Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco

Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Adam M. Smith, Bill Howe, Jeffrey Heer

Abstract—There exists a gap between visualization design guidelines and their application in visualization tools. While empirical studies can provide design guidance, we lack a formal framework for representing design knowledge, integrating results across studies, and applying this knowledge in automated design tools that promote effective encodings and facilitate visual exploration. We propose modeling visualization design knowledge as a collection of constraints, in conjunction with a method to learn weights for soft constraints from experimental data. Using constraints, we can take theoretical design knowledge and express it in a concrete, extensible, and testable form: the resulting models can recommend visualization designs and can easily be augmented with additional constraints or updated weights. We implement our approach in Draco, a constraint-based system based on Answer Set Programming (ASP). We demonstrate how to construct increasingly sophisticated automated visualization design systems, including systems based on weights learned directly from the results of graphical perception experiments.

Index Terms—Automated Visualization Design, Perceptual Effectiveness, Constraints, Knowledge Bases, Answer Set Programming

1 Introduction

Visualization designers benefit from familiarity with both the data domain under consideration and principles of effective visual encoding. Although designers can learn these principles from books, research papers, and experience, they do not always follow these principles in practice [6, 42]. Automated design tools [37, 66] are designed to help address this problem: they use formally-encoded design guidelines to promote effective visualizations. However, our design knowledge is incomplete and continually evolving. In order to incorporate new experimental results or compare different theories of effective design, we need to elaborate and refine these bodies of formal design knowledge.

Visualization researchers regularly publish empirical study results of how people decode and interpret visualizations (e.g., [24, 30, 46, 61]). However, new results often make their way into practical tools slowly: even though our knowledge is evolving, we lack a shared medium for representing and acting upon this knowledge. For example, existing automated design systems [37–39, 65] do not explicitly reuse the knowledge bases implemented in previous systems. Rather than building idiosyncratic representations of design knowledge for individual systems, we seek to make formal models of design knowledge a shared resource for the visualization community.

We present Draco, a formal model that represents visualizations as sets of logical facts and represents design guidelines as a collection of hard and soft constraints over these facts. Draco can systematically enumerate the visualizations that do not violate the hard constraints and find the most preferred visualizations according to the soft constraints. We first formulate a simple yet powerful visualization description language based on the Vega-Lite grammar [52] and then extend this language to express dataset and task characteristics. To represent design knowledge, we contribute a set of extensible constraints that can encode expressiveness criteria [37], preference rules validated in perception experiments, and general visualization design best practices.

We view the constraints in Draco as the starting point of an evolving knowledge base of design considerations for researchers and tool designers to extend and use. Hard constraints must be satisfied (e.g., shape encodings cannot express quantitative values), whereas soft constraints express a preference (e.g., temporal values should use the x-axis by default). By changing the weights associated with soft constraints, we can trade off the relative importance of these preferences. However, updating these weights presents a challenge, as local changes may have unexpected global effects. To update preferences in a principled manner, we also contribute a method to automatically configure weights from experimental data. By formulating this process as a *learning to rank* [36] problem, we can begin to integrate knowledge scattered across various research papers into a single system.

We implement Draco using Answer Set Programming, a domain-independent constraint programming language. We formalize the problem of finding appropriate encodings as the problem of finding optimal answer sets [18], which provides well-defined semantics and can be solved with efficient domain-independent algorithms.

We first evaluate Draco by using it to re-implement the APT [37] and CompassQL [65] automated design tools, demonstrating Draco's expressiveness and improved performance. We then show how Draco can go beyond these systems by adding new constraints concerning data and a user's primary task. Instead of manually specifying weights, we learn them from two independent graphical perception studies [30, 51]. We compare the learned visualization model to a hand-tuned model, demonstrating improved automated design suggestions.

Encoding design knowledge as constraints has many advantages from both practical and academic perspectives [58]. Tool builders can use evolving knowledge bases of best practices instead of (re)implementing ad-hoc rules, and can benefit from the efficient search algorithms provided by state-of-the-art constraint solvers. Most importantly, an independent knowledge base may allow anyone to formulate and disseminate design preferences as a small set of independent constraints and/or weight updates. Accordingly, we believe Draco can accelerate the transfer of research knowledge into practical tools. Researchers can also use Draco to systematically sample, enumerate, and reason about the design space of possible visualizations, or to concretely compare different design models. We make Draco available as open source software with supporting tools, documentation, and examples at `https://uwdata.github.io/draco/`.

- *Dominik Moritz, Chenglong Wang, Greg L. Nelson, Halden Lin, Bill Howe, and Jeffrey Heer are with the University of Washington. E-mails: domoritz, clwang, glnelson, haldenl, billhowe, jheer@uw.edu.*
- *Adam M. Smith is with the University of California Santa Cruz. E-mail: amsmith@ucsc.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

2 Related Work

Draco builds on prior work on automated visualization design systems, visualization specification languages, and constraint programming.

2.1 Automated Visualization Design

To recommend a visualization, automated design systems enumerate visual encodings that satisfy both user-defined constraints (such as which fields to visualize) and design constraints (Fig. 1). They then

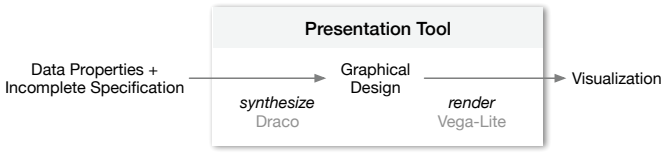

Fig. 1. A model of automated visualization design tools, inspired by APT. One component synthesizes a design from data and incomplete specifications, and the other renders the design. Draco produces Vega-Lite specifications as output.

rank candidate visualizations by a utility function. These systems may return the top encodings or perform subsequent clustering to avoid redundancy [65].

Mackinlay's APT system [37] automatically designs graphical representations of relational data using *expressiveness* and *effectiveness* criteria to prune and rank encoding choices. A visualization is considered *expressive* if it conveys all the facts in the data, and only the facts in the data. A visualization is considered most *effective* when the information it conveys is more readily perceived than with other visualizations. The original APT system consists of roughly 200 rules in a declarative logic programming language. The logic programming approach has a number of advantages over procedural approaches. First, it is flexible and can be adapted with additional constraints. Second, global optimization can find satisfiable solutions that procedural approaches may fail to identify. Draco takes the logic programming approach from APT but extends it in a few ways. The search strategy in APT is depth-first search with simple backtracking, which is inefficient for large design spaces. Draco uses a modern constraint solver and a standardized representation language. APT is built on a graphical algebra that is no longer used, whereas Draco synthesizes Vega-Lite, a more complete graphical language. Lastly, Draco can deal with multiple (possibly competing) criteria, which was beyond the scope of APT.

The SAGE project [39] extends APT by considering additional data properties such as cardinality, uniqueness, and functional dependencies. We adopt this idea in Draco. Casner's BOZ [8] additionally models the low-level perceptual tasks of reading and comparing values. In addition to value tasks, Draco's model includes summary tasks involving aggregate properties of visual ensembles [62]. ShowMe [38] uses heuristic rules to suggest encodings from groups of charts, including trellis plots. Draco similarly models faceting of data into trellis plots.

While traditional systems rely on carefully designed rules and defaults, more recent systems like Voyager's CompassQL [65–67] use hand-tuned scores to specify fine-grained criteria such as space efficiency and legibility based on encoding and data properties. Draco is most similar to CompassQL and its weighted preferences. However, CompassQL is implemented in imperative JavaScript. Moreover, like all prior systems presented in this section, it uses similar heuristics and ad-hoc rules. In contrast, Draco's learning approach offers a "programmatic" way to turn experimental results into preference rankings.

2.2 Effective Visualization Design

To rank candidate visualizations, automated design tools use models of visual encoding effectiveness. These models encode the insights of Bertin [4], Cleveland & McGill [9], and others that encoding effectiveness varies depending on the visualized data type and related perceptual tasks. For example, APT uses a ranking of encoding channels by data type informed by human-subjects studies of visual decoding performance. Other studies (e.g., [26]) confirm and extend such rankings.

However, most work on effectiveness focuses on the performance of reading or comparing individual marks in a visualization. Recent work investigates the effects of reading ensembles of visual elements [62]: for example, how users read aggregates [22], distributions, trends, or correlation [24]. Experimental results from Kim et al. [30] and Saket et al. [51] analyze how effectiveness varies by task. These results also show that effectiveness varies with respect to data characteristics, such as the cardinality or entropy of data fields. Draco's declarative design can combine classical work on effectiveness using strict preference rules with recent work that considers data and task characteristics in a single system. If activity-oriented [40] or low-level [2] task taxonomies were expressed as constraints, they could be used in automated design systems. To demonstrate this conceptually, we use a simple task classification into *value tasks* for reading and comparing values and *summary tasks* for comparing ensembles [30].

2.3 Visualization Specification

Automatic visualization tools synthesize graphical designs, which are abstract descriptions of visualizations (Fig. 1). For example, the underlying language for APT describes graphical techniques (e.g., color variation and position on axis) to encode information, whereas ShowMe [38] synthesizes encodings using VizQL [23]. Following CompassQL [65], Draco uses a logical representation of the Vega-Lite grammar [52].

Vega-Lite enables concise descriptions of visualizations by encoding data as properties of graphical marks. The language is inspired by other high-level visualization languages such as Wilkinson's Grammar of Graphics [64], Wickham's ggplot2 [63], and the VizQL formalism underlying Tableau. Vega-Lite represents single plots using a set of encoding definitions that map data fields to visual channels such as position, color, shape, and size. Encodings may include common data transformations such as binning, aggregation, sorting, and filtering. Vega-Lite was specifically designed to facilitate search and inference over the space of possible visualizations [31, 65, 66].

In addition to single plots, Vega-Lite supports faceting into trellis plots, layering, and arbitrary concatenation. In this paper, we focus on single views and faceted views (using the row and column encoding channels). Previous work has focused on similarly restricted design spaces [31,44,65,66]. This subset of Vega-Lite is capable of expressing a variety of plots of both raw and aggregate data, including bar charts, histograms, dot plots, scatter plots, line graphs, and area graphs.

2.4 Constraint-Based Knowledge Representation

Constraint programming is a declarative programming paradigm with wide applications in scheduling [12], graphical design [5], and natural language specification [45]. A constraint program is a set of constraints defining relations among several unknowns (i.e., variables) that must be or should be satisfied by its solutions. Constraints thus restrict the possible values that variables can take, representing partial information about the variables of interest [3]. Solutions are computed by constraint solvers via inference and search over the constrained space [28]. In visualization, the design recommendation process (i.e., generating, testing [50], subsequent ranking) can be modeled as a constrained combinatorial optimization problem.

The declarative nature of constraint programming allows users to focus on modeling high-level knowledge, while delegating low-level algorithmic details to off-the-shelf constraint solvers. In contrast, imperative implementations of recommendation systems are often hard to implement efficiently and maintain. For example, CompassQL wastes resources in enumerating and evaluating infeasible solutions. Changes in the specification may require a complete overhaul of generators with an imperative implementation [56]. When building on a constraint programming language of sufficient expressiveness (such as beyond-NP reach of ASP technology), complex tests can be folded into the generation part without inventing new algorithms [57].

In Draco, we model visualization knowledge using Answer Set Programming (ASP) [7, 35], a declarative constraint logic programming language that seeks to balance expressivity, efficiency, and ease of use. ASP has been deployed in contexts such as decision support systems [41], product configuration [59], and educational game design [57]. Draco uses Clingo [17, 19], a state-of-the-art answer set solver. Clingo leverages conflict reasoning [20] and heuristics [16] to direct the search process and efficiently solve problems with up to millions of variables. The Clingo guide [15] provides a comprehensive resource of the ASP language features that Draco uses (§3), advice for how to model problems (§1 and §6), and documentation of the constraint solver (§7).

2.5 Learning Preferences

Both CompassQL and Draco use weights of visualization features to trade-off competing effectiveness criteria. Although the weights can be defined by visualization experts, tuning these weights involves ad-hoc choices that are difficult to maintain and extend, especially when the visualization model is complex.

Instead, the preference model might be learned from data. For example, VizDeck [29] learns a linear scoring function that uses data statistics and chart type to predict users' up/down votes on presented charts. However, VizDeck's model features only capture direct correlations between data statistics and chart type, and its learning algorithm is limited to a small set of predefined visualizations.

To improve expressivity and extensibility, Draco leverages domain knowledge from experts (in the form of soft constraints) as visualization features. Draco's preference model forms a Markov logic network (MLN) [49], where the weights corresponding to soft constraints are learnable parameters reflecting preference levels. Draco can capture rich attribute relations using a small number of expert-defined rules due to the expressive and compact nature of MLNs [45, 55].

To train a recommendation model, a common practice is to use pairs of incomplete specifications and their corresponding optimal completion [54]. However, for visualization, such training datasets are not generally available and are hard to obtain because there typically does not exist a single optimal completion. Instead, with Draco we take a *learning to rank* [36] approach, where the preference model is learned from ordered pairs of visualizations (i.e., complete specifications). A dataset of ranked pairs can either be harvested from experimental data based on human-subject performance measures or solicited from experts. To learn a preference model, we use RankSVM [27] over the structural features defined by Draco's soft constraints.

3 Background: Answer Set Programming

The building blocks of ASP programs are *atoms*, *literals*, and *rules*. Atoms are elementary propositions that may be true or false. Literals are atoms A or their negation *not* A. Rules are expressions of the form A `:-` $L_1, \ldots, L_n$. where each L_i is a literal. The atom A of a rule (also called its head) is derivable (true) if all literals in the body (right of `:-`) $L_1, \ldots, L_n$ are true. A positive literal is true if it has a derivation, and a negative literal *not* B is true if the atom B does not have a derivation. For example, the rule `light_on :- power_on, not broken.` informally states that the light is on if we can derive that the power is on and there is no reason to say that the lamp is broken [7].

Rules can be bodiless or headless. A bodiless rule, such as `power_on :-.`, simply asserts the fact that its head is true. A fact can also be stated using only its head, e.g., `power_on.`. Headless rules of the form `:-` $L_0, \ldots, L_n$ are *integrity constraints* that derive false from their body. Thus, satisfying the body $L_0, \ldots, L_n$ results in a contradiction.

An ASP program consists of a set of rules. Note that `not` in an ASP program means "not derivable". For example, given the two rules described above, `power_on` can be derived from a fact (our bodiless rule), while `broken` cannot. Consequently, we can derive `light_on`. Such derivations are formally defined as *stable models* [21] or *answer sets*: sets of atoms that are consistent with the constraints, justified by a derivation, and minimal with respect to unknown facts. Answer Set Programming has a constructive flavor: negative literals need only be true, whereas positive ones must also be provable.

On top of the stable model formalism, the language of ASP [14] introduces powerful modeling constructs. *Aggregate* rules of the form $l\ \{A_0, \ldots, A_n\}\ k$ are read as: at least l and at most k atoms in the set $\{A_0, \ldots, A_n\}$ are true. Aggregates can appear in the head or body of a rule and aggregate rules can be used to define a design space. We can eliminate answer sets with integrity constraints (headless rules) to restrict the design space. *Soft constraints* are headless rules with an associated weight and are written as `:~` $L_0, \ldots, L_n$. $[w]$. Unlike hard constraints, soft constraints may be violated by a solution, but each violation of a soft constraint imposes a penalty (or cost) equal to its weight w. Soft constraints can express preferences in the search. In a program with soft constraints, an answer set is optimal if it minimizes the sum of weighted costs of all violated soft constraints. Although one can express richer forms of optimization in ASP (such as Pareto optimality for combining preferences without first establishing a fixed trade-off between them) [18], the weighting scheme for soft constraints is sufficient for the linear preference models we will learn from data.

Fig. 2. An example of a bar chart, its Vega-Lite specification (in Vega-Lite JSON), and its equivalent specification using Draco constraints (in ASP). The specification defines the marktype and encodings, which includes a specification of the fields, data type, and data transformations.

4 Modeling Visualization Design in Draco

In this section, we first describe how we model visualizations as sets of facts. We then explain the design space of our model and how we can query the model with constraints. Modern constraint solvers efficiently search for *optimal* visualization specifications within a defined space. Imperative systems, which use an exhaustive generate and test method, couple knowledge representation and the algorithm for finding effective designs. In Draco, solutions to the base problem of finding optimal designs should be found via automated search, whereas solutions to the higher level problem of what preferences should be used and how competing preferences are resolved should be determined by designers (via refinement of the design space and preference definition).

The term "optimal" here does not refer to the definitively best or "correct" visual design, as this would make two arrogant presumptions. First, the system would have to fully understand the user's intentions and their ability to read visualizations—an unlikely proposition. Visualization is always an abstraction where choices are made about what to emphasize. Second, visual analysis is an iterative process, involving any number of visualizations, not just a singular view. By "optimal", we refer to an optimal specification according to a set of formally-defined preferences: the system can find no other visualization that would be scored as preferred to this one. A user-facing application can show more than just an optimal visualization, and a user may select between multiple recommendations or refine their query until they have the right design(s) for their task [65, 66].

4.1 Mapping Visualization Specifications to Logical Facts

A Vega-Lite specification describes a single Cartesian plot with a backing dataset, a given mark type, and a set of one or more encoding definitions for the visual channels. Fig. 2 shows a Vega-Lite specification for a bar chart. Vega-Lite expects a relational table of records with named fields. The mark type specifies the geometric objects used to visually encode data records. Possible values include bar, point, area, line, and tick. The encodings determine how data fields map to visual properties of the marks. An encoding uses a visual channel such as spatial position (x, y), color, size, shape, or text. A detail channel can be used to add group-by fields in aggregate plots. An encoding includes the data field to visualize and its given data type (nominal, ordinal, quantitative, temporal). The data can also be transformed via binning, aggregation (sum, average, etc.), and sorting. An encoding may specify scales that define how the data domain maps to the visual range or guides that visualize scales (axis and legend). Examples include whether a scale domain should include zero or whether the scale is linear or logarithmic. If omitted, the Vega-Lite compiler will infer defaults based on the channel and data type.

We represent the Vega-Lite specification, user task, and data schema as a set of atoms. To enable reasoning over atoms, we encode them as *predicates* (i.e., relations or functions). A predicate defines what value is assigned to an attribute of a visualization specification.

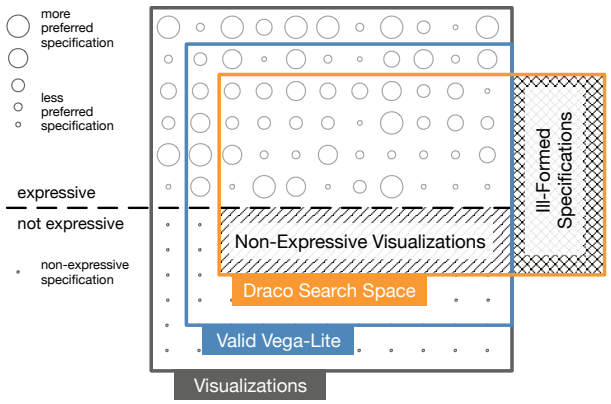

Fig. 3. Illustration of the design space in Draco. The set of valid Vega-Lite specifications is a subset of all possible visualizations, and Draco's design space overlaps with that subset. Given a preference model, expressive visualizations are ranked according to their preference scores in the model. Draco eliminates ill-formed or non-expressive specifications using hard constraints and encodes preferences using soft constraints.

To set the **mark** type, we use a predicate `mark/1`.[1] For example, `mark(bar).` defines that the visualization should use the bar mark (i.e., assign the value `bar` to the attribute `mark`).

To define an **encoding**, we use `encoding/1` to establish that it exists. For example `encoding(e).` declares the encoding `e`. We then use binary predicates to define properties of the encoding. `channel(e,x).` defines that the encoding `e` uses the `x` encoding channel. The field being visualized is defined with `field/2`. In addition, we use `aggregate/2` to define an aggregation function, `bin/2` to discretize continuous data, `stack/1` to define whether marks should be stacked, and `zero/1` and `log/1` to customize scales. Compared to Vega-Lite's JSON syntax, we un-nest scale properties to simplify the logical encoding.

The **data schema** is defined as the size of the data `num_rows/1` (e.g., `num_rows(42).`) and facts about data fields. We use `fieldtype/2` to specify the data type (string, number, date, ...) and `cardinality/2` to define how many distinct entries there are, `entropy/2` to define the entropy, and `extent/3` to set the minimum and maximum values. We see this set of data attributes as a starting point; future extensions of Draco can define other features relevant to automated design.

We currently model a user's **primary task** as a single function `task/1`. Following Kim et al. [30], we distinguish between *value* and *summary* tasks. Since tasks regard specific fields (e.g., "What is the maximum acceleration across cars?"), fields can be marked as relevant to the task with `interesting/2`.

We designed this logical visualization language to be extensible and enable reasoning. For example, we could have defined predicates for each channel such as `field_x`, `aggregate_x`. This design would automatically ensure that each channel is only used once; however, it would limit the expressiveness of the language (e.g., detail can in fact be used multiple times) and would make it more difficult to define general constraints over attributes that are not tied to a specific channel. One way of extending Draco is to define new attributes as predicates. For example, to add a data property that measures data skew, we can add `kurtosis/2` where the first argument is the field and the second is the kurtosis measure.

4.2 Representing Design Knowledge as Constraints

The goal of a visualization model is to distinguish desirable visualizations from undesirable ones. In Draco, our visualization model consists of two parts: the space of all visualizations considered valid, and an evaluation function over the space to measure preferences. Fig. 3 illustrates the design space. We describe a visualization model in Draco as a declarative answer set program.

[1]Per Prolog traditions, predicates are identified by their symbolic name and the number of arguments they take (signified with `/n`).

4.2.1 Design Space Definition

The design space of a visualization model is defined by a set of constraints. A visualization is considered valid only if all constraints are satisfied. Following best practices in logic program design [34], we define the space of possible visualization specifications with two sets of rules: (1) a set of aggregate rules that specifies the domains of the attributes defined in the previous section (mark, encoding, ...) and (2) a set of integrity constraints that defines how different attributes can interact with each other.

We use aggregate rules to define which values can be assigned to a visualization attribute. For example, the rule `1 { mark(bar); mark(line); mark(area); mark(point) } 1.` restricts choices of mark to one of bar, line, area or point. Similarly, we can use the constraint `0 { aggregate(E,count), aggregate(E,mean), aggregate(E,median), aggregate(E,sum) } 1 :- encoding(E).` to declare that each encoding may have up to one aggregate of count, mean, median, or sum. Note that `E` is capitalized, which identifies it as a variable. To extend the domain of an attribute (e.g., support tick marks), we would add facts to existing aggregate rules (e.g., by adding `mark(tick)`). The complete list of aggregate rules, and all other constraints in this section, are included as supplemental material.

The aggregate rules declare the domain of attributes, but not the validity of interactions of different attribute values. To capture such interactions, we introduce an additional set of constraints. Using ASP notation, we write these constraints as a headless rule (integrity constraint). `:- X.` is read as "it cannot be the case that X". Integrity constraints can be used to encode expressiveness and restrictions to the attributes of a visualization specification, for example, to match the implementation of Vega-Lite.

First, we use constraints to rule out specifications that do not specify a valid visualization (i.e., that are ill-formed or ungrammatical). We call these constraints *well-formedness* constraints. For example, the constraint `:- channel(_,shape), not mark(point).` is an integrity constraint stating that it cannot be the case that there exists a shape encoding unless the mark that is used to encode data is "point", as other mark types such as area, line, bar, or text cannot encode a shape. Another example is `:- log(E), zero(E).` which ensures that we do not synthesize a log scale that requires zero in its domain. We also assert that visualizations must use a text mark when the text channel is used (and vice versa) and that only discrete data can be mapped to facet (row and column) channels. Well-formedness depends on the syntax and semantics of the graphical language. We can use constraints to generate only visualization specifications that are supported by a concrete visualization model such as Vega-Lite. For example, Vega-Lite only implements 8 shape types; we can use the integrity constraint `:- channel(E,shape), cardinality(E,C), C > 8.` to model this restriction. When defining the design space, conflicting constraints must be avoided, as they result in an empty space that cannot be satisfied by *any* visualization.

Second, we use constraints to eliminate *non-expressive* visualizations that do not convey all and only the facts in the data. For example `:- mark(bar), channel(E,y), continuous(E), not zero(E).` ensures that the model will not consider vertical bar charts that do not use zero as a baseline. (We actually implement this as a more general rule `:- mark(bar), channel(E,(x;y)), continuous(E), not zero(E).`, which also covers horizontal bar charts. Here the semicolon denotes an expansion into disjunctions, implying one constraint for each channel type.) Another expressiveness constraint is `:- channel(E,size), type(E,nominal).`, which ensures that size is not used to encode nominal data, as size implies an order. We also assert that the size channel is only permitted for compatible marks, that zero baselines are used for area and bar charts, and that bar charts with a color channel encoding use stacking so that bars do not occlude each other.

4.2.2 Preference Over the Design Space

We now define an evaluation function over the visualization design space to encode preferences. The (linear) evaluation function maps a valid visualization specification into an integer representing its preference level. This function defines a total ordering over the design space,

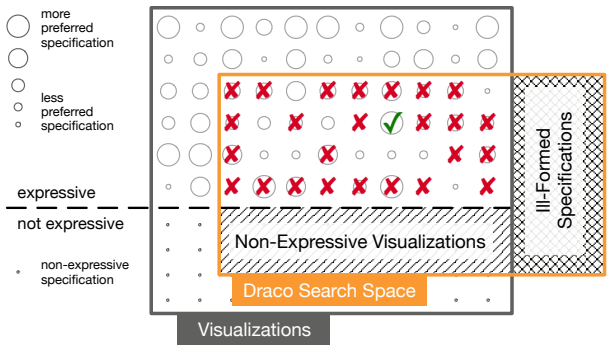

Fig. 4. To find the optimal completion of a partial specification, Draco compiles a user query into a set of constraints that removes all candidates (✗) that do not match the query from the design space illustrated in Fig. 3. Draco selects the optimal visualization (✓) within the remaining space.

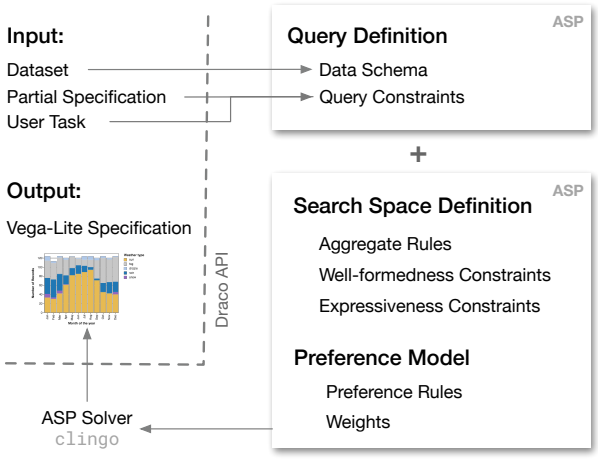

Fig. 5. Our implementation of the optimal encoding search process using constraints. Draco compiles a user query (including the dataset, the partial specification, and the task) into a set of rules and combines them with the existing knowledge base to form an ASP program. Draco then calls Clingo to solve the program to obtain the optimal answer set. Finally, Draco translates the answer set into a Vega-Lite specification.

as illustrated in Fig. 3. Instead of defining the evaluation function as a procedure, we use a set of soft constraints to implicitly define it. The weight of a soft constraint reflects its strength: the higher the weight (penalty), the higher cost that violating the constraint imposes on the cost of an answer set.

Soft constraints are similar to integrity constraints, but start with `:~` instead of `:-` and include a weight declared in square brackets. They can be read as "prefer not to . . . ". As an example, the soft constraint `:~ continuous(E), not zero(E). [5]` states that the model prefers to include zero for continuous fields and that violating the rule increases the cost of the visualization by 5. A soft constraint is appropriate: though omitting a zero baseline for ratio data can mislead [42], it is still sometimes reasonable to do. Note that a visualization may violate a soft constraint multiple times. For example, given a visualization with two encodings, the soft constraint above may be violated twice if two of its encodings use continuous fields but omit zero.

In order to extend Draco to support new visualization types or data properties, a visualization expert can add soft constraints to capture intended preferences. For example, in order to extend a visualization model to handle the relation between the new data property kurtosis (as discussed in the previous section) and using a log scale, we add the soft constraint `:~ kurtosis(F,K), field(E,F), log(E), K > 42. [w]`.

The set of soft constraints defines a cost model for visualizations in the design space that we can use to evaluate preferences. The cost of a visualization is the sum of the costs of all soft constraint violations multiplied by their count of violations. Concretely, given a set of soft

```
{
  "data": { "url": "cars.csv" },
  "encoding": [ {"channel": "x", "bin": true,
                 "field": "horsepower"} ]
}
```

Fig. 6. An example query over the cars dataset, in the form of a partial (incomplete) Vega-Lite specification.

constraints $S = \{(p_1, w_1), \ldots, (p_m, w_m)\}$, the cost of a visualization specification v is calculated as follows, where n_{p_i} is the number of violations of the soft constraint p_i by v:

$$Cost(v) = \sum_{i=1\ldots k} w_i \cdot n_{p_i}(v)$$

Given a visualization v, the vector $\mathbf{x} = [n_{p_1}(v), \ldots, n_{p_k}(v)]$ fully determines the cost of v in the given visualization model, and we refer to $\mathbf{x}$ as the *feature vector* of v. Using the feature vector x, the cost of v can be represented as $Cost(v) = \mathbf{x}^T \mathbf{w}$, where $\mathbf{w} = [w_1, \ldots, w_k]$ is the vector consisting of all soft constraint weights. Note that setting the weight w for this new rule requires the expert to know the existing constraints and carefully trade-off among competing preferences. In Sect. 5, we instead present a method to learn w from data.

Draco's preference model forms a Markov logic network (MLN), a graphical model that integrates logic with statistical reasoning to handle uncertainty in a robust way [49]. This interpretation as an MLN offers theoretical insight into the expressiveness of our model. The soft constraints are structural features of visualizations that capture hidden relations among visualization attributes, and their weights are learnable parameters reflecting their importance. We discuss MLNs in more detail in the appendix.

4.3 Completing Specifications by Solving Constraints

Given a visualization model, a user can query said model for optimal completions of a partially specified visualization. Fig. 4 illustrates the search process, while Fig. 5 summarizes our implementation of it.

4.3.1 Partial Specification

A user's query is a partial specification that describes their incomplete intent for a desired visualization. Our partial specification language allows the user to specify unknown visualization attributes by leaving them blank. Draco also supports converting CompassQL queries and Vega-Lite specifications into queries. In addition to a partial specification, a query can specify the schema of the dataset and the user's primary task.

As an example, Fig. 6 shows a query over the classic *cars* dataset [1]. In this query, the user specifies that "I want a visualization that shows binned horsepower along the x-axis". Using this query, Draco must then determine completions of the mark and other attributes of the specified encoding, as well as whether other encodings are necessary. Draco then searches to find the mark for the chart, completes the specified encoding, and potentially adds additional encodings (including which channels to use, whether to use aggregation, etc.).

4.3.2 Queries as Constraints over the Design Space

To answer a query, Draco first compiles the query into a set of facts and constraints that defines a subspace of visualizations and then searches over the subspace for the lowest-cost specifications (Fig. 4). Concretely, the subspace is defined by (1) a set of facts describing the dataset specified in the query (Sect. 4.1) and (2) a set of constraints that restrict the available choices for visualization attributes.

For example, the query in Fig. 6 is compiled into a set of facts and constraints. Unless the data schema is provided explicitly, Draco infers a schema (including fields, their types, and data properties) from the provided dataset ("cars.csv") and uses it to generate facts that describe the dataset's size and fields:

```
num_rows(407).
fieldtype(name,string).
cardinality(name,311).
```

```
fieldtype(miles_per_gallon,number).
cardinality(miles_per_gallon,130).
...
```

Based on the partial specification, Draco then generates a fact declaring an encoding `e1` and associated constraints. These constraints restrict the design space to specifications with an encoding `e1` that uses the `x` encoding channel for binned values from the horsepower `field`:

```
encoding(e1).
:- not channel(e1,x).
:- not field(e1,horsepower).
:- not bin(e1,_).
```

To find optimal specifications within the subspace, Draco sends data constraints, query constraints, and constraints from the knowledge base to the Clingo solver (Fig. 5). For example, Draco suggests the following optimal completion of the query above, which adds a new encoding `e2` on the `y`-axis for a `count` aggregate.

```
encoding(e2).
channel(e2,y).
aggregate(e2,count).
```

Finally, Draco converts the optimal solutions to Vega-Lite specifications and returns them to the user.

5 Learning Preference Models

Although it is possible for model designers to tune preference weights for small models, tuning weights for complex models is challenging: it requires the visualization expert to reason globally about competing conditions among different preferences. In this section, we describe a learning algorithm that allows the model to learn soft constraint weights $\mathbf{w} = [w_1, ..., w_k]$ from ranked pairs of visualizations.

We learn weights using a RankSVM (Support Vector Machine) model [27] trained on labeled visualization pairs. Given a visualization pair (v_1, v_2), the cost model should determine whether or not v_1 is preferred to v_2 based on $\text{sign}(Cost(v_1) - Cost(v_2))$. This model can be learned from a dataset where each entry $(v_1, v_2; y)$ is a visualization pair associated with a label y indicating if v_1 is preferred to v_2 ($y = -1$) or vice versa ($y = 1$). Given a visualization model with a set of soft constraints $S = \{p_1, \ldots, p_k\}$, we show how we train the cost model (i.e., training weights $\mathbf{w} = [w_1, ..., w_k]$ for S) using a dataset $\mathscr{D} = \{(v_{11}, v_{12}; y_1), \ldots, (v_{n1}, v_{n2}; y_n)\}$.

As shown in Sect. 4.2.2, the cost of a visualization v is determined by its feature vector $\mathbf{x} = [n_{p_1}(v), \ldots, n_{p_k}(v)]$. Accordingly, we first run Clingo on the complete specifications and count how often each soft constraint is violated to vectorize all visualizations in the dataset $\mathscr{D}$ and obtain their vector representation: $\mathscr{D}' = \{(\mathbf{x}_{11}, \mathbf{x}_{12}; y_1), \ldots, (\mathbf{x}_{n1}, \mathbf{x}_{n2}; y_n)\}$.

The cost model is a linear model over soft constraint weights. Given a pair (v_1, v_2) with feature vectors $\mathbf{x}_1, \mathbf{x}_2$, its class is determined by the sign of the following function:

$$f(v_1, v_2) = Cost(v_1) - Cost(v_2) = \mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2)$$

Using the RankSVM algorithm to train weights $\mathbf{w}$, we perform linear regression (with L_2 regularization) over the dataset $\mathscr{D}'$ by minimizing the hinge loss. The loss function L is defined as follows, and it is minimized by the solution $\mathbf{w}^*$.

$$L = \frac{1}{n}\sum_{i=1}^{k} \max\left(0, 1 - y_i \mathbf{w}^T(\mathbf{x}_{i1} - \mathbf{x}_{i2})\right) + \lambda \|w\|_2$$

$$\mathbf{w}^* = \arg\min_{\mathbf{w}} L$$

As the cost model is a linear model over inputs $(\mathbf{x}_{i1} - \mathbf{x}_{i2})$, the weights $\mathbf{w}^*$ can be efficiently found using an off-the-shelf linear optimizer. By minimizing the loss function L, we obtain a cost model with weights $\mathbf{w}^*$ that is most consistent with the rankings of visualization pairs in the dataset. The order of v_1, v_2 in a visualization pair from the training data does not matter, as the classification problem is symmetric with respect to the origin $(-y_i \mathbf{w}^T(\mathbf{x}_{i1} - \mathbf{x}_{i2}) = y_i \mathbf{w}^T(\mathbf{x}_{i2} - \mathbf{x}_{i1})$ in the loss function). Thus, a pair $(v_1, v_2; y)$ is equivalent to $(v_2, v_1; -y)$ in the training set and we can standardize all pairs in the form $(v_1, v_2; -1)$ (such that v_1 is preferred over v_2) without worrying about an imbalance between classes. For our initial experiments, we set the regularization parameter λ to 0.1.

By integrating the learned weights $\mathbf{w}^*$, the visualization model becomes a knowledge base for visualization recommendation that integrates both expert knowledge and empirical data.

6 Demonstration of Draco

We present three applications of Draco to demonstrate its expressivity, extensibility, and usability. First, we implement APT's preference rules via a set of strict preference constraints (Draco-APT); this shows Draco can express a classic yet useful automated design system. Next, we reimplement CompassQL by adding soft constraints with weights hand-tuned by experts to match the semantics of CompassQL (Draco-CQL). Finally, we introduce additional effectiveness criteria learned from data from two different studies (Draco-Learn); this shows how Draco can partially automate combining effectiveness results from different research studies.

6.1 Reimplementing APT: Draco-APT

Draco-APT provides a re-implementation of APT's channel preferences as a set of soft constraints. APT uses a principle of importance ordering: each field is assigned to the most effective channel (for the corresponding data type) in order of decreasing user-specified importance.

Draco-APT starts with the set of well-formedness and expressiveness constraints from Sect. 4.2. We add a set of soft constraints to express channel preferences. Each preference constraint is of the form `:~ type(E,T), channel(E,C), priority(E,P). [w@P,E]`, which states that for any encoding E, using channel C for a field of type T incurs a cost of w at priority level P equivalent to the priority of the field. To determine the optimal solution, the solver first satisfies all hard constraints followed by soft constraints, ordered by priority level.

```
:~ type(E,quant), channel(E,x), priority(E,P). [1@P,E]
:~ type(E,quant), channel(E,y), priority(E,P). [1@P,E]
:~ type(E,quant), channel(E,size), priority(E,P). [2@P,E]
:~ type(E,quant), channel(E,color), priority(E,P). [3@P,E]

:~ type(E,ordinal), channel(E,x), priority(E,P). [1@P,E]
:~ type(E,ordinal), channel(E,y), priority(E,P). [1@P,E]
:~ type(E,ordinal), channel(E,color), priority(E,P). [2@P,E]
:~ type(E,ordinal), channel(E,size), priority(E,P). [3@P,E]

:~ type(E,nominal), channel(E,x), priority(E,P). [1@P,E]
:~ type(E,nominal), channel(E,y), priority(E,P). [1@P,E]
:~ type(E,nominal), channel(E,color), priority(E,P). [2@P,E]
:~ type(E,nominal), channel(E,shape), priority(E,P). [3@P,E]
:~ type(E,nominal), channel(E,size), priority(E,P). [4@P,E]
```

Using Draco-APT, we can find optimal completions of partial specifications using APT's effectiveness criteria. For example, given a query with four fields with decreasing priority—two quantitative fields (encoded as `e_q1` and `e_q2`), one nominal field (`e_n`), and one ordinal field (`e_o`)—Draco-APT synthesizes the following two optimal results.

```
1 channel(e_q1,y) channel(e_q2,x) channel(e_n,color)
    ↪ channel(e_o,size)
2 channel(e_q1,x) channel(e_q2,y) channel(e_n,color)
    ↪ channel(e_o,size)
```

6.2 Reimplementing CompassQL: Draco-CQL

We now show that Draco is expressive enough to re-implement CompassQL [65], a state-of-the-art automated visualization design system that includes additional forms of effectiveness knowledge. We compare the original CompassQL implementation with our new implementation,

showing Draco-CQL is more concise, extensible, and provides superior performance when searching for optimal visualizations.

CompassQL uses a generate and test approach [50]: for a given partial specification, CompassQL generates all matching full specifications. The changes made to the partial specification may include both data query parameters (e.g., fields, aggregation) and encoding parameters (e.g., channels). It then uses data query, encoding, and expressiveness constraints similar to those described in Sect. 4.2 to eliminate invalid encodings. The system then assigns each valid candidate an effectiveness score. The scoring function incorporates preferences for type-channel interactions (e.g., it is preferred to encode nominal fields using x or y before using row or column) and mark-type interactions (e.g., point marks are preferred over tick marks for quantitative by quantitative plots). These effectiveness scores are then used to rank and recommend visualizations. Critically, this approach allows CompassQL to trade off among competing preferences.

We identified two places for improvement in the process taken by CompassQL. First, CompassQL generates and tests all possible candidate visualizations, which leads to an ineffective exhaustive search. By expressing the hard constraints as integrity constraints, we can pass this process off to a modern constraint solver. Moreover, ASP allows for concise representations. For example, the constraint that invalidates encodings that reuse channels that should only be used once, requires 14 lines of JavaScript code in CompassQL but can be expressed in Draco as `:- single_channel(C), 2 { channel(_,C) }.` stating that we prefer not to use a single channel 2 or more times.

Second, CompassQL's scoring function can be expressed naturally in Draco with soft constraints. For example, CompassQL penalizes aggregation when grouping by a continuous field, implemented in 12 lines of code. Draco's implementation is more concise and readable:

```
:~ aggregate(_,_), continuous(E), not aggregate(E,_). [3].
```

Reimplementing CompassQL in Draco requires authoring soft constraints that express CompassQL's imperative rankings and design preferences. These constraints include channel, mark type, and aggregation function rankings as well as soft constraints to prefer compact layouts (e.g., fewer encodings) or promote best practices such as placing time on the horizontal axis. For example, channel preferences for nominal fields can be expressed as follows, where lower weights (penalties) indicate higher preference:

```
:~ channel(E,y), type(E,nominal). [0]
:~ channel(E,x), type(E,nominal). [1]
:~ channel(E,row), type(E,nominal). [6]
:~ channel(E,column), type(E,nominal). [7]
:~ channel(E,color), type(E,nominal). [7]
:~ channel(E,shape), type(E,nominal). [8]
:~ channel(E,text), type(E,nominal). [9]
:~ channel(E,detail), type(E,nominal). [20]
```

Using a full set of these constraints, Draco-CQL synthesizes identical optimal specifications as CompassQL for all 17 partial specifications included in CompassQL's test suite. It does so while reducing specification complexity. Draco-CQL is implemented as 70 hard and 110 soft constraints. In contrast, CompassQL is implemented in 4,324 lines of imperative code, with 1,134 of those lines devoted to hard constraints and 786 devoted to scoring logic.

Draco-CQL also exhibits better performance, especially for highly unconstrained problems. Fig. 7 shows the results of a benchmark study comparing CompassQL and Draco-CQL across varied numbers of input dataset fields and output encoding channels. All measurements were taken on CentOS Linux 7 with an Intel Xeon CPU E5-2690 v3 with 2.60GHz; CompassQL used Node v9.9.0. Other than a constant startup overhead, Draco exhibits superior scalability. On a real dataset with 25 fields and a query with 5 encodings, Draco returns an answer in less than half a second. In contrast, CompassQL's exhaustive search runs out of heap memory after a few minutes.

6.3 Learning Preferences from Experiments: Draco-Learn

When developing automated visualization design systems, developers may hand-tune weights until the system synthesizes the desired

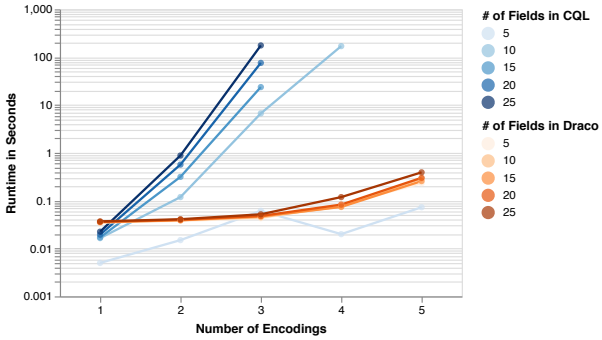

Fig. 7. Median runtime for CompassQL (blue) and Draco (orange) across different numbers of data fields and encodings. CompassQL performance rapidly degrades with additional encodings and runs out of heap memory (set to 4GB) for most queries with four or more encodings because it exhaustively searches all combinations of fields in the schema.

specifications across test cases. This process is time-consuming and error-prone. Instead, Draco can automatically learn parameters from data. Draco's preference model allows competing preferences and, via learning to rank, can learn weights for soft constraints from ranked pairs of visualizations. The same generalizability and validity issues that all empirical studies have also apply to Draco's empirically learned weights. Draco's flexible learning system allows us to harvest training pairs from data originating from different experimental studies, even those carried out under different conditions.

To demonstrate this advantage, we harvested ranked pairs from two recent experiments on effectiveness. Kim et al. [30] measured subject performance across task types and data distributions. They compared performance across 12 scatterplot encoding specifications of trivariate data involving 1 categorical and 2 quantitative fields, encoded with x and y channels along with the color, size, or row channel - in total, 185,000 responses from 1,920 participants. The visualizations tested by Kim et al. are only a fraction of the design space that Draco supports; thus, we are not able to learn the weights of a system that can compete with CompassQL from this data alone. Saket et al. [51] conducted a similar experiment with 180 participants to evaluate task performance across visualization types. Their study is limited to encodings with one quantitative y-encoding with a mean aggregate, and an x-encoding with nominal, ordinal, or quantitative data.

6.3.1 Harvesting Training Data and Learning Weights

For both studies, the data contains the visualization type, data properties, task, and whether the user correctly completed the task. To create ranked pairs, we first group the response data by data schema and task. Within each group, we group again by visualization and create every possible pair. The difference in task performance between the visualizations in each pair may or may not be significant. We use Fisher's test to check whether the accuracy scores are significantly different between the two visualizations. We keep only pairs where the p-value is lower than a threshold (in our case 0.01). We consider both accuracy and timing results and include a pair if either exhibits a significant difference. Ranked pairs of visualizations could be harvested from other studies in a similar fashion. Our harvesting results in about 1,100 pairs from Kim et al. and 10 pairs from Saket et al. We get few pairs for Saket et al. because for each data and task combination, only three visualizations are compared (Vega-Lite supports bar, line, and scatter) and few exhibit significant differences.

We then apply the learning approach described in Sect. 5. First, we transform every visualization that appears in the dataset into a feature vector of soft constraint violation counts in Draco. We start with Draco-CQL's constraints and add soft constraints for the preferences described in Kim et al.'s paper. Specifically, we added rules to capture task-channel and task-marktype interactions, along with a handful of rules for the most important interactions from the discussion of the paper (see supplemental material for a full listing). We implemented these rules in a few hours. These preference rules can be also included in

the CompassQL implementation from the previous section, as we can simply initialize the weights for new constraints to zero. With the new rules, we train a classifier on the difference between the two feature vectors for each pair of ranked visualizations using RankSVM. We trained an off-the-shelf SVM from scikit-learn [43] on the two datasets derived from the studies by Kim et al. and Saket et al.

6.3.2 Applying the Learned Model

We first evaluate our trained model directly on ranked pairs by measuring the percentage of pairs that are correctly ranked based on their costs. We train our model on a training set of 55% of the full data, validate on 15% of the data, and assess generalization of the final model with 30% test data [25]. The trained model achieves 93% accuracy on the test set, whereas Draco-CQL with hand-tuned weights achieves 65% accuracy—only slightly better than chance. Our model achieves perfect training accuracy on the dataset from Saket et al. even if we include all data harvested from Kim et al. Our model was able to learn from different datasets without degrading performance in either of them. The model correctly labels 96% of the validation dataset when we increase the p-value threshold for harvesting from 0.01 to 0.1. We also found that the test accuracy only starts to significantly fall behind the validation accuracy when we train on less than 250 pairs. This observation indicates that there is redundancy in the data and that our model generalizes. These results support our model and feature selection choices.

In practice, finding the optimal encoding given a dataset, task, and partial specification matters more than accuracy across all ranked pairs of visualizations. For example, correctly ordering the second and third best visualizations may matter less than getting the optimal encoding right. We built Draco-Learn using only the trained weights for the soft constraints in our preference model. First, we restrict the design space to only those encodings used in Kim et al. and Saket et al.'s studies (as described at the beginning of this section) by adding about 10 additional constraints each (included in supplemental material). Adding these constraints adapts Draco-Learn to synthesize only specifications that are in a restricted design space. We then query Draco-Learn to synthesize specifications for all combinations of data properties (cardinality and entropy) and tasks (value and summary). In all conditions (48 for Kim et al., 8 for Saket et al.), Draco-Learn returns a top-performing encoding for the harvested data.

Draco-Learn outperforms Draco-CQL within the restricted design space covered by the experimental data. Fig. 8 shows the optimal visualizations synthesized by Draco with default weights (Draco-CQL) and with learned weights (Draco-Learn) for a specification with three encodings across value and summary tasks. In Draco-CQL, the weights for all features related to task are zero. Consequently, Draco-CQL synthesizes the same specification regardless of task: a scatterplot with the primary variable (Q1) on y and category (N) on color. Draco-Learn synthesizes different charts depending on the task. To compare individual values, the scatterplot works well and reduces overplotting. However, to summarize Q1 relative to N, spatially grouping values by category (N) better facilitates perception of distributional properties such as min, max, or average [30].

7 Discussion and Future Work

We presented Draco, a formal model for visualization knowledge representation, and demonstrated its use for automated visualization design. Draco models visualization design knowledge using constraints and associated weights; this separation of knowledge representation from search procedures enables easier development and maintenance. The Draco-APT and Draco-CQL examples demonstrate how Draco can support increasingly sophisticated visualization design tools with less development effort and better performance than prior approaches. The Draco-Learn example demonstrates that Draco can combine data from different studies to learn weights for a state-of-the-art visualization design tool, further accelerating modeling efforts.

We now discuss how Draco's constraint system can enable new usage scenarios, such as design space enumeration, visualization model comparison, and design debugging. We go on to describe how future work might address current limitations of Draco's implementation.

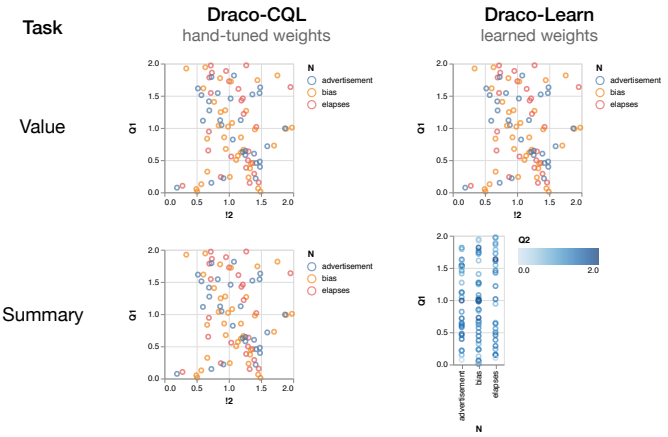

Fig. 8. The optimal visualization synthesized by Draco with hand-tuned weights (left) and Draco with learned weights (rights) for two queries with varying tasks. Draco with default weights cannot distinguish by task as the weights for all soft constraints related to task are set to zero.

7.1 Draco from a Software Engineering Perspective

Draco's use of constraint programming enables easier development and maintenance of automated visualization design tools. Due to implementation complexity, prior approaches often have to compromise the implementation of effectiveness criteria. Although Draco does not completely solve this problem, it shifts the problem to the more tractable and well-defined problem of defining weights to trade-off competing preferences. We show that these weights can be effectively learned from data even when the dataset is assembled from different sources.

Using constraints also decouples knowledge representation from the code that applies that knowledge. Although this approach aims to benefit visualization tool developers, it may also benefit end-users as it makes knowledge bases easier to contribute to. Visualization researchers can disseminate their results as constraints to make them more easily accessible by visualization designers; in a declarative system, designers might use more nuanced models that would otherwise be too complex to maintain. We contend that software engineering and developer productivity should be given more attention in the visualization community. Human designers should focus on the design of the visualization design space and preferences rather than the design of search algorithms that are already available in domain-independent constraint solvers.

Draco's knowledge base can be adapted or extended to fit specific needs. Each component of Draco can be easily modified: the definition of the design space, the preferences within the valid solution space, their weights, and how the visualization model is queried. For example, in Sect. 6.3.2, we limited the design space to scatter plots with three encodings (two quantitative, one nominal). The same expressiveness and preference constraints could be used in a tool that targets full Vega-Lite specifications or in a tool that targets only specific visualizations, such as vertical bar charts. Similarly, Draco can be extended with richer descriptions of input data (Sect. 4.1) that can then be referenced by new soft constraints (Sect. 4.2.2). A researcher who wants to extend the Draco knowledge base with new design rules could distribute their rules as an independent set of constraints or updates to the weights.

We hope that Draco's current set of constraints can serve as the starting point of an evolving knowledge base that can be extended by researchers and practitioners. For example, Draco could be extended to include richer task taxonomies [2]. One challenge for visualization design tools is that the "task" is typically inaccessible (e.g., within a user's mind). Natural language interfaces may be better suited for communicating user intent [13, 53, 60], which could then be expressed as Draco constraints. The visualization model in this paper supports synthesis of marks, encodings, and simple transformations (binning

and aggregation). We plan to extend the model to transformations such as filtering and sorting, and incorporate Vega-Lite's interaction primitives [52].

We are excited to explore how our visualization model can be extended to support chart composition, for instance into layered views or dashboards. Applying design guidelines to multiple charts separately can lead to locally effective, yet globally inconsistent views [47]. For example, different fields might confusingly be encoded with the same color scheme across charts. With the right set of weighted constraints, Draco could trade-off among the effectiveness of single views and global consistency within a multi-view display [47, 66].

In our demonstration of Draco-Learn, we modeled a restricted subspace of visualizations that mirrors the limits of the available experimental data. We hope to encourage more researchers to make data from effectiveness studies available, such that their results may be used by Draco or related systems. Future work might provide tools to help researchers convert their results into constraints or ranked pair datasets. We plan to collect more comprehensive data by systematically generating visualization pairs and having human subjects evaluate them. In addition to independent studies, we might leverage Draco's design space to guide data collection in an active learning process.

With sufficient data, it may even be possible to go beyond learning weights and attempt to learn preference rules themselves. The AI community uses inductive logic programming methods to infer logic programs from databases of positive and negative examples [48]. To learn from noisy data (common in the visualization domain!), we could combine inductive logic programming with statistical models such as Markov logic networks [11, 32]. For example, Law et al.'s ILASP (Inductive Learning of Answer Set Programs) [33] is a logic-based learning system that can learn preferences in answer set programs. To understand differences in preferences represented by two or more distinct data sources, we can use multi-objective (Pareto) optimization in ASP to enumerate designs that map the trade-off frontier.

Because the effectiveness of a visualization can depend on low-level features not captured in a high-level specification (for example, over-plotting), we can imagine applying a *re-ranking* strategy in which Draco enumerates a number of top-scoring candidate designs (ranked by high-level features) that are then re-ranked by another learned classifier operating on low-level features that may be impractical to model directly in ASP. The sub-symbolic models learned by such classifiers could constitute another valuable form of visualization design knowledge to represent and share.

7.2 Beyond Automated Visualization Design

Up to this point, we have positioned Draco as a tool for synthesizing optimal visualization designs from partial specifications. However, Draco could be used in a variety of other contexts. In the following, we discuss four directions that Draco could be extended.

First, Draco can be used as a general "visualization spell checker" to validate and auto-correct designs independently, or within a broader system for people to "learn by doing". Currently, Draco is able to use expressiveness and effectiveness constraints to report errors for designs that violate design guidelines. However, given a visualization, we could extend Draco to additionally automatically correct the visualization, removing the most severe violations and suggesting alternative valid designs to users. The problem of finding the minimal set of constraints that need to be removed for the remaining constraints to be satisfiable is known as the *unsatisfiable cores problem* [10]; related techniques could be applied to visualization design constraints. Draco might also explain those violations and why they matter, to teach students or visualization designers about best practices, help them spot (intentionally or unintentionally) misleading visualizations, critique visualizations, and perhaps contribute new visualization knowledge or explanations.

Second, Draco can facilitate exploration of the visualization design space. Besides surfacing violations of design guidelines, Draco can rank visualizations by their costs. Designers might use this function of Draco to choose among different alternative designs. Draco could also be used to cluster designs based on their violations (using the same feature vectors used in our learning to rank approach). An exciting avenue for future research is to use Draco's design space definition to systematically generate visualizations to build a corpus of visualizations and interactions. Creating such a corpus is as simple as running Clingo on the Draco design space definition without preferences, which enumerates all valid answer sets. Testing generated designs with human subjects will allow us to understand the costs and benefits of different encodings and interactions. Although the current design space in Draco is limited, as noted above we plan to extend the model further, including interaction primitives such as Vega-Lite selections [52].

Third, Draco can be used as a tool for researchers to compare the implications of different effectiveness studies. Concretely, if a researcher finds a new design guideline, they could add it to Draco as a constraint and assess whether it conflicts with, or is subsumed by, existing design guidelines. Based on comparison results, researchers could share their design results as constraints to improve the common knowledge base of visualization design tools.

Lastly, an important future extension is tooling to support developers, researchers, and designers. In addition to collecting more data to learn preference weights, we hope to provide tools to browse the visualization design space and knowledge base rules, as well as tools to understand violations and fine-tune trade-offs among competing design guidelines. With the right tooling and fine-tuned visualization models, Draco's declarative approach to automated visualization design could bring us one step closer towards building assistive interfaces for effective design that canvas a much broader swath of the visualization design space. Such interfaces should allow visualization designers to consider a greater variety of approaches, while also focusing on the creative aspects of visualization design.

Appendix: Preference Models as Markov Logic Networks

Our preference model forms a Markov logic network (MLN) [49] that describes a distribution over visualizations. In MLNs, soft constraints are structural features of the model, and their weights reflect the difference in log probability between a visualization satisfying the constraint and one that does not. The joint distribution modeled by a MLN is:

$$P(v) = \frac{e^{-Cost(v)}}{\sum_{u \in \mathscr{V}} e^{-Cost(u)}} = \frac{e^{-\sum_{i=1}^{k} w_i n_{p_i}(v)}}{\sum_{u \in \mathscr{V}} e^{-\sum_{i=1}^{k} w_i n_{p_i}(u)}}$$

The probability $P(v)$ of a visualization in the distribution is its exponentiated cost normalized by the exponentiated costs of all visualizations in the design space $\mathscr{V}$, using a softmax function. Note that the partition function $Z = \sum_{u \in \mathscr{V}} e^{-\sum_{i=1}^{k} w_i n_{p_i}(u)}$ is a fixed term in a given visualization model, and the difference of costs of two visualizations v_1, v_2 results in the log difference of their probability in the model.

The problem of finding the optimal completion of a partial specification is the same as performing maximum a posteriori (MAP) inference in the probability model [54]. Given a partial specification Y, its optimal completion X maximizes the posterior probability of $P(x \mid y = Y)$ (i.e., $X = \max_x P(x \mid y = Y)$). The ASP solver solves this inference problem efficiently by minimizing the overall cost of the generated visualization; it is not necessary to compute the partition function.

Since a MLN is a linear model over structural features rather than linear model directly over attributes, it has the advantage of capturing structural relations between attributes that cannot be captured otherwise. For example, a channel ranking that is independent of the data type would have to prefer color and size independent from the data type. Moreover, as opposed to explicitly representing correlations between every attribute and all other attributes, a MLN is more compact and interpretable.

Acknowledgments

We are immensely grateful to Alan Borning, Kevin Jamieson, Pedro Domingos, Kanit "Ham" Wongsuphasawat, and Danyel Fisher whose advice shaped this work. We also thank members of the HCI group at UW and the anonymous reviewers for their feedback. This work was supported by a Moore Foundation Data-Driven Discovery Investigator Award and the National Science Foundation (IIS-1758030).

REFERENCES

[1] D. W. Aha, D. F. Kibler, and M. K. Albert. *Instance-based prediction of real-valued attributes*, vol. 5. 1989.

[2] R. A. Amar, J. Eagan, and J. T. Stasko. Low-level components of analytic activity in information visualization. In *IEEE Symposium on Information Visualization (InfoVis 2005), 23-25 October 2005, Minneapolis, MN, USA*, p. 15, 2005. doi: 10.1109/INFOVIS.2005.24

[3] R. Barták. Constraint programming: What is behind. *Proceedings of CPDC99*, pp. 7–15, 1999.

[4] J. Bertin. Semiology of graphics: diagrams, networks, maps. 1983.

[5] A. Borning. ThingLab - an object-oriented system for building simulations using constraints. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence. Cambridge, MA, USA, August 22-25, 1977*, pp. 497–498, 1977.

[6] J. Boy, R. A. Rensink, E. Bertini, and J. D. Fekete. A principled way of assessing visualization literacy. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1963–1972, Dec 2014. doi: 10.1109/TVCG.2014.2346984

[7] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, Dec. 2011. doi: 10.1145/2043174.2043195

[8] S. M. Casner. Task-analytic approach to the automated design of graphic presentations. *ACM Transactions on Graphics*, 10(2):111–151, 1991. doi: 10.1145/108360.108361

[9] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984. doi: 10.1080/01621459.1984.10478080

[10] G. Davydov, I. Davydova, and H. K. Büning. An efficient algorithm for the minimal unsatisfiability problem for a subclass of cnf. *Annals of Mathematics and Artificial Intelligence*, 23(3):229–245, Nov 1998. doi: 10.1023/A:1018924526592

[11] R. Evans and E. Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61:1–64, 2018.

[12] M. S. Fox. *Constraint-Directed Search: A Case Study of Job-Shop Scheduling*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, December 1983.

[13] T. Gao, M. Dontcheva, E. Adar, Z. Liu, and K. G. Karahalios. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, UIST '15, pp. 489–500. ACM, New York, NY, USA, 2015. doi: 10.1145/2807442.2807478

[14] M. Gebser, A. Harrison, R. Kaminski, V. Lifschitz, and T. Schaub. Abstract gringo. *TPLP*, 15:449–463, 2015.

[15] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele. Potassco user guide. *Institute for Informatics, University of Potsdam, second edition edition*, 2015. `https://github.com/potassco/guide/releases/`.

[16] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Answer Set Solving in Practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3):1–238, 2012. doi: 10.2200/S00457ED1V01Y201211AIM019

[17] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[18] M. Gebser, R. Kaminski, and T. Schaub. Complex optimization in answer set programming. 11, 07 2011.

[19] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, Apr. 2011.

[20] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, p. 386, 2007.

[21] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. pp. 1070–1080. MIT Press, 1988.

[22] M. Gleicher, M. Correll, C. Nothelfer, and S. Franconeri. Perception of average value in multiclass scatterplots. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2316–2325, 2013. doi: 10.1109/TVCG.2013.183

[23] P. Hanrahan. Vizql: a language for query, analysis and visualization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, p. 721, 2006. doi: 10.1145/1142473.1142560

[24] L. Harrison, F. Yang, S. Franconeri, and R. Chang. Ranking visualizations of correlation using weber's law. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):1943–1952, 2014. doi: 10.1109/TVCG.2014.2346979

[25] T. Hastie, R. Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009.

[26] J. Heer and M. Bostock. Crowdsourcing graphical perception: Using mechanical turk to assess visualization design. *Proceedings of the 28th Annual Chi Conference on Human Factors in Computing Systems*, pp. 203–212, 2010. doi: 10.1145/1753326.1753357

[27] R. Herbrich, T. Graepel, and K. Obermayer. Support vector learning for ordinal regression. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, vol. 1, pp. 97–102 vol.1, 1999. doi: 10.1049/cp:19991091

[28] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *The journal of logic programming*, 19:503–581, 1994.

[29] A. Key, B. Howe, D. Perry, and C. R. Aragon. Vizdeck: self-organizing dashboards for visual analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pp. 681–684, 2012. doi: 10.1145/2213836.2213931

[30] Y. Kim and J. Heer. Assessing effects of task and data distribution on the effectiveness of visual encodings. *Computer Graphics Forum (Proc. EuroVis)*, 2018.

[31] Y. Kim, K. Wongsuphasawat, J. Hullman, and J. Heer. GraphScape: A model for automated reasoning about visualization similarity and sequencing. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pp. 2628–2638. ACM, New York, NY, USA, 2017. doi: 10.1145/3025453.3025866

[32] S. Kok and P. Domingos. Learning the structure of markov logic networks. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pp. 441–448. ACM, New York, NY, USA, 2005. doi: 10.1145/1102351.1102407

[33] M. Law, A. Russo, and K. Broda. The ILASP system for learning answer set programs. `https://www.doc.ic.ac.uk/~ml1909/ILASP`, 2015.

[34] V. Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, jun 2002. doi: 10.1016/S0004-3702(02)00186-8

[35] V. Lifschitz. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pp. 1594–1597. AAAI Press, 2008.

[36] T.-Y. Liu. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval*, 3(3):225–331, 2009. doi: 10.1561/1500000016

[37] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986. doi: 10.1145/22949.22950

[38] J. D. Mackinlay, P. Hanrahan, and C. Stolte. Show Me: Automatic presentation for visual analysis. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1137–1144, 2007. doi: 10.1109/TVCG.2007.70594

[39] V. O. Mittal, G. Carenini, J. D. Moore, and S. Roth. Describing complex charts in natural language: A caption generation system. *Computational Linguistics*, 24(3):431–467, 1998.

[40] T. Munzner. *Visualization analysis and design*. CRC press, 2014.

[41] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An a-prolog decision support system for the space shuttle. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pp. 169–183. Springer-Verlag, London, UK, UK, 2001.

[42] A. V. Pandey, K. Rall, M. L. Satterthwaite, O. Nov, and E. Bertini. How deceptive are deceptive visualizations?: An empirical analysis of common distortion techniques. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pp. 1469–1478. ACM, New York, NY, USA, 2015. doi: 10.1145/2702123.2702608

[43] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[44] J. Poco, A. Mayhua, and J. Heer. Extracting and retargeting color mappings from bitmap images of visualizations. *IEEE Transactions on Visualization*

and Computer Graphics, 24(1):637–646, Jan 2018. doi: 10.1109/TVCG.2017.2744320

[45] H. Poon and P. Domingos. Unsupervised semantic parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, EMNLP '09, pp. 1–10. Association for Computational Linguistics, Stroudsburg, PA, USA, 2009.

[46] Z. Qu and J. Hullman. Keeping multiple views consistent: Constraints, validations, and exceptions in visualization authoring. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):468–477, Jan 2018. doi: 10.1109/TVCG.2017.2744198

[47] Z. Qu and J. Hullman. Keeping multiple views consistent: Constraints, validations, and exceptions in visualization authoring. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):468–477, Jan 2018. doi: 10.1109/TVCG.2017.2744198

[48] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, Aug 1990. doi: 10.1007/BF00117105

[49] M. Richardson and P. Domingos. Markov logic networks. *Machine Learning*, 62(1):107–136, Feb 2006. doi: 10.1007/s10994-006-5833-1

[50] S. F. Roth and J. Mattis. Data characterization for intelligent graphics presentation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 193–200. ACM, 1990.

[51] B. Saket, A. Endert, and C. Demiralp. Task-based effectiveness of basic visualizations. *IEEE Vis (Proc. InfoVis)*, 2018.

[52] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer. Vega-Lite: A grammar of interactive graphics. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):341–350, 2017. doi: 10.1109/TVCG.2016.2599030

[53] V. Setlur, S. E. Battersby, M. Tory, R. Gossweiler, and A. X. Chang. Eviza: A natural language interface for visual analysis. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pp. 365–377. ACM, New York, NY, USA, 2016. doi: 10.1145/2984511.2984588

[54] P. Singla and P. Domingos. Discriminative training of markov logic networks. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*, AAAI'05, pp. 868–873. AAAI Press, 2005.

[55] P. Singla and P. Domingos. Entity resolution with markov logic. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM '06, pp. 572–582. IEEE Computer Society, Washington, DC, USA, 2006. doi: 10.1109/ICDM.2006.65

[56] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović. A case study of expressively constrainable level design automation tools for a puzzle game. In *Proceedings of the International Conference on the Foundations of Digital Games*, pp. 156–163. ACM, 2012.

[57] A. M. Smith, E. Butler, and Z. Popovic. Quantifying over play: Constraining undesirable solutions in puzzle design. In *Proceedings of the Foundations of Digital Games, FDG*, pp. 221–228, 2013.

[58] A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, Sept 2011. doi: 10.1109/TCIAIG.2011.2158545

[59] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pp. 305–319. Springer-Verlag, London, UK, UK, 1998.

[60] A. Srinivasan and J. Stasko. Natural language interfaces for data analysis with visualization: Considering what has and could be asked. In *Proceedings of EuroVis*, vol. 17, pp. 55–59, 2017.

[61] D. A. Szafir. Modeling color difference for visualization design. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):392–401, Jan 2018. doi: 10.1109/TVCG.2017.2744359

[62] D. A. Szafir, S. Haroz, M. Gleicher, and S. Franconeri. Four types of ensemble coding in data visualizations. *Journal of Vision*, 16(5):11, 2016. doi: 10.1167/16.5.11

[63] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2016.

[64] L. Wilkinson. *The grammar of graphics*. Springer Science & Business Media, 2006.

[65] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Towards a general-purpose query language for visualization recommendation. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics - HILDA '16*, pp. 1–6, 2016. doi: 10.1145/2939502.2939506

[66] K. Wongsuphasawat, D. Moritz, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager: Exploratory analysis via faceted browsing of visualization recommendations. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):649–658, 2016. doi: 10.1109/TVCG.2015.2467191

[67] K. Wongsuphasawat, Z. Qu, D. Moritz, R. Chang, F. Ouk, A. Anand, J. Mackinlay, B. Howe, and J. Heer. Voyager 2: Augmenting visual analysis with partial view specifications. *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, pp. 2648–2659, 2017. doi: 10.1145/3025453.3025768