A STOCHASTIC GRAPH GRAMMAR ALGORITHM FOR INTERACTIVE SEARCH

Matthew I. Campbell
Associate Professor
Automated Design Laboratory
Department of Mechanical Engineering
University of Texas at Austin
Austin, Texas 78712-0292
mc1@mail.utexas.edu

Rahul Rai
Assistant Professor
Advanced Design Optimization Laboratory
Department of Mechanical Engineering
California State University at Fresno
Fresno, California 93740
rarai@csufresno.edu

Tolga Kurtoglu
Research Scientist
Mission Critical Technologies
Intelligent Systems Division
NASA Ames Research Center
Moffett Field, California 94035
tolga.kurtoglu@nasa.gov

ABSTRACT
This paper presents a new search method that has been developed specifically for search trees defined by a generative grammar. Generative grammars are useful in design as a way to encapsulate the design decisions that lead to candidate solutions. Since the candidate solutions are not confined to a single configuration or topology and thus useful in conceptual design, they may be difficult to computationally analyze. Analysis is achieved in this method by querying the user. The user interaction is kept to a maximum of thirty pair-wise comparisons of candidates. From the data gathered from the comparisons, a stochastic decision making process infers what candidate solutions best meet the user’s preference. The method is implemented and applied to a grammar for tying neckties. It is shown through 21 user experiments and 4000 automated experiments that the method consistently finds solutions within the 99.8 percentile. The implications of this method for conceptual design are expounded on in the conclusions.

Keywords: Graph grammars, interactive design evaluation, stochastic search, generative methods.

1 INTRODUCTION
The use of generative grammars to develop new design concepts is a powerful approach to represent different topologies, configurations, or shapes within a single search space. This capability approaches the representational capacity of human designers that create and compare wildly different solutions during the conceptual phase of the design process. As a result, researchers are interested in how grammars can be used to interactively guide human designers or even automatically find promising concepts.

Generative grammars are comprised of rules for manipulating a seed state into feasible candidate solutions. A typical grammar rule contains a left-hand side (LHS) of conditions that determined when it is applicable, and a right-hand side (RHS) that contains the transformation instructions. As shown in Figure 1, the rules build upon a seed graph to create a tree of possible solutions. For each grammar rule who’s LHS successfully matches with a host state in the tree. A potential transition to a new state is defined. The collected set of recognized rules along with their operands (locations in host) defines the set of options for the host state. Ideally, the rules capture a certain type of design knowledge that is inherent to the problem, and are formulated in such a way that states with no valid rules (“leaves in the tree”) are valid and complete design candidates.

While a generative grammar defines a space or tree of solutions, it does not include a means to search this space. Such search trees often produce an intractably large set of solutions.
that are too large to search manually, and sometimes too large to store computationally. What is needed then is some way to evaluate concepts and prune or target specific areas of the tree that best meet the designer’s expectations.

However, automated evaluation is difficult in conceptual design for several reasons. First, concepts often lack detailed dimensions needed to perform meaningful computational analysis. Second, it is often difficult to create evaluation routines that can handle the wide variety of configurations that are explored at this stage. Finally, there is often more than one performance parameter, which needs to be found to make a meaningful comparison between concepts, thus requiring an approach to handle multi-objective decision-making.

The approach taken here is to present a sample of solutions to the designer in hopes to glean what differentiates good concepts from bad concepts and to use that information to find better solutions in the search tree. In this manner, the computational process need not be concerned about whether the designer is evaluating cost, size, efficiency, aesthetics, or some combination of performance parameters. It merely takes the user input and seeks solutions to maximize the designer’s preference. This is accomplished by reflecting the user input onto the generative grammar rules. Rule-centered data accrues through various user feedback sessions and is used to determine which rules, combination of rules, ordering of rules, or when in the tree to execute a particular rule leads to the best design. At each decision point in the search tree, a stochastic probability is assigned to the available rule choices based on this gathered data. This paper describes the approach in detail (Section 4) and presents the results of an experiment that validates the effectiveness of the approach (Section 5).

In order to complete the experiments an example generative grammar must be chosen. While the authors have established various complex engineering design generative grammars [1-3], a simple fifteen-rule graph grammar is used in this paper that defines ways to tie a necktie. This simple example is chosen because the development of a computational evaluation of different neckties seems impossible, and yet humans can immediately examine one and form a consensus on whether it is big or small, symmetric or skewed, easy to create or difficult. This generative grammar is described in detail in Section 3.

In this paper, we are expanding on earlier publications that similarly prompt the user for feedback to determine the best possible solution from a large set of candidate solutions [4]. The difference in the method presented in this paper is that this new work does not explicitly store all possible solutions. Earlier work required the computational resources necessary in enumerating the complete space of solutions. Given that the created candidates only weigh a few kilobytes, it may be possible to store as many as a million candidate solutions (a million kilobytes is one gigabyte). However, a million possible solutions is often too small given the vast number of candidate solutions that may be created through most generative grammars. As a result, the method described here only creates a small subset of solutions within the large tree of potential solutions, but through a unique stochastic process, it is able to find solutions that closely model the user’s preference.

The next section describes similar research (Section 2). This is followed by the aforementioned methodology sections (3 and 4), the experiments section (Section 5), and the Discussion and Conclusions section (6).

2 RELATED WORK

Graph transformation systems, or graph grammars, reside in graph theory research as a way to rigorously define mathematical operations such as addition and intersection of graphs. Recently, engineering design researchers have discovered that graph grammars provide a flexible yet ideally structured approach for the creation of complex engineering systems.

These approaches capture the transitions or the production rules for creating a solution, as opposed to storing the solutions themselves. The initial specification is represented as a simple seed graph which defines the starting point for the to-be-designed artifact. From this initial specification, the design process can be viewed as a progression of graph transformations that lead to the final configuration. This interpretation of the design process makes graph grammars very suitable for modeling the open-ended nature of conceptual design, where designers explore various ideas, decisions, and modifications to previous designs to arrive at feasible solutions [5].

Agarwal and Cagan’s coffee maker grammar [6] was one of first examples of using grammars for product design. Their grammar described a language that generates a large class of coffee makers. Shea et al. [7] presented a parametric shape grammar for the design of truss structures that uses recursive annealing techniques for topology optimization. Other engineering applications include Brown, et al. [8], who presented a lathe grammar, Schmidt and Cagan’s grammar for machine design [9], Starling and Shea’s grammars for mechanical clocks [10] and gear trains [11]. One of the interesting implementations of grammars to function-based design is the work of Sridharan and Campbell [2]. This research showed how a set of 69 grammar rules are developed to guide the design process from an initial functional goal to a detailed function structure. Kurtoglu et al. [1] builds upon this research and significantly extends the graph grammar approach to function-based design. It starts with a function structure and seeks multiple configuration solutions for the various sub-functions of the function structure.

Design evaluation and selection, on the other hand, is another important part of the design process and has received great attention in the design literature. Perhaps, the most common concept evaluation method is the Pugh Concept Selection Charts [12]. Pugh charts use a minimal, qualitative evaluation scale to compare design alternatives in a matrix format against a number of performance criteria. Numerical concept scoring/weighting, and decision matrices [13] are similar methods and employ qualitative and quantitative evaluation scales. The Analytic Hierarchy Process (AHP) [14]
is another multi-criteria decision making technique that uses hierarchically related performance metrics. Similar in spirit to AHP, some researchers have proposed methods that are inspired from the multi-attribute utility theory [15]. The general method for these techniques is to assign a value to each performance metric, weight the value by the importance of the metric, and then aggregate the weighted scores to convert multiple metrics into a single metric.

Finally, interactive methods for design evaluation include the Design Preference Modeler (DPM) [4] – prior work by the authors. In this method, an interaction between a designer and a computational synthesis tool is established so that the designer’s decision-making during concept evaluation can be modeled. This model is later used as a guidance strategy to find best designs in a large population of alternative solutions. The computational synthesis tool generates design alternatives whereas the designer gets involved in the process by evaluating a prescribed set from these design alternatives. In managing the computer human interaction, DPM carefully selects this set from the population of candidate designs and presents them to the designer for gathering evaluation feedback. This selection is made by following a heuristic that aims to simultaneously reduce the number of required designer evaluations and capture the variety in the design solution space. The designer’s feedback is translated into a preference model that is used to automatically search for best designs. Another example of capturing designer/user preferences include the work of Osborn [16] who developed a method to quantify individual preference for aesthetic attributes in design using statistical analysis applied to a shape graph grammar. In this research, user preferences are elicited and gathered via a graphical user interface; and subsequently used as a basis for guiding the search process. Interactive genetic algorithms (IGA) [17] are a form of evolutionary computation, in which a fitness function of traditional genetic algorithm is replaced by interactive user evaluations. IGAs have been used in a wide range of applications including user-centric design optimization [18]. Finally, the manner in which artificial intelligence (AI) reasoning methods have been used in engineering design has influenced the method developed here [19-21].

3 NECKTIE KNOT GRAPH GRAMMAR

In this paper, we use a fifteen-rule graph grammar that defines ways to tie a necktie. By simply executing different combinations of grammar rules, a variety of tie knot designs can easily be generated including the traditional Four-in-Hand or the Windsor knot as well as the lesser-known Pratt knot. Four-in-Hand, the most commonly used necktie knot, originated in 19th century Europe and it is said to resemble the reins of a four-horse carriage. This tie knot is long, thin, and easy to knot but hard to untie. Later, additional types of necktie knots like Windsor, and Half Windsor came into existence. The most recent of all necktie knots, Pratt, originated in 1989 [22]. As history alludes, necktie knots were mostly discovered by chance. Recently, however, Fink and Mao [23] presented the first formal approach to represent all possible types of neckties. They formulated the problem of neckties into random walks on a triangular lattice and derived useful results.

A necktie has two ends but only the wide end is moved in tying the knot. The moving end is termed as the active end. A tie knot is started by first forming a triagonal basis. This is achieved by wrapping the active end around the neck and over or under the passive end (Figure 2a). Knots that start on the right are identical upon reflection to their left-hand counterparts. For sake of simplicity, the reflection aspect is omitted from this discussion and all ties start on the left.

![Figure 2. Rules to tie a necktie knot](image)

(a) rules for beginning a tie knot
(b) rules for continuing a tie knot
(c) rules for terminating a tie knot
The formation of a triangular basis divides the space into three regions: the right side (R), the left side (L), and the center (C). The C region denotes the region above the knot and below the throat. Each movement in tying the knot carries the active end toward one of these three areas (Figure 2a.). Each movement in tying the knot carries the active end either away from the shirt (i) or between the shirt and the rest of the tie (o). For a valid tie knot operation the direction must oscillate between (i) and (o) and no two consecutive move regions may be the same (Figure 2b).

To complete a knot, the active end must be wrapped over the front, i.e., either RoLi or LoRi, then underneath to the center, Co, and finally through the front loop (Figure 2c). At the end of the knot tying, there is a special move known as the termination move (T), in which the active end is pulled downward through the knot. This termination move has to be preceded by move Co.

Based on this representation, 15 necktie graph grammar rules were formulated. Among the 15 rules, there are two initiation rules, one termination rule and 12 intermediate move rules (Figure 3). Since necktie knot has to start with Lo or Li move, this results in two initiation rules (Rule 13 and Rule 14). Since the necktie knots can terminate in only one possible ways as shown in Figure 2c, this results in only one termination rule (Rule 15). For each intermediate move shown in Figure 2b, there are two possible next moves. This give rise to 12 intermediate rules.

A complete sequence of seven necktie knot rules to create the cross-knot is shown in Figure 4. In order to generate a tie knot sequence the necktie graph grammar approach starts with a NULL seed (equivalent to no move or blank seed) in the left hand side (LHS) of step one (Figure 4). At this stage, it is recognized that only the two initiation rules are applicable (Rule 13, and 14). Out of the applicable rules, rule 13 is chosen and applied by a designer (or an automated computational process). This results in an Li node in the right hand side (RHS) of step one. After this stage the process of recognize, choose and apply is invoked in an iterative manner. The cross-knot can be derived by application of the seven rules: 13, 8, 9, 11, 10, 7, 15. The movement of the active end follows the sequence {Li-
Ro-Ci-Ro-Li-Co-T).

The usage of a design grammar helps to generate a wide range of solutions by altering the way the rules are applied. The grammar affords a representation of the design space as a tree of solutions built from an initial specification. Each transition in the tree corresponds to an application of a rule, thus incrementally building a final solution which is represented as one of the leaves of the tree as was shown in Figure 1.

The method described here is implemented in GraphSynth [24]. GraphSynth is a publicly available approach to create, implement, and interpret graph grammars. The rules shown in Figures 2 and 3 were easily created in GraphSynth; Figure 5 shows the interface of GraphSynth with two of the rules visible. On the left side of Figure 5, the grammar rule set is stored and managed. While it is difficult to read, it is shown to elucidate how the fifteen rules are stored; the bottom of the window includes various properties that govern the behavior of how rules are invoked. The rules shown to the right of the figure are the first and last rules in the set. They are created graphically and can be tested immediately upon creating.

The combined effect of the rules generates a design space that requires a search technique to find an optimal solution. This search process gives the designer the potential to explore a large number of alternative designs, which include many alternatives that might have been overlooked without the aid of a grammar, thus paving the way for possible innovative designs. Our approach to searching this space of solutions is described next.

4 A STOCHASTIC GRAPH GRAMMAR ALGORITHM FOR INTERACTIVE SEARCH

As in the search tree shown in Figure 1, the necktie grammar starts from a simple seed and extends to 35,498 candidate solutions at the fifteenth level of the tree. While it is not always possible, it is desirable to have no rules recognized on states in the tree only when that state represents a completed valid design. Fortunately, this is accomplished in the necktie grammar, but there are “leaves” or valid designs exist at every level of the tree after the fourth level.

In most design problems and implementations, the tree is never explicitly represented since its size is prohibitively large and difficult to visualize, and may easily contain more states than the computer can store. While 35 thousand candidates is a manageable size, we refrain from creating the full set in order to test the search method. As a rule of thumb, more than $10^{10}$ states may be viewed as impossible to enumerate, as this would require 9.3 TB of random access memory (RAM) if each state occupies one kilobyte. However, the number of rules that lead to these solutions is often a small and manageable size. This is because the grammar rules represent heuristics or constraints provided by experts in the particular design domain (in rare cases, these have been generated automatically [25, 5]). Furthermore, the grammar rules make definitive changes to a particular concept and their use is often clearly discernible within the final candidate solution. As a result, it makes sense to gather statistics on the rules used to navigate the search tree.

In the necktie grammar rules described above, rules define the basic operations that create the knot—defining whether the end of the tie is brought over from side to side, passed underneath near the neck, etc. These operations clearly dictate the shape and size of the knot.

Similar to many conceptual design problems, the results of the necktie grammar are difficult to computationally analyze. Thus, the user is queried to build up information about what is considered a good design and what is considered a poor design. From the queries, the process builds knowledge about the combination of rules used (Section 4.1). This knowledge is then used by the computational process to target better designs (as described in Section 4.2).

4.1 Gathering and Storing User Input

Since the user is required to evaluate concepts in the search process, a series of dialogs such as the one shown in Figure 6 are presented. The left-side screenshot shows a co-occurrence dominance matrix [13] corresponding to four generated candidates labeled C-1, C-2, C-3, and C-4. In the following experiments, we investigate matrices of 2, 3, 4, and 5 candidates. The values placed in the lower triangle of the matrix are based on whether the user believes the candidate of a column is better than (+1), worse than (-1) or equal to (0) a candidate presented on the row. Opposite values are copied into the upper triangle. When the user clicks “Calculate...” the columns are summed to create a score for each candidate corresponding to the candidate’s columns. In the example shown in the figure, the four candidates will receive scores of 0, +3, -1 and -2 respectively. This is a zero-sum result, where some candidates receive a negative score and others, a positive. Clearly, we seek a design that maximizes this score.
Figure 6: The user feedback is provided via a pair-wise comparison of n designs (four in this case).

These scores are not stored within the candidate, but rather are reflected onto the rules that create the candidates. On the right side of Figure 6, a text output indicates how many rules contribute to each candidate. In the necktie example application as well as many generative grammars for engineering systems, candidates exist with varying complexity – at different levels of the search tree.

A database is created where each entry contains the rule number, the fitness score as reflected by the dialogs, and the rules called previously in the candidate. For example, C-4 from the figure was created by calling rules: 2, 4, 7, 15. Through the user dialog, C-4 was assigned a value of -2. Each of the four rules that lead to C-4 leads to an entry in the database of rule knowledge. This is shown for the example in Table 1. Through successive dialogs, a large database of entries is created where each rule has many entries. The final column of Table 1 shows the previous rules that have been invoked. This is later used to provide context to how the order of rules may affect the quality of a candidate solution.

Table 1: Example entries added to database from the fourth candidate returned in Figure 6.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Rule #</th>
<th>Fitness</th>
<th>Prev. Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>-2</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>-2</td>
<td>{2}</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>-2</td>
<td>{2, 4, 7}</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td>-2</td>
<td>{2, 4, 7}</td>
</tr>
</tbody>
</table>

4.2 Guiding Search by Reading Stored Rule Knowledge

At any given state in the tree, a set of options is determined through a recognition operation that checks all rules with the given state in the tree (see Figure 1). An option is essentially comprised of a compatible rule, and possibly its operand or targeted section of the host (depending on the details of the problem, a rule may be valid at multiple locations within the host; in the simplified necktie grammar there is only ever one location per rule). A decision-making process is needed to determine which of the available options to apply. In lieu of having a human make this decision, our process will chose one option based on the following stochastic process.

Figure 7 shows an arbitrary state in the tree with three available options. Based on the database of entries described above, a score can be assigned to the fitness of a rule based on entries with the same rule number; and the popularity, or number of entries for a given rule. Note that two rules, 4 and 8, in the example of Figure 7 have no fitness assigned. They also have a popularity of zero. Clearly, no information on the performance of these rules can be determined until they have been invoked and presented to the user.

Based upon this information, which of the three options in Figure 7 would be best? A higher fitness means that the rule has performed well in the past. Calling rule 1, would best exploit what is currently understood about the rules since the fitness of that rule is higher than that of the other rules. On the other hand, calling rule 4 would lead to the exploration of a new area of the search tree. Real design mimics this decision making process. One can choose the option that leads to a tried-and-true solution, or attempt something new. This essentially becomes a multi-attribute decision amongst the available options: we want options with high fitness, but low popularity.

Prior to merging these two attributes, we normalize each term between 0 and 1.

\[
\begin{align*}
u(\text{fit}) &= (\text{fit} - \text{fit}_{\text{min}}) / (\text{fit}_{\text{max}} - \text{fit}_{\text{min}}) \\
u(\text{pop}) &= (\text{pop} - \text{pop}_{\text{min}}) / (\text{pop}_{\text{max}} - \text{pop}_{\text{min}})
\end{align*}
\]  

(1) (2)

In these equations, the “min” and “max” values correspond to the minima and maxima for the entire columns shown in the example table from Figure 7. Note that the popularity equation is formulated so that a value of one corresponds to the least popular rule. Calculating these preference functions in this way, leads to two normalized attributes with a maximum (or best) preference at 1 and the worst at 0. For the three rules in the example of Figure 7, we now have the values shown in Table 7.
One unavoidable issue in the calculation of user preferences is the lack of a fitness value for new rules. A lack of a value for fitness for rule 4 diminishes its likelihood of being chosen. Assigning a value of zero would be equivalent to saying that it performs poorly, yet no knowledge is provided as to how well it will perform. The approach taken here is to award it a 0.5 or average fitness.

Next, these two preference terms are brought together in a linear weighted sum:

\[ U = B^* u(\text{fit}) + (1 - B)^* u(\text{pop}) + u_{\text{min}}. \]  

In this equation, the calculated total preference, \( U \), for each option is a function of the aforementioned utilities and the addition of a weighting factor, \( B \). \( B \) is a tribute to G. E. P. Box, who in 1956 established the exploration versus exploitation dichotomy [26]. When \( B \) is 1 or close to 1, the process will prefer solutions with high fitness, thus exploiting what is currently known about the rules. When \( B \) is 0 or close to 0, the process favors exploring new areas of the search tree by placing more emphasis on rules that have not been called often. Finally, a small constant, \( u_{\text{min}} \), is added to all options to ensure that no option receives a preference of zero – the reason for this will be described shortly. The following table shows how the example options compare for five different values of \( B \).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Fitness, fit</th>
<th>Popularity, pop</th>
<th>( u(\text{fit}) )</th>
<th>( u(\text{pop}) )</th>
<th>( U ) as a function of ( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.2</td>
<td>5</td>
<td>0.78</td>
<td>0.00</td>
<td>( B = 0 )</td>
</tr>
<tr>
<td>3</td>
<td>0.9</td>
<td>2</td>
<td>0.69</td>
<td>0.60</td>
<td>( B = 0.25 )</td>
</tr>
<tr>
<td>4</td>
<td>N/A</td>
<td>0</td>
<td>0.50</td>
<td>1.00</td>
<td>( B = 0.5 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( B = 0.75 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( B = 1.0 )</td>
</tr>
</tbody>
</table>

This parameter, \( B \), becomes an important adjustment or "knob" for our search process. In the experiments below, we investigate what values of \( B \) lead to the best results.

Now that a single preference value is determined for each of the options, we can carefully guide the search towards candidates that best meet the user’s preference. Given the values of \( B \) shown in Table 3, we see in bold what option results in the highest preference. At this point, we introduce the stochastic quality of the new search method. Similar to the selection mechanism employed in genetic algorithms [27], our method assigns a probability to each option proportional to the aggregated preference, \( U \), of each option. As an example, when \( B \) is 0.5 as shown in Table 3, rule 1 receives a 22% probability of being chosen (\( \text{Prob}^\% = 0.396/(0.396 + 0.649 + 0.755) \)); rule 3 receives 36%; and rule 4, 42%. The stochastic decision slightly prefers choosing rule 4, but this will happen in fewer than half of the trials. As opposed to this slightly stochastic approach, one may adopt an approach to always choose the option that leads to the highest preference. However, this deterministic approach does not allow for any randomness to occur, and the process is not robust to mistakes in the user feedback or in the order solutions are presented. What is needed is a new “knob” to control the amount of randomness in the process.

Our approach builds on the p-norm matrix operation [28]. In a variety of situations, the magnitude of a matrix or vector is important – magnitude in this case meaning the distance from the zero matrix of identical dimensions. This is solved by the concept of a “norm”. The easiest approach is to simply add all the elements of the matrix or vector together. Additionally, one could determine the Euclidian norm by taking the square root of the sum of the squares of each element (also known as the magnitude in geometry). In this case, the variable, \( p \), is 2 in the generalization often referred to as the p-norm. With increasing values of \( p \), the resulting norm value is increasingly influenced by the larger elements. It can be shown that the infinity norm (infinity norm) is simply the largest element in the matrix. As a result, as \( p \) changes from 1 to infinity, the norm for a particular vector changes from the sum of all the elements down to the magnitude of the largest element.

The value of \( p \) is then applied to each of the resulting utilities shown above. If \( p = 2 \), the probability of rule 1 is calculated as:

\[
\text{Prob}(\text{rule } 1) = \frac{0.396^2}{0.396^2 + 0.649^2 + 0.755^2} \quad (4.1)
\]
\[
\text{Prob}(\text{rule } 2) = 14\% \quad (4.2)
\]

While the probability of rule 1 is effectively reduced from 22% to 14%, the probability for rule 4 increases from 42% to 50%. When \( p \) is set to 20, rule 4’s probability of being chosen increases to 95%. At \( p = 75 \), rule 4 has a 99.999% chance of being chosen. Obviously, as \( p \) increases, rule 4 becomes the definite option chosen in this example. Incidentally, \( p \) can also be reduced from 1 to values approaching zero. Since any
number raised to the zeroth power is 1, a value of $p = 0$ corresponds to a completely random choice.

The top of Figure 8 shows how changing $p$ effectively changes the process from random, through stochastic, to deterministic. Unfortunately, the sensitivity is not consistent, as the $r$ appears to be as much information between zero and 1 as there is between 1 and infinity. Additionally, numerical problems occur when $p$ is too large. Given that double-precision floating-point variables in a computer can model values between $10^{-300}$ and $10^{300}$, it is not practical to increase $p$ beyond 300 or below 0.003. As a result of these issues, we define $Q$ as the true input parameter (“knob”) in the process. In correlating $Q$ to $p$, we assume that the midpoint of the number line shown in Figure 8 ($Q = 0.5$) should correspond to the proportional selection approach ($p = 1$). An exponential function is fit through the three points to arrive at an expression for $p$ as a function of $Q$.

$$p = 0.003 + 299.997(Q^{0.23314})$$ (5)

In summary, at each stage in the construction of a new candidate, information is gathered on the recognized options. In general, the option is comprised of a grammar rule and its operand (where in the host the rule will apply). In this work, we are focused only on the rules. The computational choice relies on constructing a preference for each option, which depends on parameter $B$, and probability based on that preference, which depends on parameter $Q$.

In addition to looking at average fitness and popularity of the rule, we are also considering a rule’s context. In addition to the columns shown in Table 2, the $u(\text{fit})$ also depends on how well a particular rule performed at a specific level of the tree, and how well it performs in relation to the rules already invoked on the host. Similarly, the preference, $u(\text{pop})$ depends not only on the total number of calls to a particular rule, but how often it was called at the current level, and how often it has been called after the current list of rules already invoked. These additional “context” factors are possible since the data stored in Table 1 includes information on past rules. Currently, the additional terms are simply added together. In the future, we may explore other forms for these preference functions.

5 RESULTS

In this section, we illustrate the implementation of our interactive search strategy by using the necktie grammar. To validate the effectiveness of the approach, two sets of design experiments are run.

In running these experiments, we have designated the cross-knot (shown in Figure 4) as an arbitrarily selected “best” design from a population of 35,498 candidate solutions generated between levels one and 15 of the search tree. This gives us the ability to set a benchmark and evaluate candidate designs by comparing them to the selected benchmark design. This is accomplished by defining a distance metric, $f(d)$, which compares the recipe of the cross-knot design (Li-Ro-Ci-Ro-Li-Co-T) to those automatically posed by the search process. Specifically, a comparison is made between each element of the cross-knot and the candidate recipes. If there is a mismatch, indicating a difference from the benchmark design, $f(d)$ is incremented by 1. This is illustrated in Table 4 for six candidate designs.

There is only one design in the population with an $f(d)$ value of zero (the cross-knot design itself), for all other

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Distance Metric</th>
<th>PDF %</th>
<th>CDF %</th>
<th>Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.0000%</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.00%</td>
<td>0.01%</td>
<td>99.9972%</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0.02%</td>
<td>0.02%</td>
<td>99.9803%</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>0.03%</td>
<td>0.05%</td>
<td>99.9549%</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>0.06%</td>
<td>0.11%</td>
<td>99.8930%</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
<td>0.11%</td>
<td>0.22%</td>
<td>99.7803%</td>
</tr>
<tr>
<td>83</td>
<td>6</td>
<td>0.23%</td>
<td>0.46%</td>
<td>99.5465%</td>
</tr>
<tr>
<td>159</td>
<td>7</td>
<td>0.45%</td>
<td>0.90%</td>
<td>99.0985%</td>
</tr>
<tr>
<td>325</td>
<td>8</td>
<td>0.92%</td>
<td>1.82%</td>
<td>98.1830%</td>
</tr>
<tr>
<td>645</td>
<td>9</td>
<td>1.82%</td>
<td>3.64%</td>
<td>96.3660%</td>
</tr>
<tr>
<td>1124</td>
<td>10</td>
<td>3.17%</td>
<td>6.80%</td>
<td>93.1996%</td>
</tr>
<tr>
<td>2073</td>
<td>11</td>
<td>5.84%</td>
<td>12.64%</td>
<td>87.3599%</td>
</tr>
<tr>
<td>3465</td>
<td>12</td>
<td>9.76%</td>
<td>22.40%</td>
<td>77.5987%</td>
</tr>
<tr>
<td>4691</td>
<td>13</td>
<td>13.73%</td>
<td>35.62%</td>
<td>64.3839%</td>
</tr>
<tr>
<td>4260</td>
<td>14</td>
<td>12.00%</td>
<td>47.62%</td>
<td>52.3832%</td>
</tr>
<tr>
<td>18594</td>
<td>15</td>
<td>52.38%</td>
<td>100.00%</td>
<td>0.0028%</td>
</tr>
</tbody>
</table>

| Total     | 35498           |       |       |            |

Figure 9: The frequency of distance metric for the 35,498 necktie designs and the probability and cumulative density functions.
candidates: \( 15 \geq f(d) > 0 \). Since the necktie design problem allows the search tree to be explicitly represented thanks to its manageable size, we have computed \( f(d) \) for all of the 35,498 candidates. This is summarized in Figure 9 where the probability and cumulative density functions for the distance metric is also plotted.

Obviously, neither the population of candidates nor the “best” solution is known prior to the conceptual phase of design during which the search algorithm would normally be used. We emphasize that the designation of a benchmark design and the formulation of a distance metric is not required for employing the search strategy presented here, however, it is useful for illustrating the convergence characteristics of the algorithm and the consistency of the results obtained as is shown next.

In the first set of experiments, we explore the effects of the two aforementioned “knobs” (i.e. parameters \( B \) and \( Q \)) and the level of designer interaction on the search results. To accomplish this, experiments are run using the parameter values shown in Table 5. Accordingly, \( B \) is assigned values of \{0, 0.25, 0.5, 0.75, or 1\}. \( Q \), on the other hand, is assigned a total of ten values varying between 0.5 - 1.0. The designer interaction parameter consists of a pair of two values (# of candidates presented to the designer in a dialog window, # of series of dialogs presented in a design session). For this parameter, four values are explored \{(2, 30); (3, 10); (4, 5); (5, 3)\}. Each pair of values represents an equal number of 30 pair-wise user evaluations through which user feedback is provided.

Moreover, for each combination of parameter values (for example \( B=0.25 \), \( Q=0.8 \), and \( \text{designer interaction}=(2, 30) \)), the experiment is repeated 20 times to account for the stochastic nature of the algorithm and to obtain statistically meaningful results. In total, this corresponds to 4000 experimental runs (5*10*4*20 = 4000).

In the first set of experiments, the user evaluation of the candidates is also automated by taking advantage of the previously defined distance metric, \( f(d) \). When conducting a pair-wise comparison of \( n \) designs, the distance metric is automatically calculated and better than (+1), worse than (-1) or equal to (0) values are automatically assigned to the candidate designs presented via the dialog windows (Figure 6). The results of this experiment are shown in Figures 10 and 11.

Table 5: The parameters used for the first set of experiments

<table>
<thead>
<tr>
<th>B</th>
<th>Q</th>
<th>Candidates Presented</th>
<th>Num of Dialogs</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.5</td>
<td>2</td>
<td>30</td>
</tr>
<tr>
<td>0.25</td>
<td>0.6</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0.75</td>
<td>0.8</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0.9</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

Figure 10. Results from the automated experiments. Average distance metric, \( f(d) \), and its percentile is shown for computed best designs for varying values of \( B \) (a), \( Q \) (b), and \( \text{designer interaction} \) dialog (c).
Figure 10 illustrates the results as a function of $B$, $Q$, and the designer interaction dialog. In these plots, the average distance metric $f(d)$ is shown for the best designs generated for the different values of $B$ (each point averaged over 800 experiments), $Q$ (each point averaged over 400 experiments), and the designer interaction dialog (each point averaged over 1000 experiments) listed in Table 5. In addition, for each data point, the percentile of the reported average distance metric is shown based on the cumulative density function (CDF) values of Figure 9. The results show the consistency of the interactive search approach. By utilizing input from only 30 pair-wise comparisons, the algorithm successfully finds a best design within 99.8 percentile for all experimental test cases (with the exception of $B=0$ (99.61 percentile), and $Q=0.5$ (99.35 percentile)). For the necktie grammar, the overall best results are obtained when $B=0.75$, $Q=0.8$, and designer interaction $=(3, 10)$ with an average distance metric value of 2.8625 (averaged over 20 experiments), and a percentile value of 99.958.

Figure 11 shows the average distance metric, $f(d)$, computed for best designs generated at the end of each user feedback dialog. These results were obtained for the case where 3 design candidates were shown at each dialog window for a series of 10 dialogs. (i.e. designer interaction $=(3, 10)$, and each data point is averaged over 1000 experiments.) The results show the convergence of the interactive search approach. As more user feedback is provided, the algorithm develops a more accurate representation of the user preferences. This is reflected on the average distance metric scores computed at the end of each dialog window, which monotonically decreases as the number of dialogs increase.

In the second set of experiments, we took the parameter values yielding the best results for the automated experiments (i.e. $B=0.75$, $Q=0.8$, and designer interaction $=(3, 10)$), and ran 21 user experiments in which the three co-authors of the paper acted as the designer and provided user feedback via dialog windows. In providing this feedback, the designers considered the similarity of the presented candidates to the cross-knot design. We then recorded suggested best designs after 10 dialog windows and computed the distance metric, $f(d)$, for each of the 21 recorded best designs. Finally, we averaged the $f(d)$ values over 21 user experiments and compared it to the reported best overall case from the first set of experiments. This comparison is shown in Table 6. The user experiments yield similar results to the automated experiments with an average distance metric value of 2.904, and a percentile value of 99.957.

Overall, the two sets of experiments show the effectiveness of the interactive search algorithm in learning user preferences through very limited user feedback, and how the search approach can be used to consistently find solutions in large design spaces that best mimic the user preferences.

### 6 DISCUSSION AND CONCLUSIONS

This paper presents an approach to merge the judgment abilities of human designers with a computational search strategy. Many attempts to automate conceptual design often results in overly simplified solutions as a result of limited representational capabilities or simplified evaluation methods. The novelty of the approach presented here is the fact that it leverages the representational power of generative grammars with the evaluation power of astute human designers. When presented with a question such as picking out an aesthetically pleasing tie, choosing a musical score, or taste in coffee - it is difficult to construct a mathematical equation for accurately capturing why and how one option is preferred over another. In such problems, it is a necessity to use an interactive evaluation strategy in which a human user does the evaluation manually rather than using a mathematical equation. The results shown in this paper are a promising step in accomplishing this goal and the method should be explored further as a way to merge human intelligence and creativity with thorough and efficient computational search.

The difficulty in interactive methods such as this is the tradeoff in user feedback. Given that user fatigue limits the number of interactions, we must strive to make the most of
what data the user has provided. This introduces the following research questions:

- What types of questions should be asked to the user?
- How many times can we safely ask the user such questions before the quality of user judgment decreases?
- How should such data be used to guide the search process?

The simplest answer to the first question is perhaps the approach taken here. Presented with candidate solutions, simply input whether one candidate is better, worse or equal to another candidate. Such a query is easy for the user, but we miss opportunities to gain information on how much better one candidate is than another or about what aspect of a particular candidate is preferred. However, given the simplicity of our queries, we find that we are able to ask the user to make many comparisons (30 in this case) without significant user fatigue.

In the human experiments performed, each of the 21 experiments took approximately five minutes. It is possible for larger spaces or more complex problems that this time could be increased.

The number of solutions in the dialog and the number of dialogs is worthy of further discussion even though there is not true statistical significance in the results of Figure 10c. When presented with a larger dialog such as the one shown in Figure 6, we find that the user wants to put the four designs in order through the determination of the six pair-wise comparisons. In fact, a user interface where the user would order a small list such as this would likely be quicker for the user and lead to more consistent results (avoid circular rankings). Additionally, the experiments with dialogs of four or five candidates were completed in a fraction of the time than those involving only 2 or 3 candidates. This is partly due to the fact that more comparisons are done between designs that the user has already learned. In the 2 by 30 interaction, the user must learn two designs for every one point of feedback; whereas in the 5 by 3 interaction, there are two points of feedback for every design learned (i.e. 10 pair-wise comparisons between 5 designs). However, with such a small number of interactions (3 or 5), there is not enough time for the computational process to build upon user feedback. While 30 comparisons provided a control in the experiment, it is likely that if user time were held constant the dialogs with 5 candidates would be best.

The answer to the third question above is complicated by the exploitation versus exploration issue. Given the automated experiment, it seems that B should be set between 0.5 and 0.75 (see Figure 10a). This means that more emphasis is placed on what the user has ranked highly thus far (exploitation) as opposed to exploring the space more. It is important to see that in Figure 10a, the results that occur when B is 1 are significantly worse. This means that some amount of exploration is required to achieve good results.

A more challenging concept to understand is the affect of Q on the process. In a way, Q seems similar to the explore versus exploit issue, but it is important to understand that its impact is merely on the degree of stochastic learning. One can think of Q as how much the process is heeding the feedback provided by the user. When Q is equal to or less than 0.5 (Figure 10b), many decisions are made in the search tree that do not conform to information in the stored database. The search prefers inferior choices rather than those that lead to new or well-received candidates. When Q is one, the search always calls the option best scored from the stored database. It appears in Figure 10b that values above 0.9 are too deterministic or too trusting in the stored rule knowledge. Some degree of randomness does seem an important issue. The fluctuation in this range of the plot is also difficult to understand. While each point is an average of 400 separate runs, the standard deviation for these (not shown) is high. It seems that whatever random designs are shown in the first dialog unintentionally but greatly impacts the final result in these cases. This is perhaps the true reason that randomness is required in the process – to combat the randomness in the starting point of the search. Perhaps early on, it is wise to ignore the user feedback in order to build up a more solid bank of data. Later in the process, if Q were to be increased, earlier feedback could still be leveraged to inform decisions.

This experiment leads to further future studies for our interactive search strategy. It seems likely that performance can be increased if the B and Q knobs are scheduled to change through the process in a manner similar to temperature in simulated annealing [29]. Additionally, the user dialog could change size – perhaps starting with a large number of candidates and eventually ending in a single pair-wise comparison.

While our method is particular to generative grammar systems comprised of a set of rules, it is important that future work compare the method’s effectiveness in other problem domains. Additionally, we have ignored the “operand issue” in generative grammars. This means that we gathered statistics on the rules but not on where they applied in the construction from seed to completed candidate. We did address when the rule was applied (what level of the tree) and in reference to which other rules, and given that each rule has strict recognition conditions; it seems that the where issue is successfully curtailed. However, in larger systems this may be necessary to consider. It is definitely a complicated issue to solve on a generic level given that the generative grammar could be constructing strings, graphs, shapes, or arrays, and thus the operand would be some fragment of these fundamental mathematical constructs.

There are also some open issues that we intend to investigate in the future. First, we plan to compare the proposed approach to other interactive search techniques (most notably interactive genetic algorithms) in order to verify our results and benchmark our approach against such methods [18]. Finally, the work will need to address other more sophisticated generative grammars especially those being developed in engineering design. The benefits of the method may be useful in many applications where user input is paramount to solving the problem. One interesting extension of this work lies in the users’ lack of knowledge and interaction with the grammar. The user only needs to evaluate completed solutions. As a result, the user need not be an expert in the design of the artifact, but may in fact be a customer. Given a generative grammar that
produces only valid and feasible configurations, the work may be useful in automating the dialog necessary for mass customization [30].

REFERENCES


