SWEETPEA: TOWARD UNIFORM SAMPLING FOR EXPERIMENTAL DESIGN

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
December 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:

Matthew Flatt , Chair(s) **22 August 2019**
Date Approved

Vivek Srikumar , Member **22 August 2019**
Date Approved

Jonathan Cohen , Member **08 September 2019**
Date Approved

none , Member **none**
Date Approved

none , Member **none**
Date Approved

by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computer Science**

and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

SweetPea is a new language providing a means of declaratively describing experimental designs. Because of their combinatorial nature, realistic experimental designs have too many conforming trial sequences to reasonably enumerate all of them for random sampling. As a result, uniformly sampled trial sequences must be generated using alternative means. SweetPea currently does this by encoding the design as a SAT formula, to which a SAT sampler is then applied to create uniformly sampled solutions. Although recent SAT samplers can process formulas with large solution spaces, particular characteristics of experimental design have made this approach unfeasible. We present evidence explaining why SAT samplers in their current state are not viable for this problem and lay the foundation for an alternative encoding, based on a bijection from natural numbers to unique trial sequences. This alternative encoding allows us to generate uniformly-distributed random samples for certain classes of designs in a fraction of the time required by previous methods.

For my family

CONTENTS

CHAPTERS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

A big thank you to my advisor, Matthew Flatt, for being willing to take on a clueless Masters student. Matthew sees things so clearly. I am deeply appreciative of his guidance and perspective. Thank you to the other members of my committee: Vivek Srikumar and Jonathan Cohen. Their input brought an incredibly diverse set of possibilities to my attention; I regret not having enough time to develop all of them more fully.

Thanks to the other students involved as well. Thanks to Annie Cherkaev for always being willing to answer my questions and taking the time to help me understand. Thanks to Sebastian Musslick for being the patient guinea pig and providing valuable feedback every step of the way.

A heartfelt thank you to my friends and teammates at FamilySearch: Jim, Chris, Bob, Joseph, Barrett, Brent, and many others. They all pulled extra weight to support me when the academic load was particularly heavy, and I credit them for this achievement as much as anyone else. I couldn't ask for a better team. FamilySearch also shared a large portion of the financial burden for this endeavor, for which I will always be grateful.

Thanks to Mom and Dad for always encouraging me, and never letting me believe that I wasn't smart enough.

Lastly, and lest there be any doubt, most importantly, thank you to my extraordinary wife, Brittney. Over the last three years, she has played the single mother to our children more times than I can count while I stayed late at work or the library. She has always selflessly sought to help those around her, and I certainly could not have succeeded alone. I love her dearly.

CHAPTER 1

MOTIVATION

SweetPea [**?**] is a new language that allows researchers to express experimental designs in their research declaratively. Experimental designs typically consist of several different variables combined in different ways to generate a sequence of trials. Researchers present these trials as stimuli to a subject to observe characteristics of the subject's responses. The nature of these designs is combinatorial. Even for designs with a modest number of variables and trials, the number of possible combinations can be staggering. This complexity has plagued researchers for many years. SweetPea was created to alleviate some of this pain; although so far, it has only partially succeeded. In this chapter, we will first present an overview of the SweetPea language, followed by a brief description of its implementation. Lastly, we identify the specific contributions that SweetPea makes as well as aspects in which it still falls short.

1.1 SweetPea Overview

This section will provide a brief overview of the components of an individual experimental design in order to lay the groundwork for future discussion. For further details, refer to the original paper [**?**]. At the highest level, a *block* defines an experimental design. The properties of the block may vary depending on its type. Given a block, SweetPea will generate some number of trial sequences that conform to the design described by the block. An individual trial is simply a selected value for each variable in the design. Trial sequences are presented as stimuli to different subjects under observation to determine their effect.

At present, only a single block type is provided by SweetPea, which is the *FullyCrossBlock*. This block has three components:

1. A set of *factors*.

2. A subset of the factors to combine to form the *crossing*.

3. A set of *constraints*

These three components provide the structure and rules that govern the construction of trial sequences.

1.1.1 Factors

Factors are the primary primitive that SweetPea provides. A single factor consists of a name and a list of levels. For each trial in a sequence, each factor is assigned a single level from its list of possibilities. For example, the following excerpt defines a factor named ″color″ with two levels: ″red″ and ″blue.″

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

The above example is an example of a *basic* factor, meaning that the levels are literal values. SweetPea also provides *derived* factors in which the levels depend upon the level selections for other factors in the design. Derived factors are useful for expressing relationships between different factors which can then be used to add additional structure to the design. As an example, a user may wish to define a ″congruency″ factor that expresses whether or not two other factors in the design exhibited the same level:

```
def congruent(color, text):
    return color == text

def incongruent(color, text):
    return not congruent(color, text)

congruency = Factor("congruency", [
  DerivedLevel("congruent",   WithinTrial(congruent,   [color, text])),
  DerivedLevel("incongruent", WithinTrial(incongruent, [color, text]))
])
```

If the `congruency` factor were set to the level ″congruent″ for a given trial, that would indicate that the levels selected for the *color* and *text* factors were equivalent. Derived factors accept arbitrary python functions to express the relationship, though there are additional requirements regarding how the functions partition the space.

In the previous example, the `WithinTrial` function constructs what is referred to as the `window` for the level. Consider a sequence of trials; one may wish to define a derived factor that identifies a characteristic of the level selections across multiple trials. A window is the language construct that specifies which trials the factor will consider. Windows have four characteristics: A derivation function, a list of factors (arguments to the derivation function), a *width* and a *stride*. The width of a window specifies how many subsequent trials are considered for each occurrence of the derived factor, while the stride specifies the offset between subsequent occurrences. The `WithinTrial` window is an alias for a general window with a width and stride of one, meaning that it only considers levels within the current trial, and the derived factor applies to each trial in the sequence. Figure 1.1 presents a visual representation of this concept.

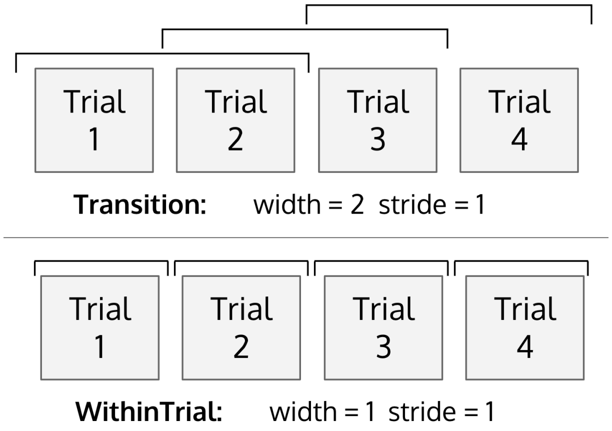
Trial 1
Trial 2
Trial 3
Trial 4
Transition: width = 2 stride = 1
Trial 1
Trial 2
Trial 3
Trial 4
WithinTrial: width = 1 stride = 1

Figure 1.1. Window Properties

A second window alias exists, `Transition`, which has a width of two and a stride of one. As its name implies, this type of window is useful for expressing relationships between consecutive trials in a sequence. For example, a `Transition` window could be used to define a factor that determines if the level for `color` is repeated or not between two trials as follows:

```
def repeat(colors):
    return colors[0] == colors[1]

def switch(colors):
    return not repeat(colors)

color_repeats = Factor("Color Repetition", [
  DerivedLevel("Repeat",    Transition(repeat, [color])),
  DerivedLevel("No Repeat", Transition(switch, [color]))
])
```

Observe that derived factors with this type of window never apply to the first trial in a sequence, as no prior trial exists with which to compare the current trial. 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 after that. Although the language does allow for windows with arbitrary widths and strides, the computational complexity of computing the levels grows exponentially as the width increases. Therefore there are practical constraints on the usable size of a window.

1.1.2 Crossing

Once the user has defined some factors, a trial sequence could be generated by randomly combining different levels from each factor, but this would be insufficient. Researchers most often need to counterbalance or cross the levels of some factors to ensure that some particular combination of levels is present in the final sequence. The block specifies a subset of factors that should be used to construct this *crossing*. The crossing is the cartesian product of the set of levels from each factor in the subset. For example, suppose two factors are selected to form the crossing, each with two levels: `[a, b]` and `[1, 2]`. The resultant crossing would be `[a, 1]`, `[a, 2]`, `[b, 1]`, and `[b, 2]`.

The crossing is the most authoritative constraint in the entire design. Every level combination in the crossing is required to be present in any generated trial sequence. Furthermore, a generated trial sequence will always have the minimum number of trials required to give full representation to the crossing. If there are n level combinations in the crossing, then there must be at least n trials in any generated trial sequence. If the crossing includes derived factors that span two or more trials (window width greater than one), then the minimum number of trials increases. For example, if the crossing contains a derived factor with a transition window, then generated trial sequences must have $n + 1$ trials because the derived factor will not have a value in the first trial.

1.1.3 Constraints

Lastly, constraints allow the designer to require or prohibit particular patterns in the final generated trial sequence. There are only two such constraints available today: `AtMostKInARow` and `ExactlyKInARow`. Each of these allows the designer to enforce that there are no more than, or exactly, *k* (user-specified) repetitions of a specific factor level in any generated sequence. These constraints are useful for ensuring some level of variety, or lack thereof, between trials in the sequence. While these constraints appear simple initially, they become quite powerful when applied to derived factors.

1.1.4 Complete Example

Having introduced each component of the block that defines a design, we can now view an example in full. This example will show how the Stroop test [**?**], a popular design among experimental psychologists, can be expressed in the SweetPea language. In the Stroop test, a subject perceives potential conflicts in the trial sequence that influence their reaction time. For example, a subject may be shown printed words representing colors and asked to read the word. However, the ink color of the word may be different from the printed word itself, leading to a delayed reaction due to the cognitive conflict. This design is easily represented in SweetPea as follows:

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

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

constraints = [AtMostKInARow(1, ("congruent?", "yes"))]

block = FullyCrossBlock([color, text, congruent],
                        [color, text],
                        constraints)
```

`color` and `text` represent the possibilities for the font color and the printed text in a trial. A `congruent` factor is also defined, to define a constraint that will ensure the subject never sees two consecutive trials in which the ink color and printed text matched. (This constraint is not inherently part of the original Stroop experiment, but is used here for purposes of demonstration.) Lastly, the final block is constructed using just `color` and `text` to form the crossing. Table 1.1 shows one possible trial sequence that conforms to this design.

Table 1.1. Example Trial Sequence

Trial	Color	Text	Congruent
1	blue	green	no
2	red	red	yes
3	blue	red	no
4	green	red	no
5	green	blue	no
6	blue	blue	yes
7	red	blue	no
8	red	green	no
9	green	green	yes

1.2 Current Implementation

SweetPea encodes the experimental design in a boolean satisfiability (also known as SAT) formula such that any satisfying assignment represents a unique trial sequence that upholds the design. The assignment can then be translated back into the original design language for use as stimuli. At present SweetPea employs a simple positional encoding. In this encoding, a distinct variable represents every level of every factor of every trial. A SAT sampler is then applied to the formula to generate approximately uniformly distributed solutions.

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 Boolean formula can encode the problem at hand, 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 a solution to the formula or indicate that the formula is *unsatisfiable*.

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 evaluate to 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*.

Tools for interacting with SAT formulas 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 it meets three conditions: the entire formula must be a conjunction of clauses, each clause consists of an individual variable or a disjunction of variables, and the negation operator may only appear on individual variables, not entire clauses. Any Boolean formula may be expressed in conjunctive normal form, though converting an arbitrary formula to CNF usually requires growth in some dimension. SweetPea uses the Tseitin Transformation [**?**] to do this conversion, which has the tightest known bounds on formula length, at the expense of allocating additional variables.

1.2.2 Example Encoding

Continuing with the example design for the Stroop test introduced in 1.1.4, we will see how SweetPea generates the SAT encoding for this design. The sequence length, as determined by the crossing, is 9. The current encoding scheme will allocate $9 \cdot 8 = 72$ variables to represent every level of every 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 combination in the crossing be present in the sequence. The first step of the encoding can be visualized easily, as demonstrated in Table 1.2.

In this encoding, 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 the current encoding scheme, refer to the original paper [**?**].

Once SweetPea generates the CNF encoding for a design, it applies a SAT sampler to

Table 1.2. Example of an Encoding Diagram

	Color			Text			Congruent?	
Trial Number	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

the formula to generate some number of solutions. SAT samplers are similar to solvers, in that they produce satisfying assignments to a boolean formula; however, they guarantee some level of uniformity in the distribution of generated samples. Historically, SAT samplers could only provide strong uniformity guarantees for small formulas, but recent research shows some potential for improving the scalability of current sampling algorithms. SweetPea targets the SAT sampler *UniGen*. SAT sampling will be discussed in further detail in Chapter 3.

1.2.3 One-to-One Correspondence

In order for the uniformity guarantees of sampling tools to translate back into the problem space of the original experiment design, the encoding scheme must 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 than S_2.

A few observations build our confidence the encoding upholds this correspondence. First, all factor and level selections, including associated constraints for derived levels, in the original problem space are represented directly in the SAT encoding. A unique variable represents every level of every factor for every trial in the sequence. Clauses are added to the formula to ensure that variable assignments are consistent with the actual constraints of the problem. Referring back to the example in Table 1.2, the levels for the `Color` factor for trial number 1 are represented by variables 1, 2, and 3 respectively. The SAT formula embeds a popcount, or Hamming weight, circuit that requires that exactly one of those variables be true. The encoding repeats this circuit for each of the nine trials for every factor. Other constraints are also embedded to represent all other rules of the design. Thes constraints enforce relationships between basic and derived factors, as well as guarantee that the full spectrum of crossed factor combinations appears in the sequence. Because SweetPea directly translates all data from the original design into SAT variables and constraints, we believe there is no risk of generating multiple solutions to the formula that represent the same trial sequence.

However, some of these constraints are represented using operators prohibited in CNF, including logical implication. Of necessity, these expressions must be rewritten in an equivalent form using only those primitives allowed by CNF. In order to avoid an exponential explosion in the length of the formula, this rewriting process introduces additional variables to represent repeating clauses. The second observation is that SweetPea applies the Tseitin transformation to do this conversion to CNF, which binds every variable that it introduces to the state of a clause in the original formula. As a result, the number of solutions remains the same, and there is no possibility of underconstrained variables skewing the results.

Lastly, we observe that for small designs, we can exhaustively enumerate all possible trial sequences. The number of enumerated sequences matches precisely the result obtained when using a SAT model counter to determine the number of solutions to the SAT formula. Were there not a one-to-one mapping, these figures would disagree.

1.3 SweetPea's Contributions

SweetPea was created to solve two specific problems facing researchers today:

1. No standard tooling exists for defining experimental designs. Researchers work independently to construct trial sequences for their designs using whatever means they deem appropriate. As a result, it is nearly impossible for researchers to collaborate and share their designs.

2. Trial sequences generated by researchers today are biased. Because they do not have sufficient tooling to sample trial sequences from a uniform distribution randomly, homebrew solutions make random choices with backtracking when constraint violations are detected. This approach skews the distribution of generated samples, making it difficult for peers to reproduce published results.

SweetPea aims to solve both of these problems by providing a standard language for expressing experimental designs as well as a synthesis engine that generates a uniform distribution of random samples.

1.4 SweetPea's Future

SweetPea sounds panacean, but it has yet to realize its objectives fully. The language is still under revision, but SweetPea shows excellent promise for fulfilling the first objective. On the other hand, uniform sampling of trial sequences has proven significantly more challenging. We hoped that leveraging recent developments in the field of SAT-sampling would allow us to solve this problem. However, the applicability of SAT samplers is fundamentally limited to formulas with solution counts below a particular threshold. Realistic designs routinely exceed this threshold.

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 previous example, the sequence length denoted l, was 9. Without the congruence constraint, there would be $9! = 362,880$ solutions, which is still rather modest. Consider an experiment however in which $l = 36$. The solution count grows to $36! \approx 2^{138}$, which is many orders of magnitude larger. Table 1.3 shows how the number of solutions explodes as l increases. As will be seen in Chapter 3, it is common for experimental designs to have sequence lengths even larger than 36.

In practice, designs have constraints; however, the constraints only exclude specific patterns in the permutations; therefore, the number of solutions remains dominated by $l!$. Furthermore, designs commonly contain several unconstrained factors, which magnifies the combinatorial explosion. Generating all possible solutions is not feasible.

It has been well established that prior SAT-samplers have traded strong uniformity

Table 1.3. Growth of Solution Space

Sequence Length (*l*)	**#SAT (*l*!)**
4	24
6	720
9	362,880
12	479,001,600
16	20,922,789,888,000
20	2,432,902,008,176,640,000
25	15,511,210,043,330,985,984,000,000
30	265,252,859,812,191,058,636,308,480,000,000
36	371,993,326,789,901,217,467,999,448,150,835,200,000,000

guarantees for scalability [**?**], and thus were not suitable for even moderately sized formulas. Unigen [**?**] contributed two primary advancements to bridge this gap between scalability and uniformity guarantees, but these advancements require specific conditions that impose fundamental limits on the number of possible solutions to the formula. This thesis shows that experimental designs generally do not uphold these conditions, and therefore SAT samplers on their own are insufficient for generating uniformly distributed solutions. However, due to the combinatorial nature of these designs, arithmetic principles can be applied to directly construct uniformly distributed samples for some classes of experimental designs. Further success will likely involve a hybrid between sampling and construction approaches.

CHAPTER 2

A COMPLEXITY SPECTRUM FOR SWEETPEA DESIGNS

The complexity of SweetPea designs dramatically varies depending on the user's requirements. At one end of the complexity spectrum are simple designs in which there are no structured relationships between trials in a sequence. At the other end are the most intricate designs, in which many structured relationships exist between multiple trials in a sequence at varying intervals. This chapter defines a precise gradation of five complexity tiers for experimental designs in order to facilitate later discussion, including evaluating different sampling strategies based on the complexity tiers to which they apply.

Recall that a block defines an experimental design. Blocks have three characteristics: 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 first characteristic: the factors present. This gradation ignores constraints on the overall design, as their complexity depends primarily upon the complexity of the factors that they constrain.

2.1 Complexity Gradation

The complexity tier for an arbitrary experiment is determined solely by the types of factors and windows present in the design. We will begin with the most straightforward designs and finish with the most complex.

2.1.1 Tier 1

Tier 1 designs consist of basic factors only. No derived factors are allowed, and the crossing contains all factors in the design. 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"])
```

2.1.2 Tier 2

Tier 2 designs still consist exclusively of basic factors, but allow additional basic factors to be present in the design which are uncrossed. Every valid trial sequence is a permutation of the crossed factors paired with random level selections for the uncrossed factors. (The crossing does not bear any impact on the uncrossed factors.)

2.1.3 Tier 3

Tier 3 designs add derived factors with windows being limited to those with *width* $= 1$ and *stride* $= 1$. (i.e., `WithinTrial`) Every valid trial sequence is a permutation of the crossed factors paired with random level selections for the uncrossed basic factors. The values for uncrossed derived factors depend solely upon the level selections of basic factors.

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

2.1.4 Tier 4

Tier 4 designs relax the limitations on derived factors to allow windows with *width* $= 2$ and *stride* $= 1$ as well. (i.e., `Transition`) Valid trial sequences are still permutations of the crossed factors; however, `Transition` windows may prohibit some permutations when included in the crossing.

```
def repeat(colors):
    return colors[0] == colors[1]

def switch(colors):
    return not repeat(colors

color_repeats = Factor("Color Repeats", [
  DerivedLevel("Repeat",    Transition(repeat, [color])),
  DerivedLevel("No Repeat", Transition(switch, [color]))
])
```

2.1.5 Tier 5

Tier 5 designs further relax the limitations on derived factors to allow windows with any width and stride, so long as they do not create constraints that are impossible to satisfy.

```
def repeat(colors):
    return colors[0] == colors[1]

def switch(colors):
    return not repeat(colors)

unconnected_color_repeats = Factor("Color Repeats (Unconnected)", [
  DerivedLevel("Repeat",    Window(repeat, [color], 2, 2)),
  DerivedLevel("No Repeat", Window(switch, [color], 2, 2))
])
```

2.2 Conclusion

This complexity spectrum identifies five specific tiers for classifying experimental designs. Each tier allows incrementally more complexity than its predecessor. This spectrum proves useful for reasoning about different designs and the complexities involved with generating conforming trial sequences.

CHAPTER 3

SAT SAMPLING

As was briefly mentioned in Chapter 1, the current implementation is unable to generate samples for realistic experiment designs without sacrificing uniformity guarantees. Although recent advancements in SAT sampling appeared promising, they only apply under specific conditions. This chapter explains in greater detail why the current encoding for arbitrary SweetPea designs does not preserve these conditions. It also shows how these conditions impose an upper-bound on the solution space, which realistic designs exceed. We also provide benchmarks empirically substantiating this claim and explain the practical limitations of the current approach. To do so, we examine UniGen's contributions and their relationship to the SweetPea encoding. Lastly, this chapter presents benchmarks for generating trial sequences using a few other SAT samplers, further showing that relying exclusively on a SAT sampler is not a viable strategy for SweetPea.

3.1 Experiments

Throughout this chapter, we will apply multiple tools to formulas representing realistic SweetPea designs to evaluate their performance. This section briefly introduces these designs to give the reader a sense of their size and complexity. Table 3.1 enumerates the designs and lists a few fundamental properties, including the number of variables, clauses, and possible solutions.

For `stroop-2` and `stroop-3`, the solution count is precise, as computed by a SAT model counter. For all other experiments, the solution count is approximated as the factorial of the sequence length, as model counters were unable to provide a count in a reasonable amount of time. These approximate solution counts ignore both constraints and independent factors. While constraints reduce the solution count, independent factors increase the solution count. In practice, independent factors contribute more solutions than are removed by constraints.

Table 3.1. Benchmark Experiments

Experiment Name	Tier	l	Variables	Clauses	Total Solutions
stroop-2	3	4	392	1,559	12
stroop-3	3	9	1,965	8,531	151,200
stroop-4	3	16	3,848	16,451	$\approx 2^{44}$
stroop-5	3	25	10,326	45,991	$\approx 2^{83}$
stroop-congruency-balanced	4	17	5,570	24,215	$\approx 2^{48}$
stroop-response	4	33	16,242	72,679	$\approx 2^{122}$
padmala-pessoa	4	37	116,289	559,080	$\approx 2^{143}$
task-switching	4	32	14,246	62,700	$\approx 2^{117}$
task-switching-2	5	146	742,832	3,672,632	$\approx 2^{844}$

All benchmarks were run on a PC running Ubuntu 18.04.2 LTS with 16GB of memory and an 8-core Intel Core i7-7700 at 3.60 GHz.

3.2 UniGen

UniGen [?, ?] 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. As explained in Chapter 1, SweetPea builds the CNF such that there is a one-to-one relationship between solutions to the formula and unique trial sequences that conform to the design. UniGen employs a hashing technique that partitions the solution space into bins that are small enough to allow random sampling of solutions from each bin. UniGen has worked well for some designs, but its capacities are quickly overwhelmed by even moderately sized designs.

UniGen relies on a family of 3-independent hash functions, known to have strong uniformity guarantees [?], to partition the solution space of a formula into bins. However, there are two problems with this approach for larger formulas:

1. It requires a parameter, denoted m, to influence the size of the bins. Choosing an appropriate value for m is non-trivial. Previous samplers have been forced to use trial and error to find a value that produced bins of the desired size.

2. It relies heavily on the `XOR` logical operation. As the number of variables per `XOR` clause grows, the computational complexity of generating solutions increases signif-

icantly [?].

UniGen's main contributions were techniques for mitigating both of these obstacles.

3.2.1 Model Counting

The first problem is choosing an appropriate value for m. Let F denote a boolean formula in CNF, and let R_F denote the set of satisfying assignments, or witnesses, for F. The best value for m depends on $|R_F|$; however, in practice $|R_F|$ is not known upfront. UniGen uses an approximate model counter, ApproxMC [?], to estimate $|R_F|$. With this estimate, a narrow range of values for m can be generated and tested for fitness. Starting with a narrow range of possibilities significantly reduces the effort required to find the best value for m, at the cost of needing to estimate $|R_F|$.

Unfortunately, ApproxMC has not been able to successfully approximate the count of any of our benchmark experiments beyond `stroop-3` in a reasonable amount of time (10 hours), as shown in Table 3.2.

Table 3.2. Time to Compute Approximate Model Count with ApproxMC

Experiment Name	Time
stroop-2	5.07ms
stroop-3	27.4s
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

In reality, the algorithm implemented by ApproxMC relies on the same family of 3-independent hash functions used by UniGen itself; therefore it will be subject to the same problem of `XOR` clause growth, which the next section will explain. A new version of ApproxMC was recently published [?]; however, it relies on the same family of hash functions, so it is unlikely that it will offer significant improvements. Also, due to the combinatorial nature of the problem, it is unlikely that this would be successful even with more time or processing power.

Although available tools struggle to approximate solution counts for SweetPea designs, this is surmountable by estimating $|R_F|$ ourselves using information from the original problem domain. Because each experimental design constructs a sequence of trials based on combinations of factor levels, we can apply techniques for counting sets from combinatorics to estimate $|R_F|$. This counting strategy will be discussed further in Chapter 4.

3.2.2 XOR Clause Reduction

The second problem requires reducing the length of `XOR` clauses. Let X be the set of variables in F. Using terminology from Chakraborty et al. [**?**], X is called the *support* of F. They observed that there is often a subset $S \subseteq X$, referred to as an *independent support* of X, which fully determines all variable assignments for any satisfying assignment. That is, for all satisfying assignments of F, there are no two assignments whose values differ only in the assignments of variables $\overline{S}$. Figure 3.1 visually represents this concept.

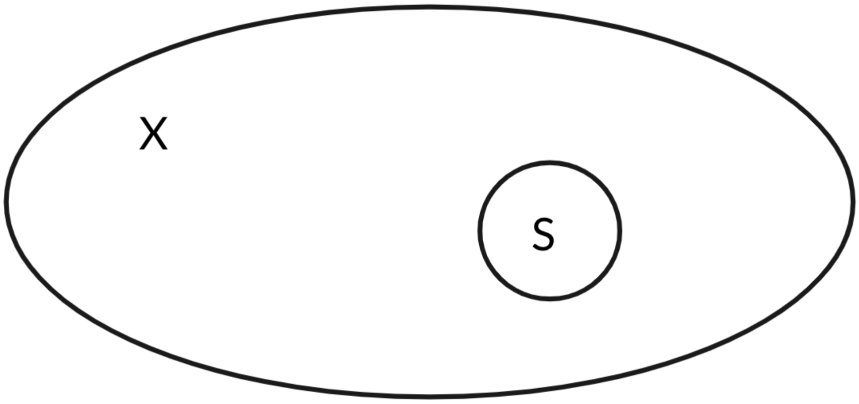

Figure 3.1. Visual Representation of an Independent Support

The primary target domain for UniGen is *Constrained Random Verification* or CRV. In the domain of CRV, $|S|$ is typically significantly smaller than $|X|$, often by many orders of magnitude. UniGen exploits these observations by only applying variables in S to the hash functions, which yields a significant reduction in the length of generated XOR clauses. Therefore UniGen indeed scales to formulas with hundreds of thousands of variables, and even millions of clauses, but *only* when $|S|$ remains small, typically $|S| < 100$. In the UniGen benchmarks, UniGen generated samples for over 200 different formulas. The largest of these, `demo3_new`, had $865,935$ variables and $3,509,158$ clauses, but $|S|$ was only 45. Similarly, the largest value of $|S|$ in the benchmarks was 78 in `diagStencil_new`, with $94,607$ variables and $2,838,579$ clauses. **[?]**

How does $|S|$ grow in the formulas generated by SweetPea? SweetPea currently selects all variables that directly represent level values for each factor in each trial as the independent support. For example, in the encoding shown in Table 1.2, all 72 variables would be included in the independent support. However, the independent support does not include variables used to encode additional constraints or intermediate variables introduced during Tseitin encoding. While this set does not grow factorially, it still quickly grows into the hundreds and thousands, as shown in table 3.3. While this independent support is certainly not minimal, we will see that even an optimal selection for the independent support could not guarantee $|S| < 100$.

Table 3.3. Independent Support Size for Benchmark Experiments

Experiment Name	$\|S\|$
stroop-2	24
stroop-3	72
stroop-4	160
stroop-5	300
stroop-congruency-balanced	270
stroop-response	526
padmala-pessoa	1,303
task-switching	636
task-switching-2	2,770

There are a few simple improvements that could be made to the independent support computation today. Rather than including the variables for *all* factors and levels in the independent support, it could exclude variables for derived factors whose levels are constrained by other factors. Additionally, a more efficient SAT encoding could be used based on combinations of variables rather than representing all levels as individual variables. (Although this would increase the complexity of encoding constraints.) Both of these would reduce $|S|$ by a degree, allowing UniGen to be useful for slightly larger formulas.

For any formula F with independent support S, there are, by definition, exactly $2^{|S|}$ distinct assignments to F. Therefore there can be no more than $2^{|S|}$ satisfying assignments, or solutions, to F. (In practice the number of solutions will be much less, as not *all* possible assignments will satisfy the formula.) Assuming an upper-bound of 100 for $|S|$, a perfectly efficient SAT encoding, and a minimal independent support, UniGen could only sample solutions to formulas with at most 2^{100} solutions. As seen in Table 3.1, SweetPea designs regularly exceed this threshold. Therefore UniGen alone is insufficient for generating solutions, regardless of any potential improvements to the SAT encoding or the independent support computation.

3.2.3 Benchmarks

This shortcoming is proved out in practice. UniGen fails to count or sample from formulas where $|S| > 100$. Table 3.4 denotes the time required for UniGen to generate 10 samples for each experiment. An estimate for $|R_F|$ is provided manually based on the factorial of the sequence length. Nearly all of them fail to complete in the allotted time (10 hours).

Table 3.4. Time to Generate 10 Samples with UniGen

Experiment Name	Time to Sample
stroop-2	-
stroop-3	2.18s
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

3.3 Other Benchmarks

As mentioned at the beginning of this chapter, we tested several other SAT samplers. This section presents the results of testing multiple SAT solving and sampling tools, as well as additional information on the implementation strategy of each tool. All sampling benchmarks were configured to generate ten samples at a time

3.3.1 CryptoMiniSat

CryptoMiniSat [?, ?] is a SAT solver. It is used internally by UniGen, as well as directly by SweePea to generate non-uniformly distributed trial sequences. SweetPea does non-uniform sampling by simply solving for any solution, 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 their distribution may be non-uniform. Sacrificing this guarantee allows them to quickly determine whether or not a particular design is even viable before investing additional resources. Lastly, in addition to logical `AND`, `OR`, and `NOT` clauses, CryptoMiniSat also natively supports `XOR` clauses, although SweetPea has not yet put these to use. (This is an opportunity for improvement.) Table 3.5 denotes the time required for CryptoMiniSat to compute a single solution to the SAT formula for each experiment.

Table 3.5. Time to Solve for 1 Solution with CryptoMiniSat

Experiment Name	Time to Solve (ms)
stroop-2	4.9
stroop-3	9.1
stroop-4	18.9
stroop-5	64.4
stroop-congruency-balanced	23.9
stroop-response	232
padmala-pessoa	3,380
task-switching	140
task-switching-2	200,000

The most complicated design for which SweetPea was able to construct a CNF encoding (`task-switching-2`) took the longest, at 200 seconds. CryptoMiniSat solved all other experiments in less than a second. Since this tool only generates a single solution, the practical time consideration is of minor importance here.

3.3.2 KUS

KUS [?] is another SAT sampler developed by some of the same team behind UniGen. However, it approaches the problem quite differently. Rather than using hash functions to partition the solution space, KUS requires a compiled deterministic decomposable negation normal form (d-DNNF) of a formula. 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 sample solutions using this form uniformly. 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 d-DNNF form. Though the number of variables and clauses in the CNF for a formula is typically modest, compiling the equivalent d-DNNF has proven intractable. Table 3.6 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.

Table 3.6. d-DNNF Compilation for Sampling with KUS

Experiment Name	Time	d-DNNF	CNF	% Growth
stroop-2	14.4ms	9.7K	24K	-60%
stroop-3	1.99s	6.3M	151K	4,072%
stroop-4	Timed Out	-	303K	-
stroop-5	Timed Out	-	881K	-
stroop-congruency-balanced	48.6m	7.3G	447K	1,633,009%
stroop-response	Timed Out	-	1.5M	-
padmala-pessoa	OOM	-	13M	-
task-switching	Timed Out	-	1.2M	-
task-switching-2	”Indeterminate”	-	92M	-

For the experiments for which the d-DNNF compiler completed, table 3.7 denotes the time required to sample solutions using KUS.

As expected, sampling occurs quickly once a reasonably-sized d-DNNF file has been obtained. However, the combinatorial nature of experimental designs makes the cost of compiling and interpreting the d-DNNF prohibitive.

3.3.3 Spur

Spur [**?**] is a more recent SAT sampler that employs 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 did not exist at SweetPea's inception. However, we

Table 3.7. 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	-

consider it here as it represents the latest research for SAT sampling. Table 3.8 denotes the time required to generate samples for each experiment using Spur, with results similar to other tools.

Table 3.8. 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
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	-

As with other Samplers, Spur can successfully generate samples for only the smallest designs.

3.4 Sample Distribution

As has been mentioned several times, there must be a one-to-one relationship between unique trial sequences and solutions to the SAT formula in order for uniformity guarantees to carry over from the SAT sampler results back to the original problem space. In this section, we will examine the distribution of the trial sequences produced by UniGen, KUS, and Spur, providing empirical evidence that the resulting trial sequences follow a uniform distribution.

We sampled 10,000 trial sequences for the Stroop-3 experiment using UniGen, KUS, and Spur. The resulting samples follow a uniform distribution; each sample only occurred once in the vast majority of cases.

3.4.1 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. UniGen generated 333 sequences twice and only generated six three times. It generated no sequences more than three times.

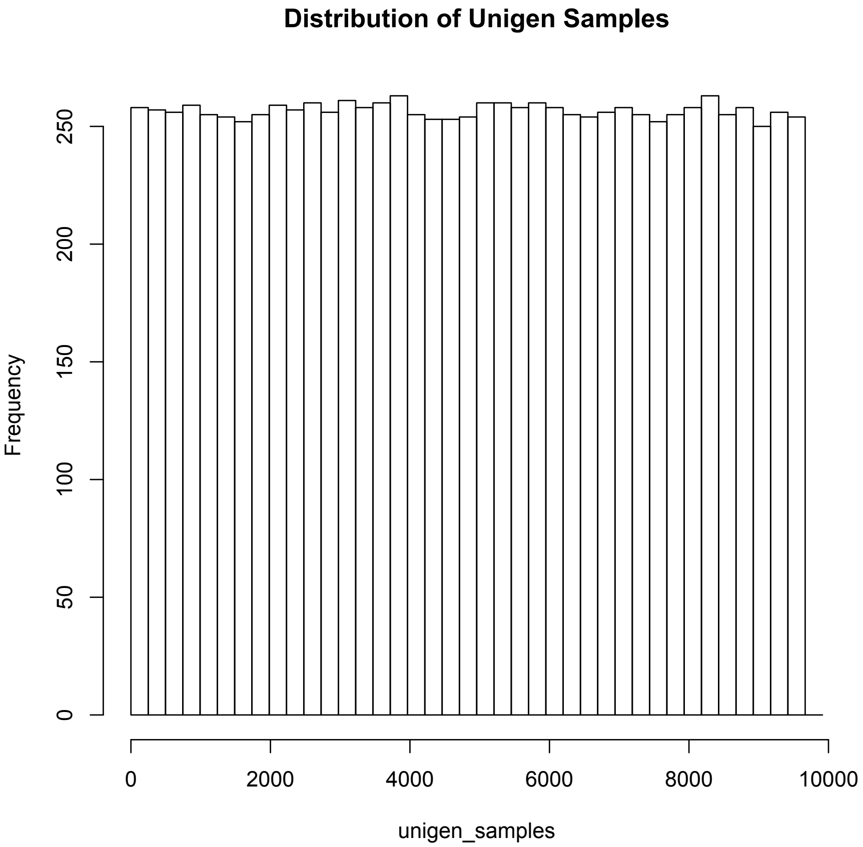

Figure 3.2. Distribution of Sequences Sampled by UniGen

3.4.2 KUS

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

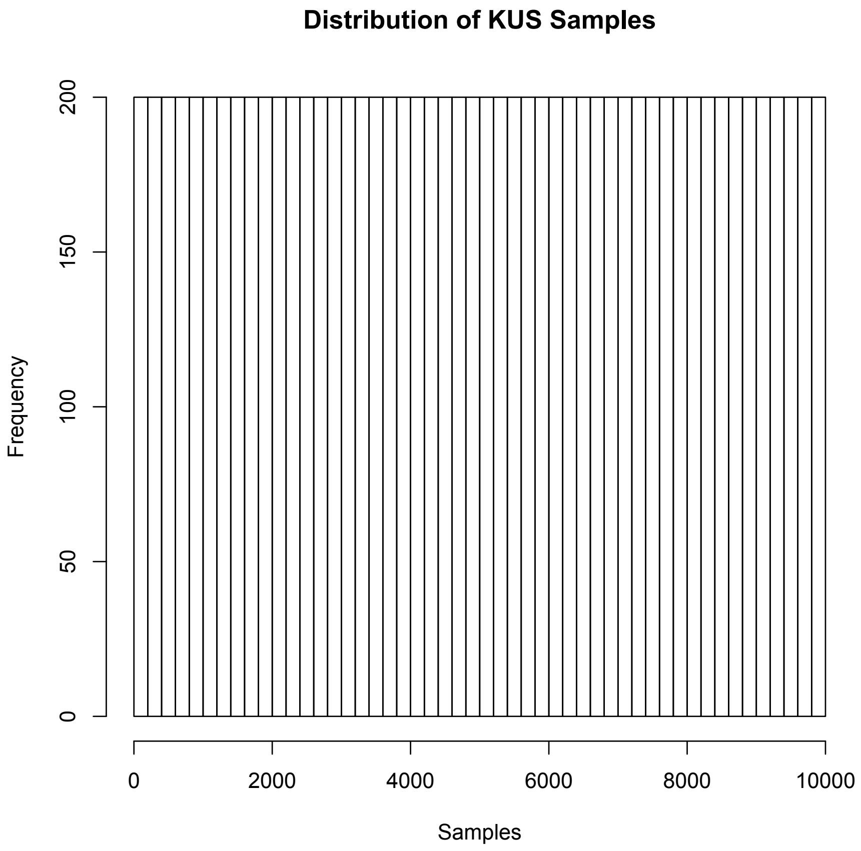

Figure 3.3. Distribution of Sequences Sampled by KUS

3.4.3 Spur

Spur sampled 10,000 trial sequences, 9,664 of which were unique. It generated three hundred thirty-two sequences twice and two sequences three times. It generated no sequences more than three times.

3.4.4 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

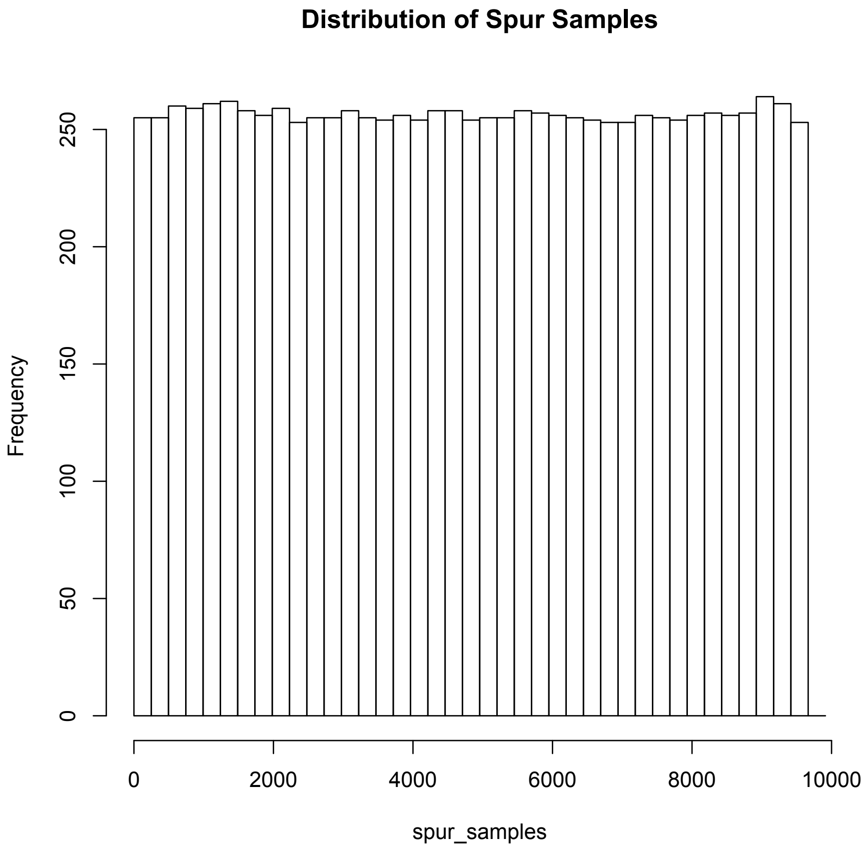

Figure 3.4. Distribution of Sequences Sampled by Spur

are generated each time. We used CryptoMiniSat to generate 10,000 samples as well, in groups of 100. It generated precisely the same set of 100 samples each time.

Of all the Samplers tested, KUS produced the most uniformly distributed samples.

3.5 Summary

We had hoped that UniGen would be able to generate uniformly-distributed solutions for experimental designs, in particular because the volume of variables and clauses in our generated CNF files fell within the range of example benchmarks provided by the UniGen authors. However, we overlooked the role played by the magnitude of the independent support set. Because UniGen requires the size of the independent support set to be less than 100 variables, there is a limit to the number of solutions a formula may have in order for UniGen to be effective. For realistic experimental designs, the number of solutions quickly outstrips this bound. Furthermore, none of the additional tested tools for sampling SAT formulae are capable of handling the scale of practical experimental designs. Although these experiments contain a reasonable number of variables and clauses, the factorial explosion in the solution space, as well as the magnitude of the independent support set, proved too large.

CHAPTER 4

SOLUTION COUNTING

Today, SweetPea offloads the entire sampling process to an external tool. While an attractive option from an implementation standpoint, this strategy does not scale to the degree needed by SweetPea. As demonstrated in Chapters 1 and 3, the solution spaces from which SweetPea is trying to sample are beyond the capacity of any existing tools. One of the shortcomings of this approach is that external tools cannot exploit domain knowledge of the problem to guide their decisions.

One may view experimental designs in SweetPea as combinatorics problems, each with a countable set of solutions. If a formula for counting the solutions to an experimental design could be derived, then it may also be possible to discover a bijection between solutions to the design and the natural numbers. If a corresponding natural number could uniquely identify every possible solution, then we could guarantee uniformity by randomly sampling natural numbers from the uniform distribution. As a first foray into solution counting, this chapter presents such a formula for tier one, two, and three designs. This exploration will also defer treatment of the set of external design constraints.

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. As explained by Brualdi [**?**], the standard formula for computing the number of permutations of r items taken from an n-element set is:

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

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* [**?**]. Both of these principles will be used to calculate the number of solutions to a SweetPea design.

4.2 Partition of an Experimental Design

An experimental design is composed of three elements: a set of factors, a subset of the factors that form the crossing, and a set of constraints. The set of crossed factors fundamentally shapes the generated trial sequences. The number of combinations of level values taken from each crossed factor governs the length of a trial sequence. 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 in the crossing; hence, trial sequences in this design will be nine trials long. The sequence length is a critical factor in the solution count, as are other factors that are not in the crossing.

To formalize the relationship between each factor and the total solution count, we define a partition for the set of factors in the design. Let D be the set of all factors in the design. Let C be the set of factors that form the crossing, and $\overline{C}$ be the set of factors not included in the crossing. Therefore $C \subseteq D$ and $\overline{C} \subset D$. We can further divide C and $\overline{C}$ by basic and derived factors. Let C_B be the set of basic factors in C, and C_D be the set of derived factors in C. $\overline{C}_B$ and $\overline{C}_D$ are defined as the basic factors not in the crossing and the complex factors not in the crossing respectively.

Lastly, we further divide $\overline{C}_B$ into two more sets. Although the set of crossed factors does not include basic factors in $\overline{C}$, this does not imply that they are fully independent. Factors in $\overline{C}_B$ may contribute to the selected levels for factors in C_D. We will refer to these factors in $\overline{C}_B$ as *source* factors, as they are at least a partial source of 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 is no benefit to doing so. Because the crossing is the highest source of authority, 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: $\{C_B, C_D, \overline{C}_{B_S}, \overline{C}_{B_I}, \overline{C}_D\}$. A visual hierarchy of this partition is shown in figure 4.1.

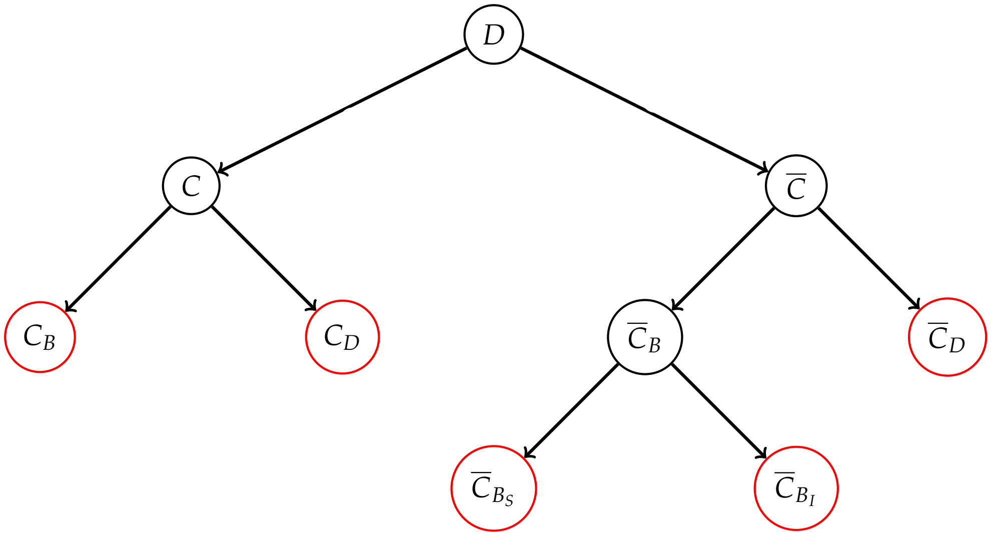

Figure 4.1. Partition of an Experimental Design

Lastly, we establish notation for referrings to individual level combinations in C. Let X be a set of sequences $t_1, t_2, \cdots t_n$, representing the combinations of levels for all factors in C. Each sequence 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 sequence t must be unique, and there is an sequence for every possible combination of levels from factors in C.

Recall the earlier design example with factors to represent the color and text of a printed word. Both the color and text factors had three levels: `red`, `green`, and `blue`. For this crossing, X can be visualized in Table 4.1.

By the multiplication principle, there will be $n = C_1 \cdot C_2 \cdots C_{|C|}$ such sequences, 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 can develop the formula for counting the number of solutions, s, for a given experimental design. We will first derive the formula for a tier 1 design, and then add additional terms for tier 2 and tier 3 designs.

Table 4.1. Expressing Level Combinations as Sequences

	i_1	i_2
t_1	red	red
t_2	red	green
t_3	red	blue
t_4	green	red
t_5	green	green
t_6	green	blue
t_7	blue	red
t_8	blue	green
t_9	blue	blue

4.3.1 Tier 1

For tier 1 designs, the sequences in X fully represent all possible level combinations in the design. $C_B \neq \emptyset$, while every other set in the partition is empty. The only variation between trial sequences is the ordering of pairs in X. Therefore, every permutation of X represents a unique trial sequence, which means there are $s = l!$ solutions to a tier 1 design.

4.3.2 Tier 2

Tier 2 designs allow basic factors that are not in C, and are thus less constrained. In otherwords, starting with tier 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, applying the multiplication principle reveals that there are $|f|^l$ possible sequences of levels of f. If $|\overline{C}_{B_I}| > 1$, then the multiplication principle is applied repeatedly to determine the total number of combinations for all factors in $\overline{C}_{B_I}$. Applying the multiplication principle a final time produces the final formula for tier 2:

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

4.3.3 Tier 3

Tier 3 designs allow the addition of derived factors. It is now possible that all sets in the partition are non-empty. C_D, $\overline{C}_{B_S}$, and $\overline{C}_D$ are the only sets in the partition remaining to account for. 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), regardless of their relationship to other factors. $\overline{C}_D$ also requires no special treatment as the level selection for uncrossed derived factors depends entirely on the levels selected for basic factors. (Regardless of whether the basic factors are in C_B, $\overline{C}_{B_S}$, or $\overline{C}_{B_I}$.) Only $\overline{C}_{B_S}$ remains, containing the uncrossed basic factors that feed crossed derived factors.

Derived factors allow the user to provide an arbitrary predicate for each level in the factor. SweetPea applies these predicates to every combination of levels that could be given as arguments, effectively constructing a truth table, to determine which level to select for each combination of arguments. We define X_S to be the set of sequences 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. Checking compatibility is necessary as some combinations in X may preclude certain combinations in X_S.

For every sequence $t \in X$, there is some number of elements 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,[1] $pred(d)$. 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.[2] If any 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, a list of subsets of X_S remains, 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 once more to complete the formula for tier 1, 2, and 3 designs:

[1] These 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. SweetPea could minimize this concern by ensuring that each predicate is only applied once to each argument combination and then caching the results.

[2] If X is very large, then testing all combinations could become intractable. However in practice, X is relatively small, typically $|X| < 500$. Additionally, SweetPea's current implementation relies on X being small enough for this to work, as it follows a similar process when generating the SAT encoding for derived factors.

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

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

It is possible for the user to specify a crossing that is not satisfiable. Referring back to the Stroop example, the user may include all three factors, `color`, `text`, and `congruent`, in the crossing. This is unsatisfiable because some level combinations violate the rules of the derived factor. For example, `color=red`, `text=blue`, and `congruent=yes` is invalid because `red` and `blue` are not congruent according to the predicates for `congruent`. If such a crossing is specified, the formula for s will correctly identify the number of potential solutions as zero.

4.4 Example

Consider a design composed of the same factors as the Stroop example:

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

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

However, rather than crossing `color` and `text`, we will demonstrate an example in which `color` and `congruent?` are crossed. For this design, the partition of D becomes:

$$C_B = \{\texttt{color}\} \quad C_D = \{\texttt{congruent?}\} \quad \overline{C}_{B_S} = \{\texttt{text}\}$$

$$\overline{C}_{B_I} = \emptyset \qquad\qquad \overline{C}_D = \emptyset$$

The crossing, or X, is the cartesian product of levels of `color` and `congruent?`. SweetPea generates this product exhaustively.

$$X = \{(red, yes), (red, no), (green, yes), (green, no), (blue, yes), (blue, no)\}$$

The size of the crossing determines the sequence length, therefore $l = |X| = 6$. As a result, the first term, $l!$, becomes $6! = 720$.

In this design, there are no basic independent factors outside the crossing: $\overline{C}_{B_I} = \emptyset$. Therefore there is no value for the second term.

For the third and final term, we need to first determine X_S, the cartesian product of levels of factors in $\overline{C}_{B_S}$. For this design, the only factor in $\overline{C}_{B_S}$ is `text`. As a result, X_S is simply the levels in `text`:

$$X_S = \{red, green, blue\}$$

At this point, each element of the product of X and X_S must be checked for validity under the derived factor definitions in the design to determine the subsets of X_S that are valid for each element in X. In terms of this example, which level selections in X_S do *not* violate the definition of `congruent?` when paired with each level selection combination in X? These subsets comprise S as defined in 4.3.3. Table 4.2 gives a visual representation of this process.

Table 4.2. Computing S for Solution Counting

	X_S			
X	red	green	blue	J
(red, yes)	✓			$\{(red\}$
(red, no)		✓	✓	$\{(green, blue)\}$
(green, yes)		✓		$\{(green\}$
(green, no)	✓		✓	$\{(red, blue)\}$
(blue, yes)			✓	$\{(blue\}$
(blue, no)	✓	✓		$\{(red, green)\}$

S is therefore:

$$S = \{(red), (green, blue), (green), (red, blue), (blue), (red, green)\}$$

With S defined, we may now compute the third term of the formula:

$$\prod_{k=1}^{|S|} |J_k| = 1 \cdot 2 \cdot 1 \cdot 2 \cdot 1 \cdot 2 = 8$$

This yields the final solution count of $s = 720 \cdot 8 = 5,760$. One may confirm this solution by generating the corresponding SAT encoding for this design and applying a model counter to the generated CNF.

4.5 Conclusion

In this chapter, we derived a formula for counting the number of solutions to SweetPea designs in tiers 1, 2, or 3. However, this formula does not account for the third component of a design: constraints. Including constraints in the formula is beyond the scope of this thesis, as it would involve complexity similar to counting solutions for tier 4 and five designs. The formula, therefore, produces an exact solution count for designs lacking constraints and produces an upper-bound for designs with constraints. Next, we will apply the counting formula to construct uniformly-distributed trial sequences without relying on a SAT sampler.

CHAPTER 5

SEQUENCE CONSTRUCTION

With an exact count of viable sequences for tier one, two, and three designs, a mapping from natural numbers to valid trial sequences is now a possibility. Such a mapping would prove valuable for guaranteeing a uniform distribution of samples, as well as generating solutions more efficiently. This chapter presents an algorithm that is guaranteed to generate uniformly-distributed solutions to tier one, two, and three designs. This is accomplished by randomly sampling natural numbers from the uniform distribution in the range of $[0, s)$, then generating the unique trial sequence associated with that number. Lastly, the generated trial sequence is checked against design constraints (if any exist), to ensure that no violations exist in the generated sequence.

5.1 Enumerating Permutations and Combinations

There are several steps involved in constructing a solution based on a natural number. In this section, we will review 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. These techniques form the basic building blocks for the final construction algorithm.

5.1.1 Interval Mapping

The construction algorithm will repeatedly need to take a single number and map it into a unique combination of multiple numbers, each from a possibly different range. We will refer to this process as *interval mapping*.

Suppose we have n numeric intervals, each ranging from $[0, r_n)$. Revisiting the multiplication principle, we can see that there are $m = \prod_1^n r_n$ unique combinations of individual values from each interval. If j is a number selected from $[0, m)$, we can map j to its unique combination of interval value selections using the modular arithmetic. (This technique will still work even when $j \geq m$, but combinations will begin to repeat.)

We begin by computing $j \bmod r_1$, 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 $r_2, r_3, ..., r_n$ until the full combination is developed. For example, if we had $U = \{3, 2, 4\}$ as the set of upper bounds for each interval, then there would be $3 \cdot 4 \cdot 2 = 24$ possible combinations of interval values. Suppose we want to compute the interval selections for 15. This would be done as follows:

$$\begin{array}{rrr}
j = 15 & R_1 = 15 \bmod 3 = 0 & j = \left\lfloor \dfrac{15}{3} \right\rfloor = 5 \\
j = 5 & R_2 = 5 \bmod 4 = 1 & j = \left\lfloor \dfrac{5}{4} \right\rfloor = 1 \\
j = 1 & R_3 = 1 \bmod 2 = 1 &
\end{array}$$

Thus the set of selected values from each interval is $R = \{0, 1, 1\}$. Figure 5.1 demonstrates this visually.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	
0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	%3
0			1			2			3			0			1			2			3			%4
0												1												%2

Figure 5.1. Decomposing a Number into Constituent Ranges

This technique will be applied multiple times during sequence construction to select a unique combination of values corresponding to a single integer.

5.1.2 Constructing Permutations

The construction algorithm also needs to map an individual number to a corresponding permutation of a set. Every permutation of a set can be identified by its *inversion* or *inversion sequence*. Loosely defined, the inversion sequence indicates how out-of-order the permutation is when compared with the original sequence. The concept of an inversion was first introduced by Hall [**?**], though we will use on Brualdi's description [**?**] here.

Let S be the set of numbers $\{1, 2, ..., n\}$ Suppose we have some permutation p of S, $i_1, i_2, ..., i_n$. 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, $a_1, a_2, ...a_n$, 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 sequence can be interpreted to mean that three elements larger than one precede one, zero elements larger than two precede two, two elements larger than three precede three, one element larger than four precedes four, and zero elements larger than five precede five. In our case, the original sequence will always be ascending; therefore, the last inversion must always be zero.

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 $a_1, a_2, ...a_n$, begin with n empty locations. Beginning with a_1, skip a_i empty locations before inserting the i^{th} element of S. Figure 5.2 demonstrates this algorithm in the previous example.

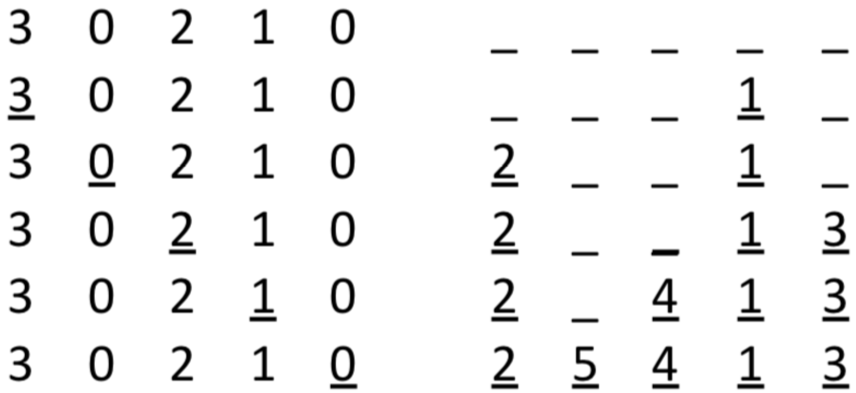

Figure 5.2. Construction of a Permutation from its Inversion Sequence

Finally, we can construct the inversion sequence for the j^{th} permutation of a set by applying the interval mapping technique introduced in 5.1.1. 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 set of upper bounds for the intervals is $U = \{n, n-1, n-2, ..., 2, 1\}$. Using interval mapping to construct an inversion sequence, we can deterministically construct any permutation of a set.

5.1.3 Combinations

We can enumerate combinations of a set of elements in a similar manner. For a combination of r elements, taken from a set of k elements, the multiplication principle dictates that there are k^r possibilities. Applying interval mapping with r intervals, each from $[0, k)$, suffices to enumerate each combination. The selected values from each interval identify the chosen items from the original set.

5.2 Construction Algorithm

Now that we have established techniques for counting solutions and constructing specific permutations and combinations based on a numeric index, we are ready to develop the full construction algorithm. The construction algorithm itself mirrors the structure of the counting formula. Each term in the counting formula becomes an upper-bound for an interval, to which interval mapping can be applied, followed by the construction of the specific permutation or combination represented by each term. 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. Determine the selected levels for factors in $\overline{C}_D$ by applying the corresponding predicates to the previously selected levels for basic factors.

Accomplishing Step 1 was the subject of a previous chapter; 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 broad range. Step 3 is where most of the work lies, particularly in selecting the ranges for interval mapping.

5.2.1 Interval Mapping for an Experimental Design

For tier 1, 2, and 3 designs, as we saw in 4.2, there exists a partition of the factors comprised of 5 sets: $\{C_B, C_D, \overline{C}_{B_S}, \overline{C}_{B_I}, \overline{C}_D\}$. This section will describe how the selected sequence number identifies the level selections for all factors in each set. The outcome will be a set of upper bounds to which interval mapping can be applied to identify aspects of the sequence. 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 combined with each other to form the crossing, labeled X in Chapter 3. Recall also that $|X|$ governs the length of a generated trial sequence: $l = |X|$, and that there are $l!$ permutations of the sequences in X. Therefore, the first interval, which identifies which permutation of X to use, is $l!$. Selecting a single integer from this range will identify a permutation that fully constrains the level selection for each trial for all factors in C_B and C_D. Therefore 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 the crossing but are sources of data for derived factors in the crossing. More precisely, $\overline{C}_{B_S}$ contains basic factors not in C_B, but that influence factors in C_D. Because the crossing constrains the levels selected for factors in C_D, it also transitively constrains potential level choices for factors in $\overline{C}_{B_S}$.

Recall from Chapter 4 that we defined X_S to represent the source crossing or the set of sequences enumerating all combinations of levels for factors in $\overline{C}_{B_S}$. We also established that there is a distinct subset of X_S, J_n, that contains sequences from the source crossing that are compatible with the n^{th} sequence in X. We also defined S to be the set of these subsets. A distinct interval for each of these subsets is required. Therefore we add l values to U: one for the size of each subset in S. U will now contain $l + 1$ elements: $U = \{l!, |J_1|, |J_2|, ..., |J_l|\}$.

Lastly,[1] we consider $\overline{C}_{B_I}$. 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 is added for each factor f with upper bound $|f|^l$. (Recall

[1] $\overline{C}_D$ has been skipped, 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 to account for factors in this set.

that $|f|$ denotes the number of levels of factor f.) U is now complete:

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

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

Once a sequence number i has been selected (Step 2), and U has been constructed, interval mapping is applied to map i into its constituent pieces representing the final sequence. The mapped value from each range identifies a specific permutation or combination for that aspect of the sequence. The bulk of the trial sequence can be assembled using these values and methods for constructing indexed permutations and combinations. Finally, we select levels for factors in $\overline{C}_D$ by applying their corresponding predicates to the selected levels for each trial.

5.2.2 Mapping Selected Values to Sequences

Once interval mapping has produced values for each interval, SweetPea must convert the value into its associated permutation or combination in the original design. The first value, U_0, identifies a particular permutation of the crossing, the next l values, U_1 to U_l, identify a particular source combination for the l^{th} trial in the sequence,[2] and the remaining values beginning with U_{l+1} identify particular l-combinations to be used for each independent basic factor in the sequence. Each of these three types of values identifies a permutation or combination that spans a different dimension of the final sequence. Figure 5.3 gives a visual representation to assist the reader.

The selected permutation value (U_0) is converted to a unique inversion sequence by again applying interval mapping. The inversion sequence is then used to construct the associated permutation using the technique introduced in 5.1.2. The algorithm then produces level values by using each integer in the permutation as an index into the sequences in X.

[2] Ordering is important here due to the association between valid source combinations and specific sequences in the crossing. The same permutation that scrambles the crossing must also be applied to the set of source combinations to preserve the integrity of the association.

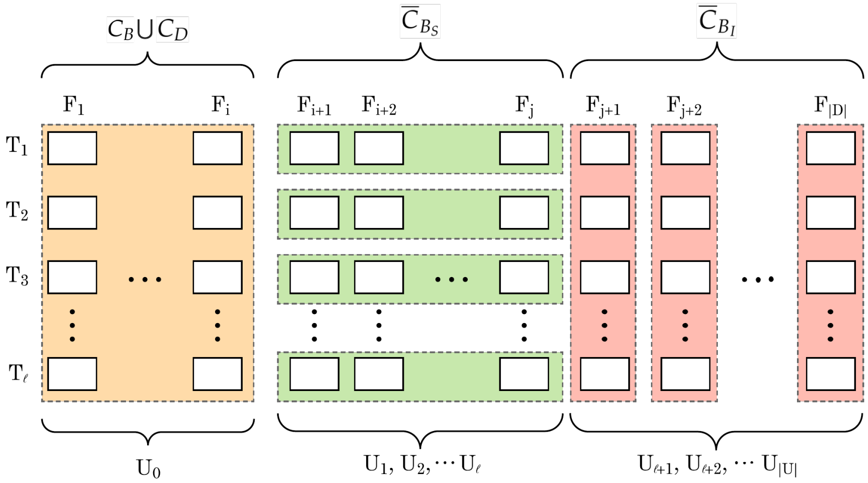

Figure 5.3. Dimensionality of Interval Values

The algorithm converts the selected values for independent basic factors (U_{l+1} to $U_{|U|}$) similarly. Each selected value identifies a combination index, which the algorithm converts to a set of values using interval mapping. Each value is an index into the list of levels for the associated factor.

Lastly, the algorithm must convert the selected values for source combinations (U_1 to U_l). These are more complex than the previous conversions because a set of valid source combinations is associated with an individual trial in the sequence. 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. Therefore care must be taken to preserve this association. The algorithm preserves this association by using the original trial index when selecting the set of valid source combinations. Figure 5.4 outlines the entire mapping process.

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.

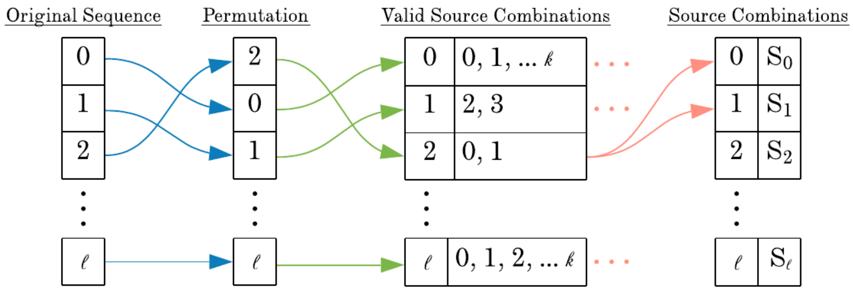

Figure 5.4. Selecting Valid Source Combinations

5.3 Example

We will now use this method to construct a single solution to the design from the counting example in the previous chapter. We will choose a solution at random from $[1, 5760]$: 2590.

First, we compute the set of value selections, R, for the $2,590^{th}$ solution. In order to do this, we first compute U, the set of upper-bounds for each range. The first upper bound is the number of permutations, $l!$, for the design. For this example, this was determined to be 720. Next, we add the size of each subset of S (See Table 4.2) to U to select valid source combinations. U now contains 7 elements: $U = \{720, 1, 2, 1, 2, 1, 2\}$. Because $\overline{C}_{B_I} = \emptyset$ and $\overline{C}_D = \emptyset$ in this example, there are no other bounds to compute, so U is complete.

When interval mapping is applied to 2590 to generate values from these ranges, R is determined to be $R = \{430, 0, 1, 0, 1, 0, 0\}$. With R computed, we can now use the selected values to generate the correct permutation, and then select level combinations for the remaining factors.

To generate the 430^{th} permutation, we first construct the 430^{th} inversion sequence by again applying interval mapping with $U = \{6, 5, 4, 3, 2, 1\}$. This yields $\{4, 1, 2, 0, 1, 0\}$ as the 430^{th} inversion sequence. Applying the algorithm to convert this sequence to its corresponding permutation, we obtain $\{3, 1, 5, 2, 0, 4\}$. The crossing, X, was identified in Chapter 4 to be:

$$X = \{(red, yes), (red, no), (green, yes), (green, no), (blue, yes), (blue, no)\}$$

The permutation $\{3, 1, 5, 2, 0, 4\}$ therefore corresponds to:

$$\{(green, no), (red, no), (blue, no), (green, yes), (red, yes), (blue, yes)\}$$

We have now selected levels for factors in C_B and C_D. The only remaining factor is `text`, which is in $\overline{C}_{B_S}$. The source combination selections from R for this factor were $\{0, 1, 0, 1, 0, 0\}$. We then use the values from the permutation as indices to select both the source combination list and the selected value from that list for each trial. This process is represented visually in Figure 5.4. In Chapter 4 we established the set of source combination lists to be:

$$S = \{(red), (green, blue), (green), (red, blue), (blue), (red, green)\}$$

The first value of the permutation is 3, which corresponds to the source combination list $(red, blue)$, and a value selection of 1. Therefore the level selected for `text` in the first trial is *blue*. Following the same pattern, the level selections for `text` in each trial are:

$$\{blue, blue, red, green, red, blue\}$$

Another perspective is that we are applying the same permutation computed previously to the list of level selections and the list of source combinations, before doing the final lookup. Combining all level selections yields a complete trial sequence:

Table 5.1. The 2590^{th} Trial Sequence

Trial	Color	Text	Congruent
1	*green*	*blue*	*no*
2	*red*	*blue*	*no*
3	*blue*	*red*	*no*
4	*green*	*green*	*yes*
5	*red*	*red*	*yes*
6	*blue*	*blue*	*yes*

5.4 Handling Constraints

Recall that the complexity gradation did not account for the external constraints that may be part of the design. Such constraints typically enforce some 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 consecutive congruent trials. Constraints may be applied to the levels of derived factors as well; therefore, the user 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. Such improvements will be discussed further in Chapter 6. In the meantime, however, rejection sampling is applied to ensure that the algorithm does not ever return sequences that violate constraints. The generated sample is checked against all repetition constraints in the design to verify that no violations exist. As will be seen in the next section, this performs adequately in practice.

5.5 Benchmarks

Of the ten benchmarks designs that were introduced in Chapter 3, four can be classified as tier 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 5.2 reviews the basic experiment data for each benchmark design. Table 5.3 summarizes the time required to generate 10,000 samples for each design using a single core, as well as the average number of rejections per sample. (Rejections caused by constructing a trial sequence that violated external constraints.)

Table 5.2. Benchmark Experiments for Sequence Construction

Experiment Name	Length	Unconstrained Solutions	Single Sample
stroop-2	4	24	0.0009s
stroop-3	9	362,880	0.003s
stroop-4	16	20,922,789,888,000	0.007s
stroop-5	25	$\approx 2^{83}$	0.016s
stroop-10	100	$\approx 2^{524}$	0.170s
stroop-20	400	$\approx 2^{2886}$	2.729s
stroop-20-extra-constraints	400	$\approx 2^{2886}$	3.234s

Even for huge designs, this approach constructs a single random sample exceptionally quickly.

Table 5.3. Sampling Metrics

Experiment Name	10,000 Samples	Avg. Rejections/Sample
stroop-2	8.763s	0.982
stroop-3	29.323s	1.393
stroop-4	72.882s	1.53 8
stroop-5	155.699s	1.589
stroop-10	1908.598s	1.698
stroop-20	26064.417s	1.684
stroop-20-extra-constraints	27338.002s	1.898

Generating a large number of samples can still take a significant amount of time. However, because samples can be generated independently of one another, this algorithm can be easily parallelized for a linear speedup in the number of available cores.

CHAPTER 6

FUTURE WORK

This thesis has presented an approach for constructing trial sequences for some 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 sample construction is nearly instantaneous. This approach represents a vast improvement over the prior approach of generating solutions using SAT sampling. However, there are many opportunities for improvement and additional research.

The counting and construction methods presented here apply to several tiers of experimental designs, yet they do not account for external constraints in the block. They only ensure that no violations of internal constraints (between basic and derived factors) exist and rely on rejection sampling to discard generated samples that violate external constraints. While rejecting sampling is sufficient in practice for many designs, it is possible to construct designs with external constraints that would make the solution space so sparse as to render rejection sampling impractical. Further research to incorporate constraints in the counting and construction process would be worthwhile. We could likely apply combinatoric techniques for generating permutations with prohibited patterns to improve this process.

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

As an implementation detail, the current construction algorithm is single-threaded, generating all samples serially. However, there is no dependent relationship between samples; therefore, we can fully parallelize the construction algorithm for a linear speedup in the number of available cores.

We could also research other approaches which do not rely wholly upon direct construction. Various hybrid approaches have been considered but not thoroughly researched. As an example, a numeric construction approach could be used to construct a partial trial sequence which could then be completed using a SAT sampler. Such a hybrid approach would make the sample generation more efficient for all designs, but care would need to be taken to preserve uniformity guarantees. A partially constructed sequence may over prune the remaining search space, leading to biased results.

Lastly, improvement opportunities exist in the SAT encoding itself. The naive encoding used currently is easy to understand and reason about but is wasteful in its representation. It uses far more variables than are theoretically required to represent the data. If we research other types of encoding schemes, we may find an alternative that could improve performance. Such an encoding would still likely need to avoid encoding all permutations as distinct solutions to avoid a combinatoric explosion. A hybrid approach, the inverse of the previous paragraph, would likely perform well so long as the size of the independent support set was kept small: 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 values could be populated using a construction approach. Splitting the complexity challenges this way would reduce the burden on the SAT tooling and potentially simplify the construction algorithm. In the same vein, SweetPea could also begin utilizing the native `XOR` support offered by CryptoMiniSat, which may also offer performance improvements.

CHAPTER 7

CONCLUSION

The current implementation of SweetPea provides a useful standard language for defining experimental designs. However, it has not yet fully achieved its objective to generate uniformly-distributed trial sequences conforming to these designs. This failure is due primarily to the inability of current SAT sampling tools to cope with such large and complex solution spaces. While SAT tools are still useful for quickly generating solutions to in-progress designs, SweetPea has not yet been successful in guaranteeing uniformly-distributed samples for arbitrary designs using these tools.

This thesis has presented background information and benchmarks detailing these shortcomings and has introduced a complexity gradation for discussing experimental designs of varying complexity. It also introduced methods for enumerating arbitrary solutions to tier one, two, and three designs. These methods allow efficient generation of samples which are guaranteed to follow a uniform distribution for these design tiers. As these techniques do not yet account for external design constraints, rejection sampling is applied to ensure that no generated sequences violate these constraints.

While not yet able to achieve SweetPea's objectives for all experimental designs, this approach is hugely beneficial for simple designs. Uniformly-distributed samples can be generated exceptionally quickly, and have the added benefit of being numbered. It also provides a framework and pattern upon which to build for more complicated designs. We expect that these methods can be extended or hybridized to guarantee uniformly distributed samples to all experimental designs.

REFERENCES

[1] D. Achlioptas, Z. S. Hammoudeh, and P. Theodoropoulos, *Fast sampling of perfectly uniform satisfying assignments*, in Theory and Applications of Satisfiability Testing – SAT 2018, O. Beyersdorff and C. M. Wintersteiger, eds., Cham, 2018, Springer International Publishing, pp. 135–147.

[2] R. A. Brualdi, *Introductory combinatorics*, Upper Saddle River, N.J.: Pearson/Prentice Hall, 2010.

[3] S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi, *On Parallel Scalable Uniform SAT Witness Generator*, in Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2015, pp. 304–319.

[4] 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.

[5] S. Chakraborty, K. S. Meel, and M. Y. Vardi, *A scalable approximate model counter*, CoRR, abs/1306.5726 (2013).

[6] S. Chakraborty, K. S. Meel, and M. Y. Vardi, *Balancing Scalability and Uniformity in SAT-Witness Generator*, in Proceedings of Design Automation Conference (DAC), 2014, pp. 60:1–60:6.

[7] A. Cherkaev, *Sweetpea: A language for experimental design*, Master's thesis, The University of Utah, ProQuest, 12 2018.

[8] C. P. Gomes, J. Hoffmann, A. Sabharwal, and B. Selman, *Short xors for model counting: From theory to practice*, in Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing, SAT'07, Berlin, Heidelberg, 2007, Springer-Verlag, pp. 100–106.

[9] C. P. Gomes, A. Sabharwal, and B. Selman, *Near-Uniform Sampling of Combinatorial Spaces Using XOR Constraints*, in Advances in Neural Information Processing Systems 19, B. Schölkopf, J. C. Platt, and T. Hoffman, eds., MIT Press, 2007, pp. 481–488.

[10] 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.

[11] 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.

[12] 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.

[13] M. Soos and K. S. Meel, *Bird: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting*, in Proceedings of AAAI Conference on Artificial Intelligence (AAAI), 1 2019.

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

[15] J. R. Stroop, *Studies of interference in serial verbal reactions.*, Journal of Experimental Psychology, 18 (1935), p. 643.

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