethe ,	Member	**17 Feb 2016** Date Approved
Niels Bohr ,	Member	**17 Feb 2016** Date Approved
Max Born ,	Member	**17 Feb 2016** Date Approved
Paul A. M. Dirac ,	Member	**17 Feb 2016** Date Approved

by **Petrus Marcus Aurelius Featherstone-Hough** , Chair/Dean of the Department/College/School of **Mathematics**

and by **Alice B. Toklas** , 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.

For my parents.

CONTENTS

ABSTRACT . . . . . **iii**

LIST OF FIGURES . . . . . **vii**

LIST OF TABLES . . . . . **viii**

NOTATION AND SYMBOLS . . . . . **ix**

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

REFERENCES 24

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

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

NOTATION AND SYMBOLS

α	fine-structure (dimensionless) constant, approximately $1/137$
α	radiation of doubly-ionized helium ions, He++
β	radiation of electrons
γ	radiation of very high frequency, beyond that of X rays
γ	Euler's constant, approximately $0.577\,215\ldots$
δ	stepsize in numerical integration
$\delta(x)$	Dirac's famous function
ϵ	a tiny number, usually in the context of a limit to zero
$\zeta(x)$	the famous Riemann zeta function
$\ldots$	$\ldots$
$\psi(x)$	logarithmic derivative of the gamma function
ω	frequency

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 resarch.

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 [7] 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 consective 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 exlude 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 stategy 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 samping. 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 [4, 6] is the main SAT solver that SweePea 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` claues, 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 sequnces 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 [5] 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 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 reprsenting several experimental designs of increasing complexity comparable to realistic experiments written by current researchers. Table 2.1 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

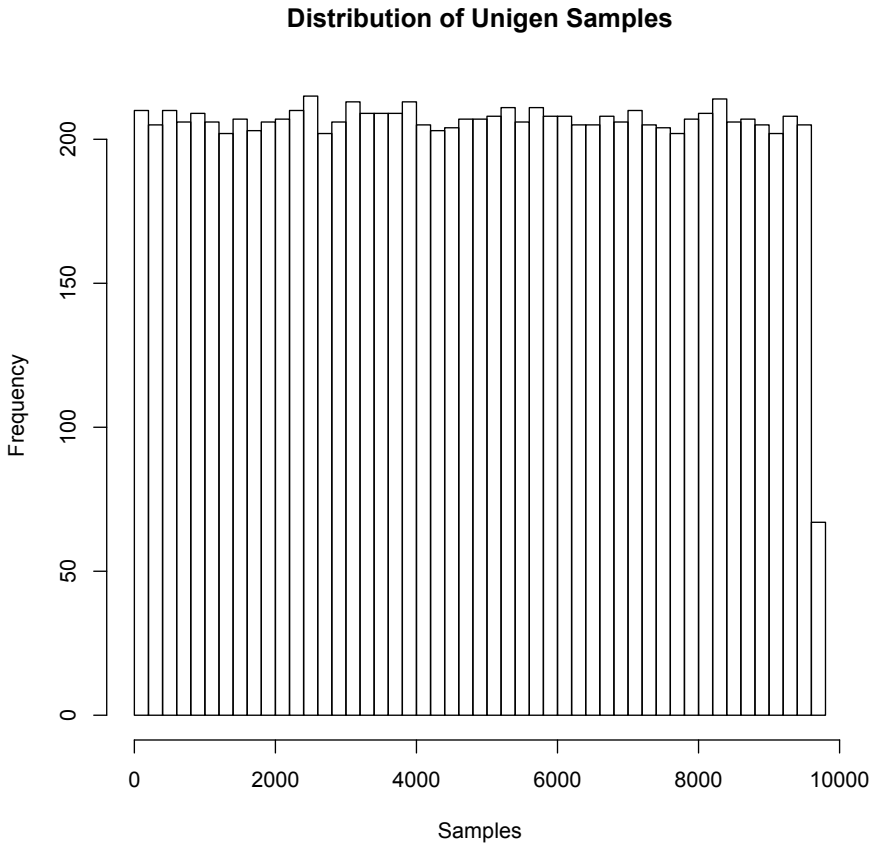

Figure 2.1. Distribution of Sequences Sampled by Unigen

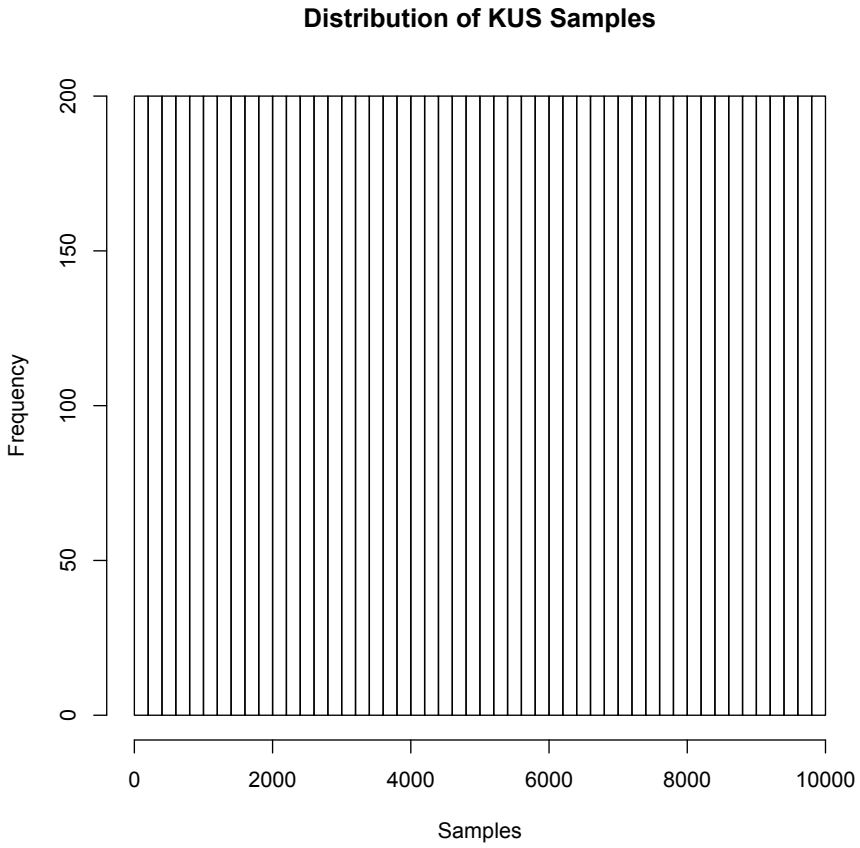

Figure 2.2. Distribution of Sequences Sampled by KUS

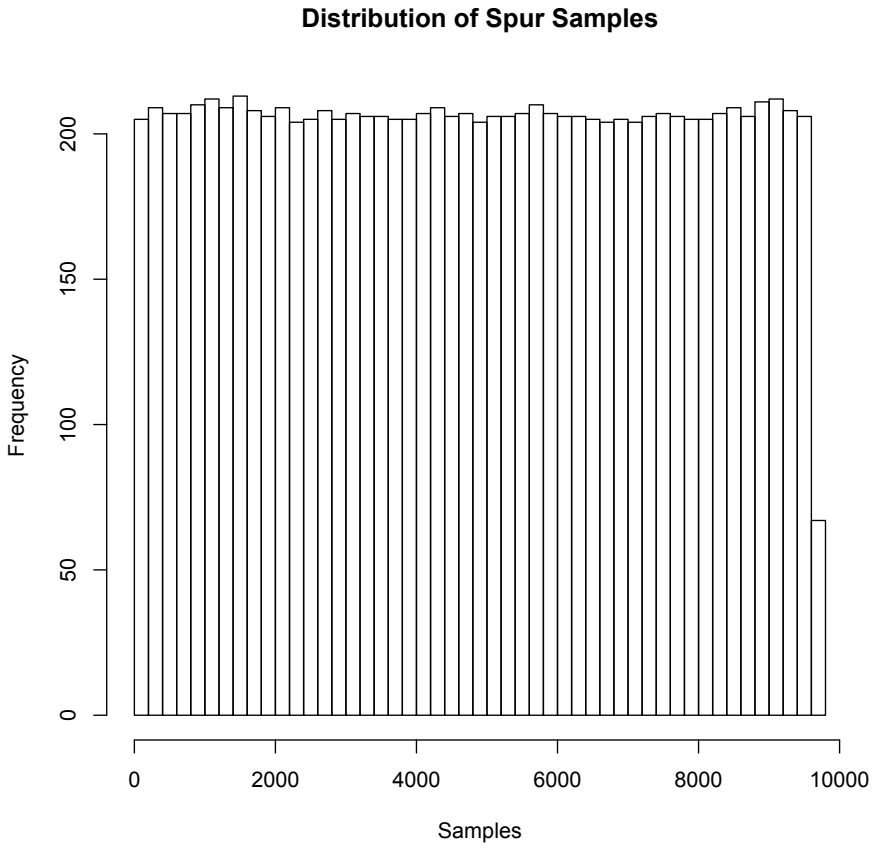

Figure 2.3. Distribution of Sequences Sampled by Spur

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 sth 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 chpater 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 then 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 mulitple other sets, $P_1, P_2, \cdots P_n$, the size of S is the product of the sizes of sets $P_1, P_2, \cdots P_n$:

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

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 $\overline{C}$ be the set of factors not included in the crossing. Therefore $C \subseteq D$ and $\overline{C} \subset D$. C and $\overline{C}$ 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 $\overline{C}_B$ and $\overline{C}_D$ similarly for $\overline{C}$.

Lastly, we further divide $\overline{C}_B$ into two more sets. Although basic factors in $\overline{C}$ 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 $\overline{C}_B$. We'll refer these factors in $\overline{C}_B$ 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 $\overline{C}_{B_S}$, while $\overline{C}_{B_I}$ represents the truly independent basic factors in $\overline{C}$.

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: $\{C_B, C_D, \overline{C}_{B_S}, \overline{C}_{B_I}, \overline{C}_D\}$, as shown in figure.... TODO.

Lastly, we define X to be a set of ordered pairs $t_1, t_2, \cdots t_n$, representing the crossing of factors in C. Each ordered pair t contains items $i_1, i_2, \cdots i_j$, 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 $n = C_1 \cdot C_2 c \vdots \cdot C_{|C|}$ 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 permutaion 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 $d_1, d_2, \cdots d_{|C_D|}$ comprising the set V_t. Each $d \in V$ is associated with a predicate, $pred(d)$. 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 $J_1, J_2, \cdots J_{|X|}$. 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.

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] 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.

[5] 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.

[6] M. Soos, K. Nohl, and C. Castelluccia, *Extending SAT solvers to cryptographic problems*, in Kullmann [4], pp. 244–257.

[7] G. S. Tseitin, *On the complexity of derivation in propositional calculus*, in Automation of reasoning, Springer, 1983, pp. 466–483.