A Stochastic Tree-Search Algorithm for Generative Grammars[1]

Matthew I. Campbell[2]
Associate Professor
Automated Design Laboratory,
Department of Mechanical Engineering,
University of Texas at Austin,
Austin, TX 78712-0292
e-mail: mc1@mail.utexas.edu

Rahul Rai
Assistant Professor
Advanced Design Optimization
(ADOPT) Laboratory,
Department of Mechanical Engineering,
California State University,
Fresno, California 93740
e-mail: rarai@csufresno.edu

Tolga Kurtoglu
Research Scientist
Palo Alto Research Center,
3333 Coyote Hill Road,
Palo Alto, CA 94304
e-mail: kurtoglu@parc.com

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. A formal definition of a rule-based interactive tree-search is presented in this paper. The user interaction is kept to 30 pair-wise comparisons of candidates. From the data gathered from the comparisons, a stochastic decision-making process infers what candidate solutions best match the known optimal. 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 computational complexity of the proposed algorithm is also studied. The implications of this method for conceptual design are expounded on in the conclusions. [DOI: 10.1115/1.4007153]

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 [1]. 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 graph in which the desired inputs and outputs are cast as arcs and nodes of 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. A generative grammar is 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 determine when it is applicable, and a right-hand side (RHS) that contains the transformation instructions. As shown in Fig. 1, the rules build upon a seed graph to create a tree of possible solutions. For each grammar rule, whose 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. This interpretation of the design process makes graph grammars suitable for computationally modeling the open-ended nature of conceptual design, where designers explore various ideas, decisions, and modifications to previous designs to arrive at feasible solutions.

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 to identify optimal solutions.

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 evaluate candidate solutions by presenting a sample of solutions to a knowledge user or expert in hopes to glean what differentiates good concepts from bad concepts and to use that information to find better solutions in the search tree. Given the context of design, the expert user in this method is likely a designer, and thus we use the terms user, designer, and expert interchangeably in this paper. In this manner, the computational process need not be concerned about whether the user is evaluating cost, size, efficiency, aesthetics, or some combination of performance parameters. It merely takes the user

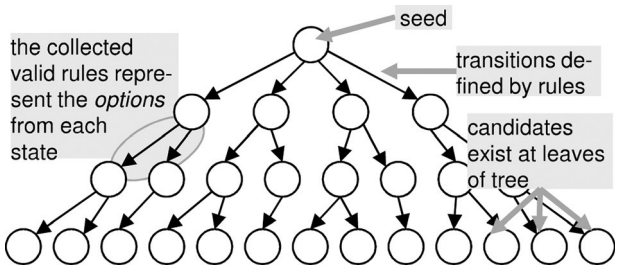

Fig. 1 An example of a search tree that results from invoking a generative grammar on a seed

[1]A shorter version of this manuscript has been published in ASME 2009 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (DETC09/DTM-86804).

[2]Corresponding author.

Contributed by the Computers and Information Division of ASME for publication in the JOURNAL OF COMPUTING AND INFORMATION SCIENCE IN ENGINEERING. Manuscript received July 11, 2012; final manuscript received July 12, 2012; published online xx xx, xxxx. Assoc. Editor: Bahram Ravani.

input and seeks solutions to maximize the user's ratings. This is accomplished by reflecting the user input onto the generative grammar rules through various user feedback sessions that happen at intervals in the computational search process. Rule-centered data accrues 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 the gathered data. This paper presents a formal definition of this problem, describes the approach in detail and presents the results of an experiment that validates the effectiveness of the approach.

In order to complete the experiments, an example generative grammar must be chosen. While the authors have established various complex engineering design generative grammars [2–5], a simple 15-rule grammar is used in this paper that defines ways to tie a necktie. This 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 Sec. 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 [5]. The difference in the method presented here 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 occupy a few kilobytes of computational memory, it may be possible to store as many as a million candidate solutions. However, a million possible solutions are 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 match the user's goals.

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 [6]. Recently, engineering design researchers have discovered that graph grammars provide a flexible approach for the creation of complex engineering systems.

Agarwal and Cagan's coffee maker grammar [7] was one of the first examples of using grammars for product design. Their grammar described a language that generates a large class of coffee makers. Shea et al. [8] presented a parametric shape grammar for the design of truss structures that uses recursive annealing techniques for topology optimization. Other engineering applications include Brown and Cagan [9] who presented a lathe grammar, Schmidt and Cagan's grammar for machine design [10], Starling and Shea's grammars for mechanical clocks [11], and gear trains [12]. One of the implementations of grammars to function-based design is the work of Sridharan and Campbell [3]. This research showed how a set of 69 grammar rules is developed to guide the design process from an initial functional goal to a detailed function structure. Kurtoglu and Campbell [2] 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 subfunctions of the function structure.

Design evaluation and selection, on the other hand, are another important part of the design process and has received great attention in the design literature. A common concept evaluation method is the Pugh concept selection charts [13]. 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 [14] are similar methods and employ qualitative and quantitative evaluation scales. The analytic hierarchy process (AHP) [15] is another multicriteria 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 [16]. The general method for these techniques is to assign a value for 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 performance score.

Finally, interactive methods for design evaluation include the design preference modeler (DPM) [5]—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 Orsborn et al. [17], 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 (IGAs) [18] 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 [19]. However, problems in which the natural representation of the state space is a tree are usually not amenable to evolutionary search [20]. The algorithm proposed in this paper has been designed for searching trees. Depending on the design problem, it may be possible to encode the representation of the search space so that stochastic processes like a genetic algorithm may be used. However, this is difficult, if not impossible, when a complex set of grammar rules is defined for the problem. One possible encoding scheme is to represent the path through the search tree with a genotype encoding (e.g., an array of integers or bits); however, difficulties arise if the tree lacks a constant branching factor or fixed depth. In a generic case, one must accommodate the fact that solutions exist at different levels of the tree and that each nonterminating state may have a unique number of transitions. As a result, this method has more in common with artificial intelligence (AI) reasoning methods [21–23] used in engineering design in the past than interactive genetic algorithms.

3 Necktie Knot Graph Grammar

In this paper, we use a 15-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 neckties knots like Windsor and Half Windsor came into existence. The most recent of all necktie knots, Pratt, originated in 1989 [24]. As history alludes, necktie knots were mostly discovered by chance. Recently, however, Fink and Yong [25] presented the first formal approach to represent all possible

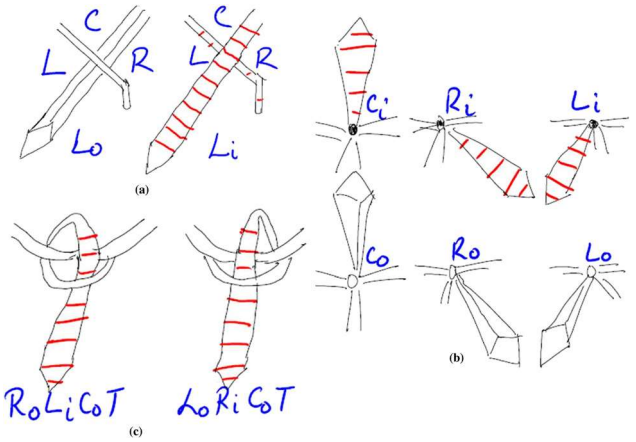

Fig. 2 Rules to tie a necktie knot (*a*) rules for beginning a tie knot, (*b*) rules for continuing a tie knot, and (*c*) rules for terminating a tie knot

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 necktie 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 (Fig. 2(*a*)). 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.

The triagonal 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 neck. Each movement in tying the knot carries the active end toward one of these three areas, and each movement in tying the knot carries the active end either away from the shirt (o) or between the shirt and the rest of the tie (i). For a valid tie knot operation, the direction must oscillate between (i) and (o) and no two consecutive move regions may be the same (Fig. 2(*b*)).

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 (Fig. 2(*c*)). 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 grammar rules were formulated. Among the 15 rules, there are two initiation rules, one termination rule and 12 intermediate move rules (Table 1). Since the 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 way as shown in Fig. 2(*c*), this results in only one termination rule (rule 15). For each intermediate move shown in Fig. 2(*b*), there are two possible next moves. This gives rise to 12 intermediate rules. Overall, the set of 15 rules provides an exhaustive grammar that enables the creation of all possible configurations for tying a necktie.

A complete sequence for one candidate necktie, the cross-knot, is shown in Fig. 3. In order to generate a tie knot sequence, the necktie grammar approach starts with a NULL seed (equivalent to no move or blank seed) in the LHS of step one (Fig. 3). At this

Table 1 Fifteen graph grammar rules to tie a necktie knot

1	Lo→LoRi	6	Co→CoRi	11	Ci→CiRo
2	Lo→LoCi	7	Li→LiCo	12	Ci→CiLo
3	Ri→RiCo	8	Li→LiRo	13 (initiation)	Null→Li
4	Ri→RiLo	9	Ro→RoCi	14 (initiation)	Null→Lo
5	Co→CoLi	10	Ro→RoLi	15 (termination)	Co→CoT

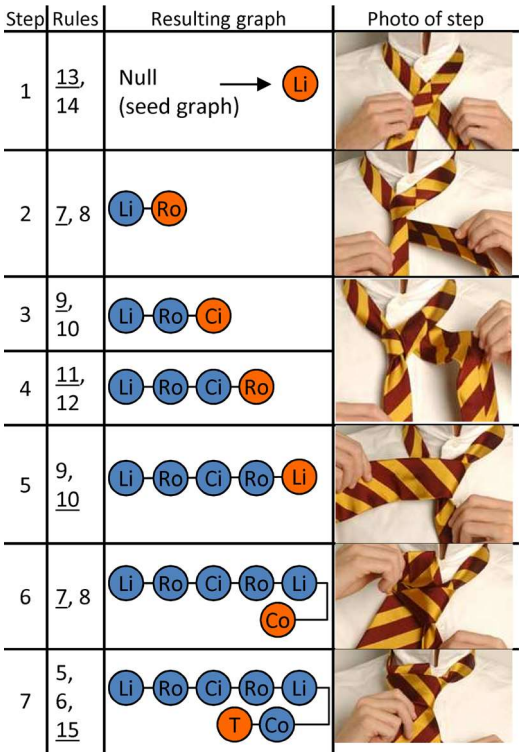

Fig. 3 Application of rule sequence {13,8,9,11,10,7,15} creates a cross-knot in seven steps from initial Null seed

stage, it is recognized that only the two initiation rules are applicable (rules 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 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 created by invoking the following seven rules: 13, 8, 9, 11, 10, 7, and 15. The movement of the active end follows the sequence {Li-Ro-Ci-Ro-Li-Co-T}.

The usage of a design grammar affords the creation of a wide range of solutions simply by choosing different rules at each point in the tree. 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 shown in Fig. 1.

The method described here is implemented in GraphSynth [26]. GraphSynth is a publicly available approach to create, implement, and interpret graph grammars. The rules shown in Fig. 3 were easily created in GraphSynth; Fig. 4 shows the partial interface of GraphSynth with two of the rules visible. On the left side of Fig. 4, the grammar rule set is stored and managed. While it is difficult to read, it is shown to elucidate how the 15 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.

4 A Stochastic Graph Grammar Algorithm for Interactive Search

As in the search tree shown in Fig. 1, the necktie grammar starts from a simple seed and extends to 35,498 candidate solutions at

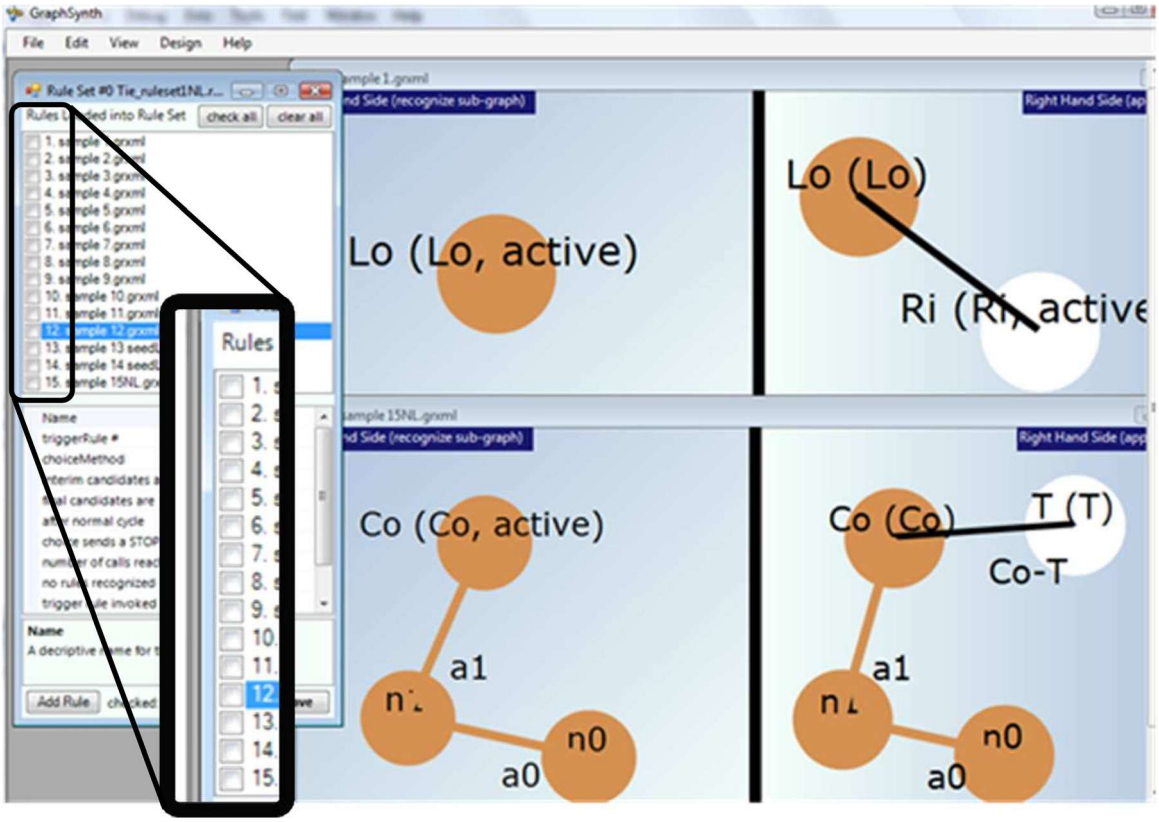

Fig. 4 A screenshot of GraphSynth depicting the list of rules (on the right) and two rules from the necktie grammar

the fifteenth level of the tree (notice that we have limited the length of a recipe to 15). 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. Unfortunately, there are "leaves" or valid designs existing at every level of the tree after the fourth level and the tree never terminates.

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 35,000 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 if each state occupies 1 kb. However, the number of rules that lead to these solutions is often small and manageable. 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 [27] and [1]). 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 (Sec. 4.1). This knowledge is then used by the computational process to target better designs. The pseudocode of the heuristic used in the interactive graph search algorithm is shown in Fig. 5. Details pertaining to each step of the pseudocode are elaborated in subsequent subsections.

4.1 Gathering and Storing User Input to Generate Knowledge Nuggets (Steps 1–4). Since the user is required to evaluate concepts in the search process, a series of dialogs such as the one shown in Fig. 6 are presented. The left-side screenshot shows a co-occurrence dominance matrix [14] corresponding to four generated candidates labeled C-1, C-2, C-3, and C-4. These candidate solutions form the feedback design set. 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. The definition of "better than" and "worse than" is completely determined by the user's perception of the resulting candidate graphs. It may be a combination of a variety of performance parameter (e.g., cost, size, weight) or it may simply be aesthetic preference as is the case in this necktie problem. The computational presentation of each necktie candidate is merely the set of steps that need to be followed. In the subsequent study, the user was prompted to follow the steps in the candidate and judge whether the resulting necktie was aesthetically pleasing. When the user clicks "Calculate…" the columns are summed to create a score for each candidate relative to the other candidates in the set. In the example shown in the figure, the four candidates will receive scores of 0, +3, −1, and −2, respectively. These scores form the cumulative score set (CSS). 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. FDS and CSS combine to form the experiment set E.

Since these scores are only relative to a small set of solutions, it is not meaningful to store the value with the candidate as if this

```
Create M random candidates
FOR N Iterations
  #1: Present M candidates on a dominance
      matrix
  #2: Retrieve user scores from table
  #3: SUM columns in table to form relative
      candidate scores
  #4: FOR EACH rule in EACH of the M
      candidates
      #4.1:     Create a knowledge nugget
         with rule, relative score and
         previous rules
  #5: Create M stochastically driven
         candidates
         (RECOGNIZE-CHOOSE-APPLY Process)
      FOR EACH
         #5.1:  RECOGNIZE rules on the seed
                graph (top of search tree),
                store as list of options.
         #5.2:  Compute Fitness of each
                recognized option
         #5.3:  Compute popularity of each
                recognized option
         #5.4:  Compute U(fit)
         #5.5:  Compute U(pop)
         #5.6:  Compute combined score, U
         #5.7:  Compute option probability,
                P(U)
         #5.8:  CHOOSE option based on P(U)
                (stochastic choice)
         #5.9:  APPLY option on host
         #5.10: RECOGNIZE rules on new host
         #5.11: IF no recognize options,
                THEN BREAK
                ELSE GO TO STEP 5.2.
      ENDFOR
ENDFOR
  #6: Create single best solution by choos-
      ing best options in RECOGNIZE-CHOOSE-
      APPLY Process (deterministic, no
      longer stochastic).
```

Fig. 5 Pseudocode for the heuristic used in the interactive graph search algorithm

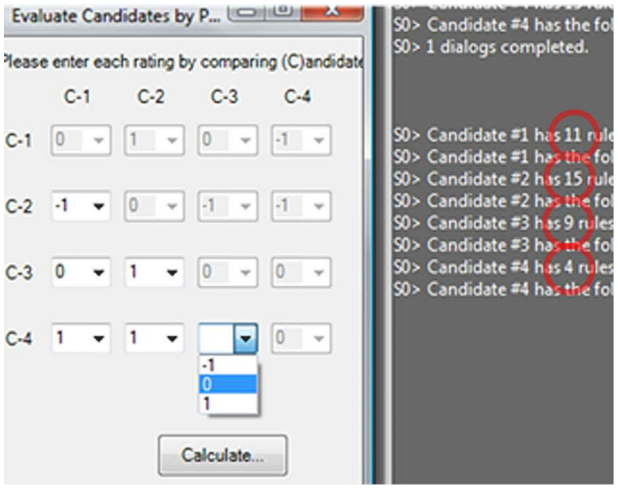

Fig. 6 The user feedback is provided via a pair-wise comparison of *n* designs (four in this case)

Table 2 Example knowledge nugget entries added to database from the fourth candidate returned in Fig. 6

Entry	Rule number	Fitness	Previous rules
1	2	−2	{}
2	4	−2	{2}
3	7	−2	{2, 4}
4	15	−2	{2, 4, 7}

score represents a quantitative assessment of the aesthetics and thus an objective function value that is sought to be optimization. Instead, these scores are reflected onto the rules that created each candidate. On the right side of Fig. 6, a text output indicates how many rules contribute to each candidate. In the necktie example 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 known as knowledge nugget 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 definea knowledge nugget entry in the database of rule knowledge. This is shown in Table 2. Through successive dialogs, a large database of knowledge nuggets is created where each rule has many entries. The final column of Table 2 shows the previous rules that have been invoked. Later on, this is used as a basis for determining how the order of rules may affect the quality of a candidate solution.

4.2 Computing Rule Fitness and Rule Popularity (Steps 5.2 and 5.3). In step 5 of the pseudocode shown in Fig. 5, a set of solutions are created through by invoking grammar rules. This is referred to as the recognize-choose-apply process. The grammar rules are first *recognized* on the current host to generate a list of options, an agent (computer or human) chooses one of the options, and the option is applied to the host. This process is modified here by including a method to evaluate options and stochastically *choose* between them. In steps 5.2 and 5.3 of the algorithm (see Fig. 5), rule fitness and rule popularity are calculated from the database of knowledge nuggets. For each rule, corresponding entries are located in the knowledge nugget database. Total rule fitness is calculated by adding the fitness of all the entries in the knowledge nugget database. Dividing the total rule fitness by the number of the entries corresponding to a rule gives average rule fitness. Rule popularity is calculated by counting the number of times a rule appears in the knowledge nugget database. At this stage of the example, rule fitness and popularity are tabulated as shown in the right side of Fig. 7.

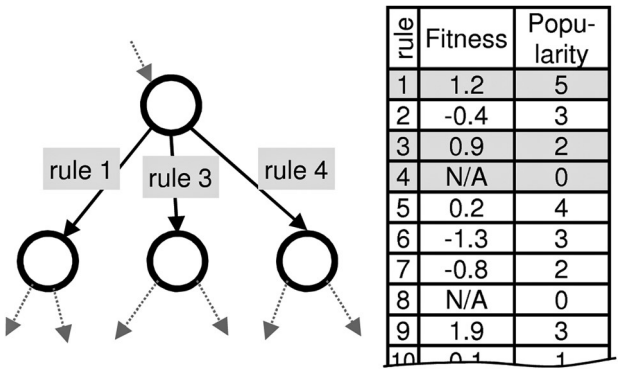

Fig. 7 An example of a search tree that results from invoking a generative grammar on a seed

4.3 Computing U (Fit) and U (Pop) (Steps 5.4 and 5.5). A set of options is determined through a recognition operation that checks all rules with the given state in the tree (see Fig. 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 knowledge nugget database, a score is assigned to the fitness of a rule; and the popularity, or number of entries for a given rule. Note that two rules, 4 and 8, in the example of Fig. 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 Fig. 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.

At this stage, we normalize each term between 0 and 1 for each rule

$$u(\text{fit}) = (\text{fit} - \text{fitmin})/(\text{fitmax} - \text{fitmin}) \quad (1)$$

$$u(\text{pop}) = (\text{pop} - \text{popmax})/(\text{popmin} - \text{popmax}) \quad (2)$$

In these equations, the "min" and "max" values correspond to the minima and maxima for the entire columns shown in the example table from Fig. 7. Note that the popularity equation is formulated so that a value of one corresponds to the least popular rule. Calculating the metrics in this way lead to two normalized attributes with a maximum (or best) value at 1 and the worst at 0. For the three rules in the example of Fig. 7, we now have the values shown in Table 3. These values are then used to calculate the combined user score, U, for a given rule.

4.4 Computing the Combined Score (Step 5.6). One unavoidable issue in the calculation of rule score 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 terms are brought together in a linear weighted sum

$$U_i = B^* u_i(\text{fit}) + (1 - B)^* u_i(\text{pop}) + u_{\min} \quad (3)$$

In this equation, the calculated total score, U_i, for each option i is a function of the aforementioned utilities and the addition of a weighting factor, B. B is a tribute to Box, who in 1956 established the exploration versus exploitation dichotomy [28]. 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, $u_{\min}$, is added to all options to ensure that no option receives a score of zero—the reason for this will be described shortly. Table 4 shows how the example options compare for five different values of B. In each column, where a B value is provided, a bolded cell value indicates the winner or highest score for the three options. Note that the choice of a winner significantly depends on B. Thus, 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.

Table 3 The calculated utilities for the valid option from Fig. 6

Rule	Fitness, fit	Popularity, pop	u(fit)	u(pop)
1	1.2	5	0.78	0.00
3	0.9	2	0.69	0.60
4	N/A	0	0.50	1.00

Table 4 The aggregated utility, *U*, for the three options based on different values of *B*

			U as a function of B				
Rule	u(fit)	u(pop)	$B = 0$	$B = 0.25$	$B = 0.5$	$B = 0.75$	$B = 1.0$
1	0.78	0.00	0.005	0.200	0.396	0.591	**0.786**
3	0.69	0.60	0.605	0.627	0.649	**0.671**	0.693
4	0.50	1.00	**1.005**	**0.880**	**0.755**	0.630	0.505

4.5 Computing Option Probabilities (Step 5.7). Now that a single score is determined for each of the options, we can guide the search toward candidates that have maximal fitness. At this point, we introduce the stochastic quality of the new search method. Similar to the selection mechanism employed in genetic algorithms [29] our method assigns a probability (Eq. (4)) to each option proportional to the aggregated score, U_i, of each option i

$$\text{Probability (option } i) = \frac{U_i^p}{\sum_{i=1}^{N} U_i^p} \quad (4)$$

As an example, when B is 0.5 and p is 1 as shown in Table 4, option 1 (rule 1) receives a 22% probability of being chosen (Prob = 0.396/[0.396 + 0.649 + 0.755]); option 2 (rule 3) receives 36%; and option 3 (rule 4) receives 42%. The stochastic decision slightly prefers choosing option 3 (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 winning option—the option with highest score. 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 [30]. 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 ∞-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.

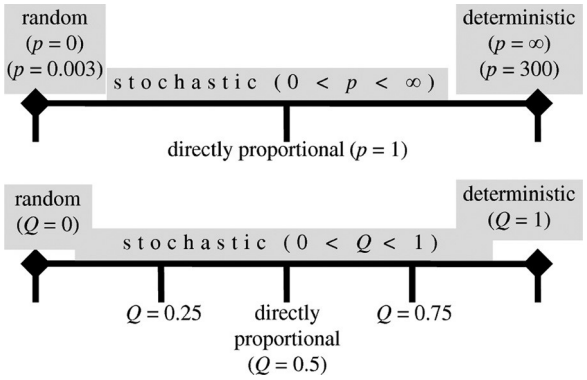

Fig. 8 The number lines show corresponding values of *p* and *Q*

The value of p is then applied to each of the resulting utilities shown above. If p is 2, the probability of option 1 (rule 1) is 14% (Prob(rule1) $= 0.396^2/[0.396^2 + 0.649^2 + 0.755^2]$).

While the probability of option 1 (rule 1) is effectively reduced from 22% to 14% by changing the value of p from 1 to 2, the probability for option 3 (rule 4) increases from 42% to 50%. When p is set to 20, rule 4's probability of being chosen increases to 95%. At $p = 75$, option 3 (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 Fig. 8 shows how changing p effectively changes the process from random, through stochastic, to deterministic. Unfortunately, the sensitivity is not consistent, as there 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 Fig. 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

$$p = 0.003 + 299.997^{*}\left(Q^{8.23314}\right) \tag{5}$$

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 score for each option, which depends on parameter B, and the probability based on that score, 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 3, the u(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 score, u(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 2 includes information on past rules. Currently, the additional terms are simply added together. In the future, we may explore other forms for these scoring and probability functions. The computational complexity of the proposed algorithm is linear with n, where n is the number of rules in the production rule set. Hence, it is a computationally efficient algorithm.

4.6 Generating Candidates and Final Output (Steps 5.9 to end). After an option is chosen, it is applied and the process returns to step 5.2 if there are more recognized rules. When no more rules are recognized, a completed solution is stored until all M candidates are created. These M candidates are then presented to the user in step 1 and the process repeats for a total of N iterations and hence N user interactions. At the end of the process, one final best candidate is found by invoking a recognize, choose, and apply process where the knowledge nuggets are referenced to choose the best option based purely on fitness and not popularity ($B = 1$). The option with the highest probability is always chosen ($Q = 1$) so that there is no randomness in this final step. While there is no guarantee that choosing options in this fashion will lead to the best candidate, this is the only vehicle available in finding a best solution as there is no objective function defined or modeled computationally; just a sense of what rules are preferred and how rules are combined based on the user's comparison of past solutions. As we will see in the next section, this detailed store of rule knowledge is enough to create a successful final candidate solution.

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 Fig. 3) 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, the Hamming distance is calculated between the candidate recipes and the cross-knot design. (The Hamming distance between two strings of data values is the number of positions for which the corresponding data values are different) This is illustrated in Table 5 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 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

Table 5 Calculated distances, *f(d)*, of six candidate designs from the cross-knot design

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Difference
Cross-knot										**Li**	**Ro**	**Ci**	**Ro**	**Li**	**Co**	**T**	**Datum**
c-6										Li	Co	Ri	Lo	Ri	Co	T	4
c-1													Lo	Ri	Co	T	5
c-7										Li	Ro	Ci	Lo	Ri	Co	T	2
c-5397	Lo	Ri	Lo	Ri	Lo	Ri	Lo	Ci	Lo	Ri	Lo	Ri	Lo	Ri	Co	T	13
c-43							Lo	Ri	Lo	Ri	Lo	Ri	Lo	Ri	Co	T	8
c-11									Lo	Ci	Lo	Ci	Ro	Li	Co	T	3

Frequency	Distance metric	PDF %	CDF %	Percentile
1	0	0.00%	0.00%	100.0000%
1	1	0.00%	0.01%	99.9972%
6	2	0.02%	0.02%	99.9803%
9	3	0.03%	0.05%	99.9549%
22	4	0.06%	0.11%	99.8930%
40	5	0.11%	0.22%	99.7803%
83	6	0.23%	0.46%	99.5465%
159	7	0.45%	0.90%	99.0985%
325	8	0.92%	1.82%	98.1830%
645	9	1.82%	3.64%	96.3660%
1124	10	3.17%	6.80%	93.1996%
2073	11	5.84%	12.64%	87.3599%
3465	12	9.76%	22.40%	77.5987%
4691	13	13.21%	35.62%	64.3839%
4260	14	12.00%	47.62%	52.3832%
18594	15	52.38%	100.00 %	0.0028%

35498 total

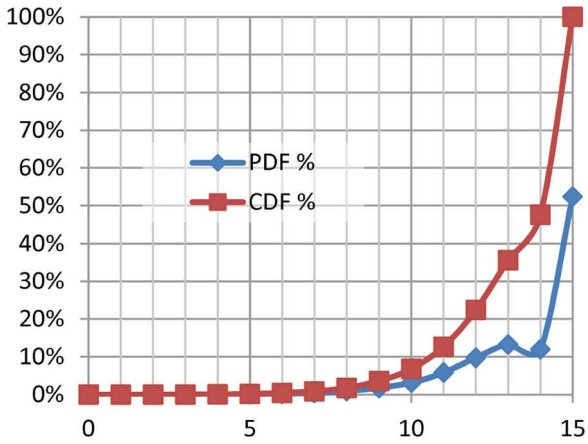

Fig. 9 The frequency of distance metric for the 35,498 necktie designs and the probability and cumulative density functions

summarized in Fig. 9 where the probability and cumulative density functions (CDFs) 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 6. 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 and 1.0. The designer interaction parameter consists of a pair of two values (# of candidates presented to the designer in a dialog window, number 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 *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). Each experimental run involves generating 20–40 candidate solutions. Thus, we have generated more than 12,000 candidates (assuming an average of 30 solutions per experiment) to derive the statistics behind our results.

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 (Fig. 6). The results of this experiment are shown in Figs. 10 and 11.

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 since there are five distinct values of B), Q (each point averaged over 400 experiments), and *the designer interaction dialog* (each point averaged over 1000 experiments) listed in Table 6. In addition, for each data point, the percentile of the reported average distance metric is shown based on the CDF values of Fig. 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 three 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's goal. 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 *user 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

Table 6 The parameters used for the first set of experiments

B (0 is explore, 1 is exploit)	0	0.25	0.5	0.75	1					
Q (0 is random, 1 is deterministic)	0.5	0.6	0.7	0.8	0.9	0.93	0.95	0.97	0.99	1
Design interaction via dialogs	2 by 30	3 by 10	4 by 5	5 by 3		(Candidates presented) by (Number of Dialogs)				

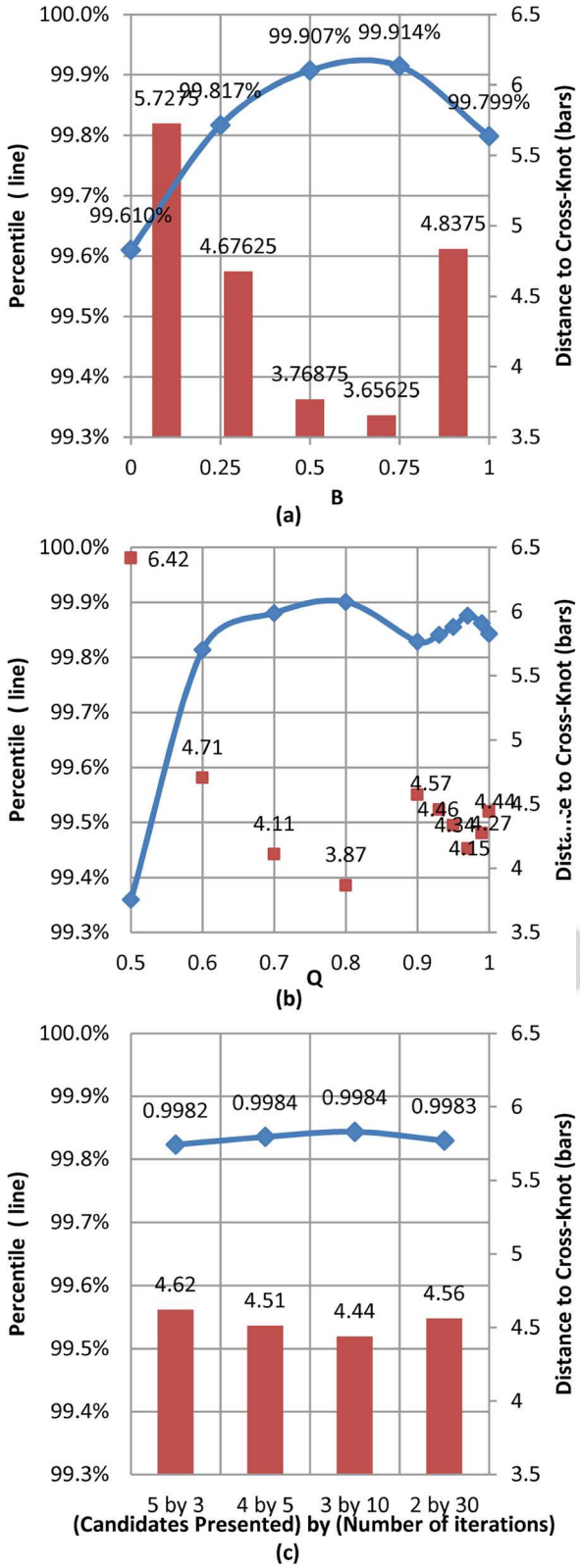

Fig. 10 Results from the automated experiments. Average distance metric, *f(d)*, and its percentile is shown for computed *best designs* for varying values of *B* (*a*), *Q* (*b*), and *designer interaction dialog* (*c*).

recorded the 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

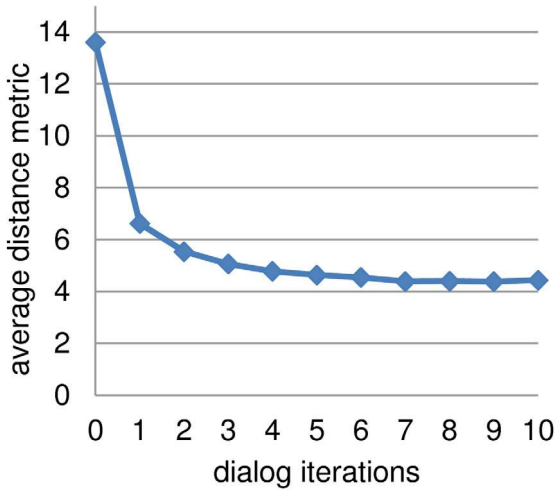

Fig. 11 Average distance metric computed for best designs generated at the end of each user feedback dialog

Table 7 Comparison of user experiments to automated experiments

	Automated experiments (20 experiments)	User experiments (21 experiments)
B	0.75	0.75
Q	0.8	0.8
(Candidates presented) by (number of dialogs)	3 by 10	3 by 10
Average distance metric, $f(d)$	2.8625	2.904
Percentile	99.96%	99.96%

from the first set of experiments. This comparison is shown in Table 7. 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.

6 Discussion of Results

The difficulty in interactive methods such as the one presented in this paper is the tradeoff in designer feedback. Given that "designer fatigue" limits the number of user interactions with the search process, one must strive to make the most of the information provided by the designer. In this section, some of the pertinent issues related to interactive methods are discussed by introducing a set of research questions. Notable observations from our experimental results are presented to shed some light on the formulated research questions.

What types of questions should be asked to the designer?

The simplest answer to this question would be a binary response: yes versus no, or like versus dislike. Then, perhaps the ternary response utilized in this study. Presented with candidate solutions, simply input whether one candidate is better, worse, or equal to another candidate. Such a query is easy for the designer, but we miss opportunities to gain information on how much better one candidate is than another. Higher order responses such as Likert scale ratings {Likert} of one to five or one to ten would afford more sophisticated feedback to the search process but would likely introduce more concerns about user error. An even more advanced mode of feedback would involve the user indicating what aspects of a solution are liked or disliked. The AI related to this type of feedback would require significant development and testing, and the resulting data gleaned for informing rules may not be much more valuable than the simpler approaches.

How many times can we safely ask the designer such questions before the quality of designer judgment decreases?

The number of solutions in a single dialog and the number of dialogs is worthy of further discussion and research. When presented with a larger dialog of 4-by-4 or 5-by-5 candidates such as the one shown in Fig. 6, we find that the designer 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 (by avoiding circular rankings). Additionally, the experiments with dialog interactions, [4 in a dialog]-by-[5 separate dialogs] and [5 in a dialog]-by-[3 separate dialogs] were completed in a fraction of the time than those involving only two or three candidates in a dialog ([3 in a dialog]-by-[10 separate dialogs] and [2 in a dialog]-by-[30 separate dialogs]). This is partly due to the fact that more comparisons are done between designs that the designer has already learned. In the [2]-by-[30] interaction, the designer 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 five designs). However, with such a small number of interactions (three or five), there is not enough time for the computational process to build upon designer feedback. While 30 comparisons provided a control in the experiment, it is likely that if designer time were held constant, the dialogs with 5-by-5 candidate comparison would be the best.

How should such data be used to guide the search process?

The answer to this third question is complicated by the exploitation versus exploration issue. Given the automated experiment, it seems that the measure of exploring versus exploiting, B, should be set between 0.5 and 0.75 (see Fig. 10(*a*)). This means that more emphasis is placed on what the designer has ranked highly thus far (exploitation) as opposed to exploring the space more. It is important to see that in Fig. 10(*a*), the results that occur when B is 1 are significantly worse. This means that some amount of exploration is required to achieve good results. It seems likely that performance could be increased if the B and Q knobs are scheduled to change through the process in a manner similar to temperature in simulated annealing [31]. Additionally, the designer dialog could change size—perhaps starting with a large number of candidates and eventually ending in a single pair-wise comparison.

7 Future Work

The primary goal of this research is to develop the rule-based interactive tree-search algorithm and demonstrate its implementation on a simple design problem. Future research efforts will address more sophisticated generative grammars especially those being developed in engineering design. Currently, the developed technique has been shown to be effective at finding a target design with only a few simple interactive comparisons (one design greater-than, less-than, or equal to another). Now that the effectiveness has been shown it is important to test the work in more realistic domains.

In this work, the search method is guided only by the user's relative comparisons regardless of what the true preference or performance is comprised of. In a domain like mechanical product design, where real performance and costs can be computationally predicted, the search may also be able to predict and reason with the tradeoffs for such calculated metrics. In these problems, designs are also governed by functional requirements, physical and spatial constraints, and other performance criteria derived from customer needs. Nevertheless, the benefits of the proposed algorithm may be useful in many applications where input from designer or "human in the loop" is paramount to solving the problem. One benefit of the proposed approach is that the designer only needs to evaluate concepts. As a result, the user need not be an expert in the design of the artifact, and may in fact be a customer. Given a generative grammar that produces only valid and feasible configurations, the work may also be useful in automating the dialog necessary for mass customization [32].

8 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. Traditional generative algorithms optimize against explicit fitness functions, but design problems involving multiple competing objectives or qualitative measures, (such as the one chosen in this paper) cannot be easily reduced to a mathematical function of goodness [33]. 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.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant CMMI-0448806. Any opinions, findings, and conclusions or recommendations presented in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Cagan, J., 2001, "Engineering Shape Grammars," *Formal Engineering Design Synthesis*, E. K. Antonsson, and J. Cagan, eds., Cambridge University Press.
[2] Kurtoglu, T., and Campbell, M., 2009, "Automated Synthesis of Electromechanical Design Configurations From Empirical Analysis of Function to Form Mapping," J. Eng. Des., **20**(1), pp. 83–104.
[3] Sridharan, P., and Campbell, M. I., 2004, "A Grammar for Function Structures," Proceedings of ASME 2004 International Design Engineering and Technical Conference and Computers and Information in Engineering Conferences, Salt Lake City, UT, Sept. 28–Oct. 2, DETC04/DTM-57130.
[4] Soman, A., Padhye, S., et al., 2003, "Towards an Automated Approach to the Design of Sheet Metal Components," Artif. Intell. Eng. Des. Anal. Manuf., **17**(4), pp. 187–204.
[5] Kurtoglu, T., and Campbell, M., 2009, "An Evaluation Scheme for Assessing the Worth of Automatically Generated Design Alternatives," Res. Eng. Des., **20**(1), pp. 59–76.
[6] Mosbah, M., 1996, "Probabilistic Hyperedge Replacement Grammar," Theor. Comput. Sci., **159**(1), pp. 81–102.
[7] Agarwal, M., and Cagan, J., 1998, "A Blend of Different Tastes: The Language of Coffee Makers," Environ. Plan. B: Plan. Des., **25**(2), pp. 205–226.
[8] Shea, K., Cagan, J., and Fenves, S. J., 1997, "A Shape Annealing Approach to Optimal Truss Design With Dynamic Grouping of Members," ASME J. Mech. Des., **119**(3), pp. 388–394.
[9] Brown, K. N., and Cagan, J., 1997, "Optimized Process Planning by Generative Simulated Annealing," Artif. Intell. Eng. Des. Anal. Manuf., **11**, pp. 219–235.
[10] Schmidt, L., and Cagan, J., 1995, "Recursive Annealing: A Computational Model for Machine Design," Res. Eng. Des., **7**(2), pp. 102–125.
[11] Starling, A. C., and Shea, K., 2003, "A Grammatical Approach to Computational Generation of Mechanical Clock Designs," Proceedings of ICED'03 International Conference on Engineering Design, Stockholm, Sweden.
[12] Starling, A. C., and Shea, K., 2005, "Virtual Synthesizers for Mechanical Gear Systems," Proceedings of ICED'05 International Conference on Engineering Design, Melbourne, Australia.
[13] Pugh, S., 1991, *Total Design: Integrated Methods for Successful Product Engineering*, Addison-Wesley Publishing Company, Workingham, UK.
[14] Ullman, D., 1995, *The Mechanical Design Process*, McGraw-Hill, New York.
[15] Saaty, T., 1980, *The Analytic Hierarchy Process*, McGraw-Hill, New York.
[16] Keeney, R. L., and Raiffa, H., 1976, *Decisions With Multiple Objectives: Preferences and Value Tradeoffs*, John Wiley & Sons, New York.
[17] Orsborn, S., Cagan, J., and Boatwright, P., 2008, "Quantifying Aesthetic Form Preference in a Utility Function," ASME J. Mech. Des., **131**(6), pp. 397–407.
[18] Semet, Y., 2002, "Interactive Evolutionary Computation: A Survey of Existing Theory," University of Illinois, Technical Paper No. GE493DEG.

AQ5 AQ11 AQ6 AQ7 AQ8

[19] Takagi, H., 2001, "Interactive Evolutionary Computation: Fusion of the Capacities of EC Optimization and Human Evaluation," Proceedings of the IEEE 89, **9**, pp. 1275–1296.

[20] Juille, H., and Pollack, J. B., 1998, "A Stochastic Search Approach to Grammar Induction," *Grammatical Inference*, Vol. 1433 (Lecture Notes in Artificial Intelligence), Springer-Verlag, pp. 126–137.

[21] Sycara, K., and Navin-Chandra, D., 1989, "Integrating Case-Based Reasoning and Qualitative Reasoning in Engineering Design," *Artificial Intelligence in Engineering Design*, J. Gero, ed., Computational Mechanics Publications.

[22] Maher, M. L., 1988, "Engineering Design Synthesis: A Domain-Independent Representation," Artif. Intell. Eng. Des. Anal. Manuf., **1**(3), pp. 207–213.

[23] Chandrasekaran, B., 1990, "Design Problem Solving: A Task Analysis," AI Mag., Winter, pp. 59–73.

[24] Schmidt, W. E. 1989, "As Neckwear Goes, This Knot's News," N.Y. Times, August 30.

[25] Fink, T. M., and Yong, M., 2001, *The 85 Ways to Tie a Tie: The Science and Aesthetics of Tie Knots*, Fourth Estate Ltd.

[26] Campbell, M. I., 2006, "The Official GraphSynth Site," http://www.graphsynth.com, University of Texas at Austin.

[27] Hornby, G., 2004, "Functional Scalability Through Generative Representations: The Evolution of Table Designs," Environ. Plann. B, **31**(4), pp. 569–588.

[28] Box, G., and Wilson, K., 1954, "The Exploration and Exploitation of Response Surfaces: Some General Considerations and Examples," Biometrics, **10**(1), pp. 16–60.

[29] Goldberg, D. E., and Deb., K., 1991, "A Comparative Analysis of Selection Schemes Used in Genetic Algorithms," Found. Genet. Algorithms, **1**, pp. 69–93.

[30] Golub, G., and Van Loan, C., 1996, *Matrix Computations*, The Johns Hopkins Press.

[31] Kirkpatrick, S., Gelatt, C., Jr., and Vecchi, M., 1983, "Optimization by Simulated Annealing," Science, **220**, pp. 671–679.

[32] Pine, B. J., 1992, *Mass Customization: The New Frontier in Business Competition*, Harvard University Press, Cambridge, MA.

[33] Hornby, G., and Kurtoglu, T., 2009, "Toward a Smarter Web," Science, **325**, pp. ■–■.