Excel Constraints

March 8, 2016

1 Use cases

There are different ways that learned constraints could be used in practice:

- Find mistakes
- Suggest a formula for a specific field
- Suggest a next value / autosuggest
- Find structure and functions in a plain text sheet (such as a csv)

2 Desired output

The annotated example (fig. 1) demonstrates the most used constraints in Excel which we aim to identify automatically.

Aggregates Aggregate functions like sum, max, average, ... that reduce a range to a value

Conditional aggregates Aggregates that use a filter on their input

Series Ranges of integer numbers, ascending or descending (a special case of a permutation)

Lookups Exact or fuzzy lookups that use a key to find a corresponding value

Ranks Uses a range and a value to determine an order over elements

Structural constraints Foreign keys to test value consistency between ranges

Previous Uses row values and the previous value to compute the current value

Notable omission so far is the *if-then-else* constraint. Whether it will be included and if how this will be done exactly is still to be determined.
Generally these constraints can be subdivided into:

- Row constraints
- Column constraints
- Inter-table constraints
- Nested constraints

Figure 1: Demo example

3 Approach

Currently we are considering a ModelSeeker like approach were ranges and values are generated and constraints tested upon them. One option would be to use constraint-specific generators to avoid the explosion of the search space. Meta-information would be used to find the most specific constraints. A heuristic could use various information to determine which learned constraints are useful.

3.1 Group discovery

Vectors consist of columns or rows that are consistent in their type (e.g. the Salesperson column or the Average row). Groups are sets of vectors that are consistent in their type and size. If only neighboring vectors are allowed to be added to a group, the group consists of a range. The groups partition the data and for every group subgroups may be defined that can overlap.
Groups describe subsets of data and can be used to form an $m \times n$ matrix with a specific type. Constraints can then be defined in terms of these matrices.
Our current approach would be to precompute all groups and subgroups and store metadata about their content (e.g. type, size) and hierarchy (e.g. child and parent groups).
An important consideration is that some data could be considered input while other data might have to be computed. This influences the search algorithm.

3.2 Finding constraints

Currently we see three approaches to finding constraints:

Constraint view In this approach the search for constraints is structured foremost from the point of view of the constraints. This means that for every type of constraint groups and transformations are searched such that they fulfill the constraint.

Vector view From an Excel point of view constraints are typically functional. Constraint are either a property of a vector (e.g. series) or describe how a specific vector can be calculated in terms of other vectors (e.g. sum). This lead us to the vector view in which for every vector constraints are searched that hold on the vector or describe how to calculate it.

Set view The last approach is to generate sets of groups and test various constraints on the set. This approach seems simpler but also more naive than the previous approaches.

In order to choose what groups may occur in what constraints and what combinations are valid, metadata about the constraints is also required (e.g. consistency in sizes, types). Additionally the system needs information about the relation between different constraints to remove redundancies.

3.3 Transformations

So far the concept of transformations is only defined losely. The idea is that some constraints cannot act on groups without applying a transformation on one or multiple groups first (e.g. the *previous* transformation). Therefore, there should be a catalog of transformations which might be applied to groups. As with constraints, metadata about the transformations should be given.
It is yet a bit unclear what the difference is between transformations and constraints. On one hand, one might see some constraints as transformations and vice versa. One could generalize to *functions* that map input to output. On the other hand, however, a property of transformations could be that they preserve their input size and distantiate them from constraint / functions in that way. Considering transformations separately might avoid some of the combinatorial explosion that arbitrary function chaining might provide.
One notable decision about transformations in general will be if the system supports types of backwards reasoning. An easy example of this would be adding a constant to the input or multiplying it with a constant.

4 Notation

4.1 Ranges

Every table T is considered like a matrix. Indexing allows to specify a subset of the table, for example, $T[r, c]$ represents the cell at row r and column c. The colon : is used to represent ranges. For example, $T[:, c]$ selects all cells in column c and $T[r_1{:}r_2, :]$ selects all cells in the rows r_1 to r_2. Disjoint columns can be specified by using $T[:, \{c_1, ..., c_n\}]$.

4.2 Example

Aggregates There are two types of aggregates in the example: The *Total* column consists of the sum of the four previous columns and the *Total* row consist of the sum of the column corresponding to each cell. These constraints can be written as:

$$T_1[:, 7] = SUM(T_1[:, 3{:}6], row) \tag{1}$$
$$T_2[:, 1] = SUM(T_1[:, 3{:}7], column). \tag{2}$$

Conditional aggregates Conditional aggregates form more challenging constraints. There is an arbitrary reduction in size of the input data based on a condition to be matched. The complexity of this constraint is also dependent on the restrictions imposed on the constraint part. The conditional sum that calculates the *Items sold total* column uses exact matching the salesperson name. Sticking close to the Excel notation this could be written as:

$$T_1[:, 10] = SUMIF(T_5[:, 1], T_1[:, 2], T_5[:, 2]). \tag{3}$$

Here, the condition will be applied to the *Salesperson* column of table 5 (the test vector). Generally speaking the test vector has to match the sum column in size and the condition vector has to match the output vector. However, the condition can also consist of a boolean test such as > 5 or a single value. To represent these constraints the Excel style using strings or exact values could be used but this might change yet to accomodate possibly more expressive conditions.

Unary The *ID* column consists of a series:

$$SERIES(T_1[:, 1]). \tag{4}$$

Series are general cases of permutations. If a vector of size n is a permutation of the numbers $[1, n]$, this can be an interesting observation, but often it is an interesting property of the vector which might unlock other constraints. See, for example, the column *Rank* which is a permutation ($PERMUTATION(T_1[:, 8])$). This property allows the rank constraint to be applied to it (See below). Similarly, all-different constraints can be uselful as constraint or precondition. For numberic data all-different is not very useful but it might be for textual data such as colum *Salesperson* ($ALLDIFFERENT(T_1[:, 2])$). Here, the all-different property allows the column to be refered to by a foreign key.

Lookups Consider the exact lookup in the example. This constraint can be expressed as:

$$T_4[:, 3] = LOOKUP(T_1[:, 1], T_1[:, 2], T_4[:, 2]). \tag{5}$$

Lookups, however, also support more complex constraints. For example, a fuzzy lookup assigns values to intervals to lookup the output value. Moreover, the input values can be transformed. The fuzzy lookup for the *Label* column demonstrates this:

$$T_1[:, 9] = LOOKUP(T_1[:, 7]/T_2[2, 5], T_3[:, 1], T_3[:, 2]). \tag{6}$$

Here, the *Total* column is divided by the calculated total average before being used to lookup what interval the performance falls into.

Ranks The column *Rank* ranks based on the the column *Total*, which can be expressed as:

$$T_1[:, 8] = RANK(T_1[:, 7]). \tag{7}$$

Structural constraints The *Salesperson* column in table 5 can be seen as a foreign key refering to the corresponding column in table 1. There are different options of representing this constraint:

$$UNIQUE(T_5[:, 1]) \subseteq T_1[:, 2] \tag{8}$$
$$FOREIGNKEY\,(T_1[:, 2], T_5[:, 1]) \tag{9}$$

Equation 8 uses a size-changing transformation or non-reversible function to express the underlying property. Instead one could also introduce a binary constraint expressing the foreign-key property directly (as in equation 9).

Previous Using the previous value in the constraint for a field can be solved easily by using a transformation. The calculation of the *Total* column of table T_6 can be expressed as:

$$T_6[:, 4] = T_6[:, 2] - T_6[:, 3] + PREVIOUS(0, T_6[:, 4], column). \tag{10}$$

The previous transformation drops the last value and adds an initial value, maintaining the size of the transformation input. In this specific case it is worth noting that this constraint can also be expressed differently:

$$T_6[:, 2] = SUM\,(T_6[:, 3{:}4], row). \tag{11}$$

This constraint makes a different assumption about which columns are considered input and which are considered output. If no additional information is given, the algorithm could decide to provide both (redundant) constraints, or make a decision based on generality / specificity or a heuristic.

5 Constraint-view Algorithm

5.1 Input

The input to the algorithm consists of a set of version spaces S describing a hierarchical set of vectors.
Additionally the user specifies if he wants the most general, specific or all solutions. This bias is captured is captured in the refinement operator ρ_2 which traverses version spaces based on the bias.
Implicitly this algorithm also uses a catalogue of constraints C, an oracle O to calculate constraints and background knowledg B about 1) constraints and 2) properties.

5.2 Output

This algorithm ρ_1 returns a set of constraints where the variables are instantiated by (sub-)version spaces contained in S.

5.3 Method

In this case we will assume that a constraint $c \in C$ has be selected to be learned.

- For every free variable x in c, prune S to obtain candidate values S_x using the properties that have been calculated so far: repeat:
- For every version space s in S_x, choose s as a value for x and proceed
- Use ρ_2 to return the