Learning constraints in tabular data

***Abstract*—Spreadsheets, CSV (comma separated value) files and other tabular data representations are in wide use today. However, modeling, maintaining and discovering formulas in tabular data and between spreadsheets can be time consuming and error-prone. In this work, we investigate the automatic discovery of constraints (functions and relations) from raw tabular data. We see multiple promising applications for this technique, e.g. in rediscovering constraints, auto-completion and error checking. Our method takes inspiration from inductive logic programming, constraint learning and constraint satisfaction. Common spreadsheet functions are represented as predicates whose arguments must satisfy a number of constraints. Constraint learning techniques are used to identify predicates (constraints) that hold between blocks of rows and columns of the tables. We show that our approach is able to accurately discover constraints in spreadsheets from various sources.**

I. Introduction

Millions of people across the world use spreadsheets every day. The tabular representation of the data is often intuitive, and the programming of functions in individual cells is quickly learned. However, large and complex sheets (possibly with multiple tables and relations between different them) can be hard to handle. Many end-users lack the understanding of the underlying structures and dependencies in such sheets and the data they contain. Especially when spreadsheets have been exported from other software such as Enterprise Resource Planning (ERP) systems; in this case, often a comma-separated values (CSV) format is used meaning that all formula's are lost, including inter-sheet formulas and relations. Even in manually created spreadsheets, it can be challenging to be consistent and correct with formula's across big spreadsheets. For example the influential Reinhart-Rogoff economical paper "Growth in a Time of Debt" had some of its claims contested [8], after an investigation of the used Excel sheets was shown to contain some mistakes in formulae.

In this paper, we investigate whether learning techniques can be used to infer constraints (formula's and other relations) from raw spreadsheet data. This is a new and unconventional machine learning problem. Consider the example in Figure 1a, where the header's names already suggest the usage of spreadsheet operations such as *average* or *sum*. Looking at the first row in Table 2 and the data in Table 1, it is clear that the values in this row were computed by summing the values in the corresponding column of Table 1. Examining Table 1 also shows that computations are not only performed column-wise but row-wise as well: the cells in the column **Total** are obtained by taking the row-wise sum of the previous four columns. This provides a flavor of how this problem is different from standard data mining settings, where the data is just in rows and variables are in columns. Here everything is mixed. The data is relational on the one hand, since we have multiple tables with relationships between them. And on the other hand, the data is most often mixed textual and numeric. One can see that understanding and learning constraints in tabular data is hence a new and challenging problem for machine learning.

The approach that we introduce borrows techniques from logical and relational learning [15], where the discovery of clausal constraints has been studied in an inductive logic programming setting [16, 9]; and from constraint learning, where several approaches to learning sets of constraints have been developed in a constraint programming setting [2, 3, 1]; and from mining in databases, where inductive techniques have been applied to discover constraints (such as functional and multi-valued dependencies) in relational databases [17]. It also builds on work on program synthesis, in particular, on Flashfill [7], where the definition of a function (over textual cells only) is learned in spreadsheet data from very few examples. Our approach contrasts with these in that it focusses on learning both column- and row-constraints, as does Modelseeker [1] which is restricted to traditional CSPs. It contrasts with Flashfill in that it can learn from numeric data too, as well as general constraints in addition to functions. A more detailed discussion on related work is contained in Section VI.

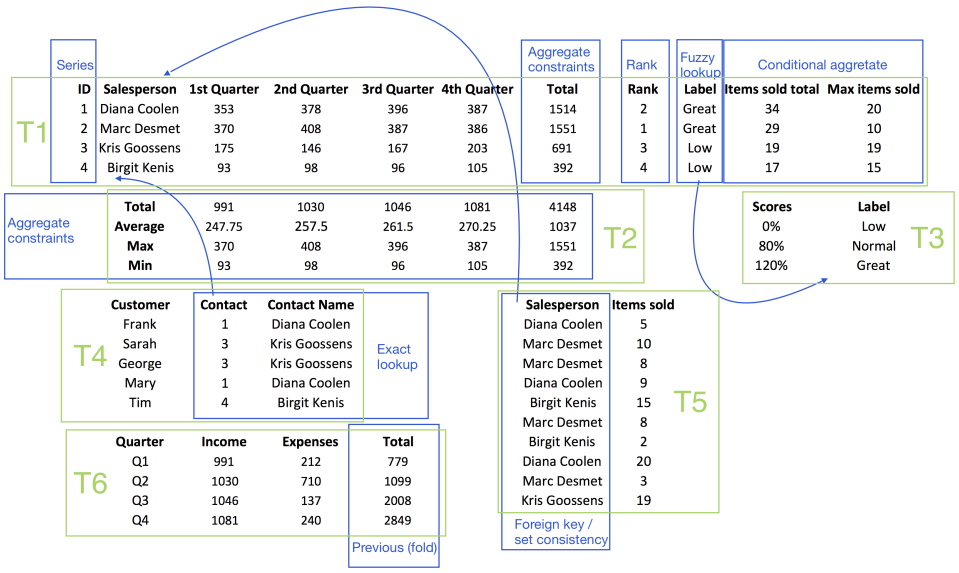
The question that we answer in this paper is: is it possible to discover or reconstruct structural constraints (relations, functions) in flat tabular spreadsheet data? To answer this question we contribute a general-purpose method and system, named *TaCLe* (pronounced as the word "tackle"), for discovering row-wise and column-wise constraints. It operates directly on (headerless) tables of a spreadsheet in an unsupervised setting, as it reasons on that raw tabular data directly with no example constraint instantiations given. We demonstrate the utility of our approach in an experimental evaluation. Moreover, we sketch additional application scenario's such as autocompletion and error detection.

This paper is organized as follows. Section II introduces concepts relevant to our approach. Section III presents the problem statement and the approach. Section IV presents the evaluation of the approach. Section V shows how our system can be used for applications. Section VI presents an overview of the related work. Section VII provides conclusions.

II. Formalization

Our goal is to automatically discover constraints (functions and relations) between the rows and the columns of tables in a spreadsheet. This is applicable not just to data from spreadsheets, but any data in tabular form, hence the name.

We first introduce some terminology and the concept of *constraint template*, after which we define the problem and make some additional considerations.



(a) Example spreadsheet (black words and numbers only). Green borders indicate headerless tables; the blue elements are added to clarify some constraints and relations. TIAS: TODO: headers not in green boxes SERGEY: show blocks? orientation only over blocks, not over tables? TIAS: Use specific constraint names, e.g. 'SUM-row' above Total in T1

$SERIES(T_1[:,1])$
$PERMUTATION(T_1[:,1]), PERMUTATION(T_1[:,8])$
$T_1[:,8] = RANK(T_1[:,3]), T_1[:,8] = RANK(T_1[:,4])$
$T_1[:,8] = RANK(T_1[:,7]), T_1[:,1] = RANK(T_1[:,5])$
$T_1[:,1] = RANK(T_1[:,6]), T_1[:,1] = RANK(T_1[:,10])$
$T_1[:,7] = SUM_{row}(T_1[:,3{:}6])$
$T_1[:,10] = SUMIF(T_5[:,1], T_1[:,2], T_5[:,2])$
$T_1[:,11] = MAXIF(T_5[:,1], T_1[:,2], T_5[:,2])$
$T_2[1,:] = SUM_{col}(T_1[:,3{:}7])$
$T_2[2,:] = AVERAGE_{col}(T_1[:,3{:}7])$
$T_2[3,:] = MAX_{col}(T_1[:,3{:}7]),$
$T_2[4,:] = MIN_{col}(T_1[:,3{:}7])$
$T_4[:,2] = LOOKUP(T_4[:,3], T_1[:,2], T_1[:,1])$
$T_4[:,3] = LOOKUP(T_4[:,2], T_1[:,1], T_1[:,2])$
$T_6[:,2] = SUM_{col}(T_1[:,3{:}6])$
$T_6[:,4] = PREV(T_6[:,4]) + T_6[:,2] - T_6[:,3]$

(b) Constraints present in the running example (left), except *ALLDIFFERENT* (16) and *FOREIGNKEY* (5).

A. Terminology

Spreadsheets and tabular data may conceptually consist of multiple tables, such as in Figure 1a. Note that a table can contain a header; however, we wish to reason over entire rows and columns of data, and hence we will consider **headerless tables** only.

Formally, a (headerless) table is an $n \times m$ matrix. Each entry is called a *cell*. A cell has a **type**, which can be numeric or textual. We further distinguish numeric types in subtypes: integer and float. We also consider *None* as a special type when a cell is empty; *None* is a subtype of all other types.

A row or a column is **type-consistent** if all cells in that row or column are of the same base type, that is, numeric or textual. We will use notation $T[a,:]$ to refer to the a-th row of table T, and similarly $T[:,a]$ for the a-th column. For example in Figure 1a, $T_1[1,:] = [1,2,3,4]$ and $T_5[:,1] = [$'Diana Coolen', $5]$. The latter is not type-consistent while the former is.

The most important concept is that of a **block**.

Definition 1. A **block** has to satisfy three conditions: 1) it contains only entire rows or entire columns of a single headerless table 2) it is contiguous and 3) it is type-consistent. The rows or columns have to be contiguous in the original table meaning that they must visually form a block in the table; and each of the rows/columns has to be of the same type. If it contains only rows we say it has *row-orientation*, if only columns, *column-orientation*.

In line with this definition, we can use the following notation to refer to blocks: $B = T[a{:}b,:]$ for a row-oriented block containing rows a to b in table T; and similarly $B = T[:,a{:}b]$ for a column-oriented block. We will refer to the *vectors* of a block when we wish to refer to its rows/columns independent of their orientation.

A block has the following properties:

- *type*: a block is type-consistent, so it has a type
- *table*: the table that this block belongs to
- *orientation*: either row-oriented or column-oriented
- *size*: the size of a block is the number of vectors in it, that is, the number of rows or columns.
- *length*: the length of its vectors; as all vectors are from the same table, they always have the same length.
- *rows*: this is orientation specific, for row-based blocks this is the size, for column-based blocks the length
- *columns*: also orientation specific, for row-based blocks this is the length, for column-based blocks the size
- TIAS: What is missing is 'discrete', not needed?

Example 1. Consider headerless Table 1 in Figure 1a, its rows are not type consistent (i.e. they contain both numeric and textual data). The table can be partitioned into the following five column-oriented blocks:

$$B_1 = T_1[:,1], \quad B_2 = T_1[:,2],$$
$$B_3 = T_1[:,3{:}8], \quad B_4 = T_1[:,9],$$
$$B_5 = T_1[:,10{:}11].$$

TIAS: Indicate in figure? see also Sergey's comment in figure

Definition 2. Block containment $\sqsubseteq$. A block B' is contained in a block B, $B' \sqsubseteq B$, iff both are valid blocks (single orientation, contiguous, type consistent) and each of the vectors in B' is also in B. For row-oriented blocks: $B' \sqsubseteq B \Leftrightarrow B = T[a{:}b,:] \wedge B' = T[a'{:}b',:] \wedge a \leq a' \wedge b' \leq b$ and similarly for column-oriented blocks.

SERGEY: do we need to mention explicitly that blocks have the same orientation and are in the same table? We will sometimes write that B' is a *subblock* of B or that B is a *superblock* of B'. An example is that $T_1[:,3{:}6] \sqsubseteq T_1[:,3{:}8]$, which contains the sales numbers of all employees for the four quarters.

B. Constraint templates

The goal is to learn constraints over blocks in the data. The knowledge needed to learn a constraint is expressed through *constraint templates*. A *constraint template* s is a triple $s =$ *(Syntax, Signature, Definition)*:

- *Syntax* specifies the syntactic form of the constraint $c(B_1, ..., B_n)$, that is, the name of the constraint c together with n abstract arguments B_i. Thus a constraint c is viewed as a relation or predicate of arity n in first order logic. Note that a function $B_r = f(B_1, ..., B_n)$ can be represented with the $(n+1)$-ary predicate $c_f(B_r, B_1, ..., B_n)$. Each argument will have to be instantiated with a block.
- the *Signature* defines the properties that the arguments of the predicate must satisfy. This can be any property of individual blocks, such as type, length, size and orientation, as well as relations between properties of arguments, for example that the corresponding blocks must belong to the same table, or have equal type or length. In terms of logical and relational learning [15], the Signature is known as the *bias* of the learner, it specifies when the constraint its arguments are well-formed.
- *Definition* is the actual definition of the constraint that specifies when the constraint holds. Given an assignments of blocks to its arguments, it can be used to verify whether the constraint is satisfied or not by the actual data present in the blocks. In logical and relational learning this is known as the background knowledge.

Example 2. Several constraint templates are illustrated in Table I. A non-trivial example is the constraint template for the row-based sum:

- Syntax: $B_r = SUM_{row}(\mathbf{B_x})$, where B_r and $\mathbf{B_x}$ are the arguments;
- Signature: B_r has to be a single vector (*size* $= 1$) while $\mathbf{B_x}$ can be a block (*size* $>= 1$), which can be derived from the use of a normal or **bold** font. The two blocks have to be numeric. This constraint is orientation-specific, so it requires that the number of rows in $\mathbf{B_x}$ equals the length of B_r. While not strictly needed, we also add to the bias of this template that the number of columns to sum over is larger than 2 as a block with two columns will also be captured by the $B_r = B_1 + B_2$ constraint;
- Definition: each value in the vector B_r is obtained by summing over the corresponding row in $\mathbf{B_x}$.

It is helpful to see the analogy of constraint templates with first order logic (FOL) and constraint satisfaction. From a FOL perspective, the name of the constraint (e.g. *SUM*) is just the predicate name, and the arguments B_r and B_x are the terms, which can be seen as either uninstantiated variables or as concrete values. This also holds in our setting, where an instantiation of a variable corresponds to a concrete block. For example for $B_r = RANK(B_x)$ and the spreadsheet in Figure 1a, when we write $T_1[:, 8] = RANK(T_1[:, 7])$, then the value of B_r is the 8th vector in T_1: $B_r = T_1[:, 8] = [2, 1, 3, 4]$ and the value of B_x is the 7th vector: $B_x = T_1[:, 7] = [1514, 1551, 691, 392]$.

With this interpretation, we can speak about the signature and definition of a constraint template being *satisfied*. We say that a signature (definition) of a constraint template with n arguments is satisfied by the blocks $(B_1, ..., B_n)$ if $Sig_c(B_1, ..., B_n)$ (respectively $Def_c(B_1, ..., B_n)$) is satisfied. Likewise, the template is satisfied if both the signature and definition are satisfied; in logic programming, we would write a Prolog-like clause: $c(B_1, ..., B_n) \leftarrow Sig_c(B_1, ..., B_n) \wedge Def_c(B_1, ..., B_n)$. Under this interpretation, the term constraint and constraint template can be used interchangeably.

Definition 3. A **valid argument assignment** of a constraint c is a tuple $(B_1, ..., B_n)$ such that $c(B_1, ..., B_n)$ is satisfied, that is, both the signature and the definition of the corresponding constraint template are satisfied by the assignment of $(B_1, ..., B_n)$ to the arguments.

C. Problem Definition

The problem of learning constraints from tabular data can be seen as an inverse *constraint satisfaction problem* (CSP). In a CSP one is given a set of constraints over variables that must all be satisfied, and the goal is to find an instantiation of all the variables that satisfies these constraints. In the context of spreadsheets, the variables would be (blocks of) cells, and one would be given the actual constraints and functions with the goal of finding the values in the cells. The inverse problem is, given only an instantiation of the cells, find the constraints that are satisfied in the spreadsheet.

We define the **Tabular Constraint Learning Problem** as follows:

Definition 4. *Tabular Constraint Learning.*
Given a set of instantiated blocks $\mathcal{B}$ and a set of *constraint templates* $\mathcal{S}$: **find** all constraints $c(B'_1, ..., B'_n)$ where c is a constraint with a corresponding template in $\mathcal{S}$ and $(B'_1, ..., B'_n)$ is a valid argument assignment of the constraint c.

Figure 1b shows the solution to the tabular constraint learning problem when applied on the blocks of Figure 1a and constraint templates listed in Table I.

D. Other considerations

1) Dependencies: In Table I one can see that for some constraints we used the predicate of another constraint in its signature, e.g. for *PERMUTATION*. This expresses a dependency of the constraint on that other constraint. This can be interpreted as follows: the signature of the constraint consists of its own signature plus the signature of the depending constraint, and its definition of its own definition plus the definition of the depending constraint. In FOL, we can see that one constraint entails the other, for example if $PERMUTATION(B_x)$ holds for a block B_x, then $ALLDIFFERENT(B_x)$ also holds.

Apart from easing the specification of the signature and definition, in Section III we will see how such dependencies can be used to speed up the search for constraints.

TABLE I: Overview of constraint templates implemented in *TaCLe*. The templates marked with † are aggregate constraint templates, the table shows the version for sum but the implementation also supports max, min, average, product and count. Subgroups in normal font have to be vectors, while subgroups in **bold** may contain more than one vector. Templates marked with $*$ are *structural*, i.e. they are not functional and cannot be implemented in most spreadsheet software.

Syntax	Signature	Definition
ALLDIFFERENT$(B_x)^*$	*discrete*(B_x)	All values in B_x are different: $B_x[i] \neq B_x[j]$ if $i \neq j$
PERMUTATION$(B_x)^*$	*numeric*(B_x), *ALLDIFFERENT*(B_x)	The values in B_x are a permutation of the numbers 1 through *length*(B_x).
SERIES(B_x)	*integer*(B_x) and *PERMUTATION*(B_x)	$B_x[1] = 1$ and $B_x[i] = B_x[i-1] + 1$.
FOREIGNKEY$(B_{fk}, B_{pk})^*$	B_{fk} and B_{pk} are both *discrete*; *type*(B_{fk}) = *type*(B_{pk}); *table*(B_{fk}) $\neq$ *table*(B_{pk}); and *ALLDIFFERENT*(B_{pk})	Every value in B_{fk} also exist in B_{pk}
$B_r =$ *LOOKUP*$(B_{fk}, B_{pk}, B_{val})$	B_{fk} and B_{pk} are both *discrete*; arguments $\{B_{fk}, B_r\}$ and $\{B_{pk}, B_{val}\}$ within the same set have the same *length*, *table* and *orientation*; B_r and B_{val} have the same type; and *FOREIGNKEY*(B_{fk}, B_{pk}).	$B_r[i]$ is the same value as looking up $B_{fk}[i]$ in B_{pk} and returning the corresponding value in B_{val}.
$B_r =$ *LOOKUP*$_{fuzzy}(B_{fk}, B_{pk}, B_{val})$	*Same as lookup*	$B_r[i]$ is the same value as looking up the last item in B_{pk} smaller than $B_{fk}[i]$ and returning the corresponding value in B_{val}.
$B_r = B_1 \times$ *LOOKUP*$(B_{fk}, B_{pk}, B_{val})$	Arguments $\{B_r, B_1, B_{fk}\}$ are *numeric*, arguments $\{B_{pk}, B_{val}\}$ are *discrete* and within both sets all arguments have the same *length*, *table* and *orientation*; also *FOREIGNKEY*(B_{fk}, B_{pk}).	$B_r[i]$ is the obtained by multiplying $B_1[i]$ with *LOOKUP*$(B_{fk}, B_{pk}, B_{val})[i]$.
$B_r = B_1 \times B_2$	Arguments $\{B_r, B_1, B_2\}$ are all *numeric* and have the same *length*	$B_r[i] = B_1[i] \times B_2[i]$.
$B_r = B_1 - B_2$	Arguments $\{B_r, B_1, B_2\}$ are all *numeric* and have the same *length* and *orientation*	$B_r[i] = B_1[i] - B_2[i]$.
$B_r =$ *PROJECT*$(\mathbf{B_x})$	Arguments $\{B_r, \mathbf{B_x}\}$ all have the same *length*, *orientation*, *table* and *type*; $\mathbf{B_x}$ contains at least 2 vectors; and $B_r = SUM(\mathbf{B_x}, orientation(\mathbf{B_x}))$	At every position i in 1 through *length*(B_r) there is exactly one vector v in $\mathbf{B_x}$ such that $v[i]$ is a non-blank value, then $v[i] = B_r[i]$.
$B_r =$ *RANK*(B_x)	*integer*(B_r); *numeric*(B_x); and *length*(B_r) = *length*(B_x)	The values in B_r represent the rank (from largest to smallest) of the values in B_x (including ties)
$B_r =$ *PREV*$(B_r) + B_{pos} - B_{neg}$	Arguments $\{B_r, B_{pos}, B_{neg}\}$ are all *numeric* and all have the same *length*, which is at least 2	The values in B_r are a running total, each value $B_r[i] = B_r[i-1] + B_{pos}[i] - B_{neg}[i]$.
$B_r =$ *SUM*$_{row}(\mathbf{B_x})^\dagger$	B_r and $\mathbf{B_x}$ are *numeric*; *columns*$(\mathbf{B_x}) \geq 2$; and *rows*$(\mathbf{B_x})$ = *length*(B_r)	Each value in B_r is obtained by summing over the corresponding row in $\mathbf{B_x}$.
$B_r =$ *SUM*$_{col}(\mathbf{B_x})^\dagger$	B_r and $\mathbf{B_x}$ are *numeric*; *rows*$(\mathbf{B_x}) \geq 2$; and *columns*$(\mathbf{B_x})$ = *length*(B_r)	Each value in B_r is obtained by summing over the corresponding column in $\mathbf{B_x}$.
$B_r =$ *SUMIF*$(B_{fk}, B_{pk}, B_{val})^\dagger$	B_{fk}, B_{pk} are *discrete*; B_r, B_{val} are *numeric*; within the sets $\{B_{val}, B_{fk}\}$ and $\{B_{pk}, B_r\}$ arguments have the same *length* and *orientation*; B_{fk} and B_{val} have the same *table*; B_{fk} and B_{pk} must have different *table*s but the same *type*; and *ALLDIFFERENT*(B_{pk})	The value for $B_r[i]$ is obtained by summing all values $B_{val}[j]$ where $B_{fk}[j] = B_{pk}[i]$
$B_r =$ *SUMPRODUCT*(B_1, B_2)	Arguments $\{B_r, B_1, B_2\}$ are all *numeric*; *length*(B_1) = *length*$(B_2) \geq 2$; and *rows*(B_r) = *columns*$(B_r) = 1$	$B_r[i] = \sum_{i=1}^{length(B_1)} B_1[i] \times B_2[i]$.

a) Redundancies: Depending on the application, some constraints in the solution to the tabular constraint learning problem may be considered *redundant*. This is because constraints may be logically equivalent or may be implied by other constraints.

TIAS: This should contain examples that recur later in the text

For example, for some constraints, the order of some of the arguments may not matter. For example for the product constraint, $B_r = B_1 \times B_2 \equiv B_r = B_2 \times B_1$ so one can be considered redundant to the other.

SAMUEL: Should this not be part of approach / optimizations / redundancy? One way to deal with such equivalences is to identify a *canonical* constraint among the equivalent ones. For example, for the product constraint above, we could define a lexicographic ordering over the blocks and only consider the constraint with the smallest ordering.

Because dealing with redundancy is often application-dependent, we explain in the next section our generic method for finding all constraints.

III. Approach to Tabular Constraint Learning

The aim of our method is to detect constraints between rows and columns of tabular data. Recall that a valid argument assignment for a constraint is an assignment of each argument of the constraint to a block, such that the constraint, that is, its signature and definition, is satisfied. SAMUEL: In general: should it not be: assign blocks to arguments instead of arguments to blocks? TIAS: I here use constraint and constraint template interchangeably

Our proposed methodology contains the following steps:

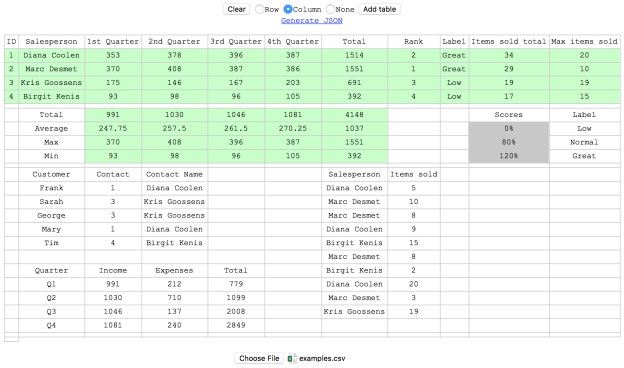
1) Extract headerless tables from tabular data
2) Partition the tables into maximally contiguous, type-consistent *superblocks*
3) Generate for each constraint all valid argument assignments in two steps:
   a) For each constraint c, generate all atoms $c(B_1, \ldots, B_n)$ where each B_i is a maximal *superblock* and the B_i are compatible with the signature.
   b) For each generated atom $c(B_1, \ldots, B_n)$, find all valid $c(B'_1, \ldots, B'_n)$ such that for all i holds $B'_i \sqsubseteq B_i$ and the signature and definition of $c(B'_1, \ldots, B'_n)$ are satisfied.

The core of our method is step 3. In principle, one could use a generate-and-test approach by generating all possible blocks from the superblock and testing each combination of blocks for each of the arguments of the constraints. However, this ... *blows up* ...

Hence, we divide this step into two parts: in the first part, we will not reason over individual blocks, but rather over all possible combinations of superblocks as candidates for the arguments. TIAS: Example To develop in more detail and clarity: Consider the sum constraints, the row-blocks of T1 are ... and column-blocks are ... instead of considering all possible blocks in each super-block, we reason at level of superblocks, for example block (with ID) can not be in a row-sum and neither can (block with Items sold total, Max items sold) because its size (1 and 2) is smaller than length of blocks (4). The only possible combination of superblocks is (1st Quarter .. Rank) for both arguments.

In the second part, we start from such a superblock combination and generate and test all possible block assignments to arguments. As we only have to consider the blocks contained in each superblock, this is typically much easier. For the above example, TIAS: to develop with more clarity each of the vectors in the block will be considered as candidate for the left-hand side, and one could enumerate all subblocks for the right-hand side and verify for each of the rows in the vector. In practice, this can be optimized further.

We now describe in more detail how the headerless tables are extracted and how the superblocks are generated from them (step 1 and 2, Section 4.1), how the candidate superassignments are generated (step 3a, Section 4.2) and how the actual assignments are extracted from that (step 3b, Section 4.3). We then describe a number of optimizations that we perform to speed up step 3 and 4 (Section 4.4).



ID	Salesperson	1st Quarter	2nd Quarter	3rd Quarter	4th Quarter	Total	Rank	Label	Items sold total	Max items sold
1	Diana Coolen	353	378	396	387	1514	2	Great	34	20
2	Marc Desmet	370	408	387	386	1551	1	Great	29	10
3	Kris Goossens	175	146	167	203	691	3	Low	19	19
4	Birgit Kenis	93	98	96	105	392	4	Low	17	15

	Total	Average	Max	Min
	991	247.75	370	93
	1030	257.5	408	98
	1046	261.5	396	96
	1081	270.25	387	105
	4148	1037	1551	392

Scores	Label
0%	Low
80%	Normal
120%	Great

Customer	Contact	Contact Name
Frank	1	Diana Coolen
Sarah	3	Kris Goossens
George	3	Kris Goossens
Mary	1	Diana Coolen
Tim	4	Birgit Kenis

Salesperson	Items sold
Diana Coolen	5
Marc Desmet	10
Marc Desmet	8
Diana Coolen	9
Birgit Kenis	15
Marc Desmet	8
Birgit Kenis	2
Diana Coolen	20
Marc Desmet	3
Kris Goossens	19

Quarter	Income	Expenses	Total
Q1	991	212	779
Q2	1030	710	1099
Q3	1046	137	2008
Q4	1081	240	2849

Fig. 2: This figure shows the table selection tool of *TaCLe*. A user selects rectangular ranges of cells that correspond to (headerless) tables. If the data has an unambiguous orientation, this information can be provided as well.

A. Table extraction

Many spreadsheets contain headers that provide hints at where the table(s) in the sheet are and what the meaning of the rows and columns is. However, detecting tables and headers is a complex problem on its own [4]. Furthermore, headers may be missing and their use often requires incorporating language-specific vocabulary, e.g. in English.

Our algorithm requires tables to be specified, this can, however, be done in various ways. Tables are specified in a JSON configuration file which contains the table names, their coordinates within a CSV file and optionally the orientation of certain tables. The orientation can be used to indicate that a table should be interpreted as only rows or only columns.

The generation of this JSON file can be done in many ways, ranging from manually specifying the tables to automatically detecting tables. Requiring the user to manually generate these files decreases the usability of our tool in a standalone setting, therefore, we offer two proof of concept approaches that aim to facilitate the specification of tables within a CSV.

too technical we store for each table:...

1) Automatic detection: As mentioned above, automatic detection of headers and tables is a complex task. Therefore, we define a restricted setting in which the task becomes easy enough to be automated. We assume that 1) tables are rectangular blocks of data not containing any empty cells; and 2) tables are separated by at least one empty cell from other tables. Additionally, if the first row contains only textual values it treated as a header, otherwise, if the first column consists of only textual data is will be the header. If a header row (column) is detected, it is removed from the table and the orientation of the table is fixed to columns (rows), otherwise, we assume there is no header and the orientation is not fixed.

The tool will now detect ranges of non-empty cells in every row and greedily merge these ranges across columns to obtain rectangular tables. If the first row of a table contains This tool removes the need for human intervention, however, it is rather limited and can only be used in simple settings.

2) Visual: Since the specification of tables is usually easy for humans, we propose a second approach which aims to decrease the complexity of specifying tables by allowing users to indicate tables using a visual tool (Figure 2). Users

select tables excluding the header and optionally specify an orientation. The tool then generates the table definitions automatically.

Aside from extending the automatic detection to be more applicable, one could combine the visual and automatic detection to let the user approve or correct detected table.

B. Block detection

The goal of the block detection step is to partition tables into type-consistent (all-numeric or all-textual) blocks. We generate blocks automatically, partitioning tables into maximal type-consistent blocks. First, the spreadsheet data is preprocessed so that currency values and percentual values are transformed into their corresponding numeric (i.e. integer or floating point) representation (e.g. 2.75$ as 2.75 or 85% as 0.85). Afterwards, tables are partitioned into maximal type-consistent blocks. If a table has no fixed orientation, the algorithm will attempt to construct both row-blocks and column-blocks. In order to find row-blocks (column-blocks), the type of each row (column) is determined, i.e. whether it is numeric or textual. All adjacent rows (columns) that have the same type are merged to form a block. If any row (column) of the table contains both numeric and textual values, it is type inconsistent and the algorithm will not produce any row-blocks (column-blocks) for that table. For tables that have a fixed orientation only blocks of that orientation are built.

While this step can be done algorithmically, there are cases where specifying finer-grained blocks manually may be beneficial. Firstly, blocks may group data from different contexts. For example, consider the type consistent block $B_3 = T_1[:, 3{:}8]$ from the example in Figure 1a; $T_1[:, 8]$ is contextually different from $T_1[:, 3{:}7]$, therefore, we could split B_3 into two blocks. SAMUEL: Secondly, accidental blocks Finer grained groups can reduce the amount of redundant constraints that are found.

C. Algorithm

Our method assumes that tables and blocks are given and solves the Tabular Constraint Learning problem by checking for each template: what combination of given (super)blocks may satisfy the signature and which specific (sub)blocks satisfy both the signature and the definition.

The main challenge we aim to address with our approach is to avoid having to check each possible combination of blocks for each constraint template as well as developing an *extensible* method in which new constraint templates can be plugged in with relative ease. To answer this challenge we separate checking the signature (concerning properties of a block) of a constraint template from checking the definition (concerning the values in a block) of a constraint template. Furthermore, we use constraint satisfaction technology to jointly search for and test the signature over all combinations of superblocks.

The pseudo-code of the main method is shown in Algorithm 1. For every constraint template s, the algorithm first generates all combinations of superblocks (i.e. superblock assignments) that are consistent with the signature. In a second step it looks at every superblock assignment and finds subassignments that satisfy the signature and definition of s.

Algorithm 1 Learn tabular constraints

Input: $\mathcal{B}$ – blocks, S – constraint templates
Output: C – learned constraints
procedure LEARNCONSTRAINTS($\mathcal{B}$, S)
  $C \leftarrow \emptyset$
  for all s **in** S **do**
    $n \leftarrow$ number of arguments of template s
    $A \leftarrow$ *SuperblockAssignments*$(s, \mathcal{B})$
    for all $(B_1, \ldots, B_n) \in A$ **do**
      $A' \leftarrow$ *Subassignments*$(s, (B_1, \ldots, B_n))$
      for all $(B'_1, \ldots, B'_n) \in A'$ **do**
        $C \leftarrow C \cup \{s(B'_1, \ldots, B'_n)\}$
  return C

1) Generating superblock assignments: Given a constraint template s and the set of all blocks $\mathcal{B}$, the goal is to find all combinations of blocks that satisfy the constraint signature. Let there be b blocks and let s have n arguments, then a simple generate-and-test method would have to test b^n combinations (given that blocks can be repeated), which can grow very rapidly with b, depending on n.

However, the properties defined in the signature, such as those related to the type or size of a block, can be used to avoid considering certain groups for certain arguments. Furthermore, the choice of one argument can influence the possible candidates for the other arguments, for example, if they must have the same length. Instead of writing specialized code for each of these cases, we make use of the built-in reasoning mechanisms of constraint satisfaction solvers.

A Constraint Satisfaction Problem (CSP) is a triple (V, D, C) where V is a set of *variables*, D the domain of possible values each variable can take and C the set of constraints over V that must be satisfied. In our case, we define one variable V_i for each argument of a constraint template. Each variable can be assigned to each of the blocks in $\mathcal{B}$, so the domain consists of $|\{\mathcal{B}\}|$ identifiers.

To reason over the blocks, we automatically add to the constraint program a set of background facts that contain the properties of each block, namely its type, size, number of columns and rows, length, orientation, and table. The actual constraints correspond to the requirements defined in the signature, such as whether arguments are of a certain type, have certain dependencies or have to be equal length or orientation, etc.

In order for our approach to be complete, the signature constraints can not always be enforced directly on a block. If a block matches the signature it is a valid candidate, however, if it does not there might exist subblocks that still satisfy the signature.

Example 3. Consider, for example, the constraint template *SERIES*(x) and a superblock candidate B for x containing both integer and floating point values. While the signature of the template requires x to be integer we cannot discard B since there might be subblocks $B' \sqsubseteq B$ that only contain integer values and satisfy the signature.

Table II shows the how constraints from the signature are converted to constraints on superblocks. Requirements on the lengths, orientations and tables of blocks can be directly enforced since they are invariant under block containment($\sqsubseteq$).

TABLE II: Translation of signature requirements to superblock constraints

Constraint	Superblock constraint
$length(B) = x$	$length(B) = x$
$orientation(B) = x$	$orientation(B) = x$
$table(B) = x$	$table(B) = x$
$type(B) = x$	$supertype(type(B)) = supertype(x)$
$size(B) \geq x$	$size(B) \geq x$
$size(B) = x$	$size(B) \geq x$
$columns(B) = x$	*if* $orientation(B) = column : columns(B) \geq x$
	if $orientation(B) = row : columns(B) = x$
$rows(B) = x$	*if* $orientation(B) = column : rows(B) = x$
	if $orientation(B) = row : rows(B) \geq x$

Typing requirements need to be relaxed to check only for supertypes (numeric and textual). Minimum sizes can be directly enforced, but exact size requirements are relaxed to minimum sizes, since subgroups might contains fewer vectors. Finally, restrictions on the number of rows or columns behave as length or size constraints based on the orientation of the group they are applied to. Subgroups of row-oriented (column-oriented) group will always have the same number of columns (rows), however, the number of rows (columns) might decrease.

The *SuperblockAssignments*(s,$\mathcal{B}$) method will use these conversion rules to construct a CSP and query a solver for all satisfying solutions. These solutions will correspond with the valid superblock assignments for constraint template s.

Example 4. Consider the template $x = SUMPRODUCT(y, z)$, then the generated CSP will contain three variables V_x, V_y, V_z corresponding to arguments x, y and z respectively. Given blocks $\mathcal{B}$, the domain of these variables are $D(V_x) = D(V_y) = D(V_z) = \{1, \ldots, |\mathcal{B}|\}$. Finally, the signature will be translated into constraints:

$$numeric(V_x) \wedge numeric(V_y) \wedge numeric(V_z)$$
$$(orientation(V_x) = row) \implies (columns(V_x) = 1)$$
$$(orientation(V_x) = column) \implies (rows(V_x) = 1)$$
$$length(V_y) \geq 2 \wedge length(V_z) \geq 2$$
$$length(V_y) = length(V_z)$$

Every solution is a tuple of values for V_x, V_y and V_z corresponding directly with a superblock assignment over $\mathcal{B}$.

2) Constraints over satisfying subgroups: In the previous step superblock assignments are generated for a constraint template s with n arguments. In this step, given such a superblock assignment $(B_1, \ldots, B_n)$, the goal is to discover valid *subassignments*, i.e. assignments of subblocks $(B'_1, \ldots, B'_n)$ (for all i, $B'_i \subseteq B_i$) that satisfy both the signature and the definition of template s. While the previous step reasons about the *properties* of blocks (type, size, length, ...), this step reasons about the actual *content* of the vectors in the blocks, that is, the data.

Example 5. Consider the sum-over-rows constraint template $B_r = SUM_{row}(\mathbf{B_x})$. An example superblock assignment from Figure 1a is $(B_r, \mathbf{B_x}) = (T_1[:, 3{:}8], T_1[:, 3{:}8])$. Note that this assignment does not satisfy the signature yet, B_r contains more than one vector and $size(\mathbf{B_x}) \neq length(B_r)$. This step aims to generate subassignments $(B'_r \sqsubseteq B_r, \mathbf{B_x}' \sqsubseteq \mathbf{B_x})$ that *do* satisfy the signature and test whether they satisfy the definition: $\forall i, B_r[0][i] = \sum_j \mathbf{B_x}[j][i]$, where $B[i][j]$ indicates the ith value of the jth vector in B. In Figure 1a there is exactly one such satisfying subassignment: $(T_1[:7], T_1[:, 3:6])$.

The search for subassignments is represented as the generic procedure *Subassignments*. However, the actual implementation is specified for every constraint template separately, allowing different methods to be used based on the type of template. One possible method is to reformulate the problem as a CSP, similar to the approach we employed in the last section. Often, however, a simple generate-and-test approach will suffice. In contrast to the previous step, the number of subblocks to generate here are typically manageable: $O(n * m^2)$ with m the size of the largest block and n the number of arguments of the template. We only consider continuous ranges as subblocks, so for a block of size m there are $m * (m + 1)/2$ possible subblocks to consider. Furthermore, few constraint solvers support floating point numbers, which are prevalent in spreadsheets.

For this reason, the *TaCLe* system uses generate-and-test to implement *Subassignments* for most constraint templates. Given a superblock assignment, all subassignments are generated taking into account the required size, columns or rows. The subassignments are sequentially tested using a template-specific procedure *Test* and valid subassignments are returned. The *Test* procedure verifies if the definition is satisfied and, if necessary, checks signature constraints that have not yet been enforced (e.g. subtype constraints).

Algorithm 2 Generate-and-test for *Subassignments*

```
procedure SUBASSIGNMENTS(s, (B_1, ..., B_n))
    n ← number of arguments of template s
    A_sub ← ∅
    for all (B'_1, ..., B'_n) ∈ {(B'_1, ..., B'_n) | ∀i : B'_i ⊑ B_i} do
        if disjoint({B'_1, ..., B'_n}) ∧ Test((B'_1, ..., B'_n)) then
            A_sub ← A_sub ∪ {(B'_1, ..., B'_n)}
    return A_sub
```

D. Optimizations

This section discusses various essential design decisions and optimizations aiming to improve the speed and extensibility of our system as well as to find less redundant constraints.

1) Template dependencies: As discussed in Section II-D1, some constraints depend on other constraints. This is apparent when looking at the template signatures in Table I, where one can see that some signatures depend on other constraints, such as *LOOKUP* depending on *FOREIGNKEY*. If a constraint template s_1 includes template s_2 in its signature, we say that s_1 depends on s_2.

In Inductive Logic Programming, one often exploits implications between constraints to structure the search space [15]. Our approach uses the dependencies between templates

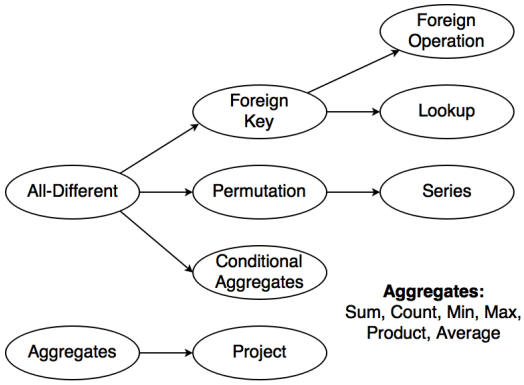


Fig. 3: Dependency graph; an arrow from s_1 to s_2 indicates that s_2 depends on s_2 (the Signature of s_2 includes s_1). The graph is extracted from the Signature column in Table I. SAMUEL: Correct aggregates

to define a dependency graph. We assume that signatures do not contain equivalences or loops, and hence the resulting graph is a directed acyclic graph (DAG). Figure 3 shows the dependency graph extracted from the signature definitions in Table I. Constraint templates that have no dependencies are omitted.

Using the dependency graph ($\mathcal{D}$) we can reorder the templates such that a template occurs after any template it depends on. Concretely, the input templates S (see Algorithm 1) are reordered to agree with the partial ordering imposed by $\mathcal{D}$.

The dependency graph represents a partial ordering and is used produce a linear ordering of the given templates such that more general templates occur before templates that depend on them (e.g. *ALLDIFFERENT* before *FOREIGNKEY*). Considering Algorithm 1, this step can be seen as preprocessing the input S. Learning in this order allows templates to use constraints they depend on as these will have already been found. This is exploited in the *SuperblockAssignments* step to speedup the algorithm by reducing the search space. Let us illustrate this step with an example.

bit too vague: modify algo at line X to do Y and ...

Example 6. Consider $FOREIGNKEY(B_{fk}, B_{pk})$ whose signature includes $ALLDIFFERENT(B_{pk})$ and assume that C are the constraints found for *ALLDIFFERENT*. Instead of generating assignments for B_{pk} using the given blocks $\mathcal{B}$, we will only consider (sub)blocks $P = \{B' | ALLDIFFERENT(B') \in C\}$. P can be seen as a set of partial assignments in which the blocks for certain arguments have been fixed.

The *SuperblockAssignments* implementation now takes into account previously found constraints to obtain partial assignments P for a template. Instead of generating one CSP to find all assignments, a CSP is generated for every $p \in P$ which finds all assignments completing p. The procedure to generate CSPs remains the same, except that the partial assignment is used to fix the domains of the dependent variables.

This approach has the additional benefit of ensuring that the (sub)blocks assigned to the dependent arguments already satisfy both the signature and definition of the original template. Therefore, the implementation of *Subassignments* does not have to take these blocks into account.

2) Constraint solvers: SAMUEL: Tias, Sergey, have a look at his section, I'm torn about rewriting it, feeling it should either be reduced or expanded, especially the second part. Generic constraint solving technology can be used for the candidate group generation as well as for enumerating the satisfying subgroups. Constraint solvers can be categorized as being stand-alone solvers, meaning that they have a text-based modeling language in which the CSP must be formulated, and *embedded*, meaning that they have a programmatic API to formulate the CSP.

maybe a bit technical reduce?

TaCLe supports the ASP [11] formalism, using the stand-alone Clingo 4.5.4 [5] solver, the MiniZinc language [12] with the stand-alone Gecode 4.4.0 solver (which supports floats) and uses an embedded Python-based CP solver [13]. The constraint signature that must be satisfied to generate the groups can be specified in each of the three formalisms, to ease prototyping and extendability towards new constraints. The embedded solver is fastest as it does not have the overhead of file-writing and calling an external program, and is used in the experiments below.

no real point here

3) Redundancy: We consider two types of redundancy that we aim to eliminate during the search. As noted in Section II-D1, for some constraint templates there are *symmetries* over the arguments that lead to trivially equivalent solutions, for example, $B'_R = B'_1 \times B'_2 \Leftrightarrow B'_R = B'_2 \times B'_1$. Such duplicates are avoided by defining a *canonical* form for these constraints. In practice, we define an arbitrary but fixed block ordering and require those blocks that are interchangeable to adhere to this ordering. SAMUEL: Mention B1*B2 vs PRODUCT?

The second type of redundancy we aim to reduce is that there may be multiple *overlapping* subblocks that satisfy the definition of a constraint template. For example, consider the constraint $B_r = SUM_{col}(B[1:n])$ where $B[k:n]$ consists of only zeros, then $B_r = SUM_{col}(B[1:j])$ will be true for all $k-1 \leq j \leq n$. In *TaCLe* we have, therefore, chosen to only consider *maximal* subblocks. In some cases the maximal subblock might falsely include irrelevant columns, in this case we suggest to put these irrelevant columns into a separate superblock.

no, it just might, period... you still do it because prefer to avoid swamping user with many redundant constraints

4) Constraints: The constraints that are currently supported in our system are shown in Table I. Their inclusion was based on the analysis of the most popular Excel constraints as well as the spreadsheets we encountered. All these constraints are available in popular spreadsheet software, except for the structural constraints (*ALLDIFFERENT*, *PERMUTATION* and *FOREIGNKEY*). These constraints added so they can be used in the signature of other constraint templates; they are also popular in constraint satisfaction [1].

Note the the template $B_r = B_1 - B_2$ can be rewritten as $B_1 = B_r + B_2$. Our choice to use difference is a hardcoded preference based on the prevalence of this type of constraint in financial spreadsheet data.

bit more elaborate: - and + same, so we choose only one (same for / and *)

IV. EVALUATION

SERGEY: we need to add a summary of the dataset, avg number of constraints, cells, rows, columns In this section

we experimentally validate our approach. We studied various questions, most notably with what accuracy our algorithm can find essential constraints.

The implementation is illustrated using a case study on the spreadsheet corresponding with the previously introduced example (figure 1a). In order to quantify the results and generalize our findings we also evaluate our algorithm on a benchmark of 30 (SAMUEL: check number) spreadsheets that we assembled from various sources.

In this section we focus on *functional* constraints that could be used in spreadsheets, ignoring constraints such as all-different or foreign-key.

All experiments were run on a Macbook Pro, Intel Core i7 2.3 GHz with 16GB RAM.

A. Case study

TIAS: Move explanation of origin to first time it is introduced Let us illustrate our implementation using the example presented in Figure 1a. This example combines several smaller examples that were used in an exercise session to teach Excel into one spreadsheet. Figure 1b shows the constraints that we expect to find. strange sentence

1) Results: TaCLe takes a few seconds to find 18 constraints, including all of the 12 solutions described in Figure 1b. Of the 6 remaining (redundant) constraints, 5 are *rank* constraints that are true by accident, such as:

$$T_1[:,1] = RANK(T_1[:,5])$$

update with new 1b)

The other redundant constraint is an additional *lookup* that holds because the two vectors in $T_2[:,2{:}3]$ can both be used to look each other up in $T_1[:,1{:}2]$ and we considered only one of them to be essential (lookup using the ID).

For this example our primary goal of finding all constraints is achieved. The implementation also returns a number of redundant constraints (33.33% of the total). This ratio is, as we will show, rather high compared to other spreadsheets. However, this example contains many short vectors which increases the chance for constraints to be true by accident.

precision/recall? would be standard measures...

B. Benchmark

There are three main categories of spreadsheets in the benchmark: spreadsheets from an exercise session teaching Excel at an affiliated University, spreadsheets from tutorials online and publicly available spreadsheets such as crime statistics or financial reports that demonstrate more real world usage[1]. The case study is also included.

All spreadsheets have been converted to CSV with no manual intervention unless noted. Definitions of tables were added manually, providing a way to compare results and overcoming shortcomings in the table detection algorithm (in some harder cases groups where also provided manually).

For every spreadsheet a ground-truth has been provided manually, specifying the essential (or original) constraints that are expected to be discovered. We consider three types of constraints:

present in the data

[1](attached to to the submitted paper)

which and why...

right? like, from the original spreadsheet...

Implemented Constraint that have been implemented and are expected to be found (all constraints in Table I)

Essential Constraint that have or have not been implemented but could be found using our algorithm (e.g. fuzzy conditional sum)

Non-trivial Constraints currently outside of the scope of this system (e.g. generic nested mathematical or logical formulas or n-ary constraints)

we should discuss limitations earlier too...

We examine three main questions concerning the accuracy, redundancy and efficiency of our approach, our main focus being accuracy.

precision recall...

Q1. How accurate is the approach?: Fortunately, our system is currently able to find 100% of the implemented constraints (90) on the benchmark suite. There are only three essential constraints that have not been implemented yet, therefore our system currently identifies 96.77% of the essential constraints.

not clear how!

Q2. How many redundant constraints are discovered?: Our primary focus was, as mentioned before, accuracy, therefore we sometimes traded off less redundancy for more accuracy. The motivation is that solutions can still be pruned using entailment or heuristics afterwards. The constraints we considered to be redundant are either duplications (results that can be calculated in different ways, one of which seems superior) or constraints that hold by accident. =spurious

Across all spreadsheets our implementation finds 121 constraints, 28 (23.14%) of which are redundant. However, the average redundancy per spreadsheet is only 8.33%. Examining the redundant constraints reveals that many (12) of them are duplications occurring in a single spreadsheet. Many of the remaining constraints are duplications stemming from the overlapping role of difference and sum. Accidental constraints are limited to the 5 rank constraints that were discussed in the case study.

but included only one?

Redundant constraints can be reduced in different ways. Duplications should be detected in a post-processing step that uses entailment or heuristics to filter constraints. Accidental constraints may be detected through heuristics, however, they can be also be removed by adding additional data.

Q3. How fast is the algorithm?: Concerning the speed of the algorithm we also prioritized accuracy when a trade-off had to be made. For the 32 spreadsheets in the benchmark our implementation ran in $16.12s \pm 0.62s$. The execution times vary widely though between spreadsheets, only four spreadsheets taking more than $0.2s$. Most of the execution time in these cases goes towards searching either aggregate constraints or conditional aggregates. The search for aggregates will be slow on spreadsheets containing larger groups of numeric data. For conditional aggregates the number of candidate primary keys (all-different) and numeric vectors determines the running time (e.g. the case study example).

SAMUEL: Expand summary SERGEY: would it make sense to specify the size of a spreadsheet in the number of non-zero cells?

a) Dependencies: In order for the algorithm to run efficiently it is crucial to use dependencies and find constraint incrementally whenever possible. This avoids some of the

show Q1/2/3 per 'category' of spreadsheets

	Total	Average per spreadsheet
Constraints	93	2.91
Accuracy	96.77%	94.27%
Redundant	23.14% (28)	8.33% (0.88)
Speed (s)	$16.12s \pm 0.62s$	$0.50s \pm 0.02s$

TABLE III: Overview evaluation on essential constraints of the collected Spreadsheet dataset

explosion of combinations for constraints that have many arguments. For example, using foreign keys as a base constraint to find conditional aggregates reduces the running time for the case study from about 3 seconds to below 1 second. Unfortunately, this assumption is sometimes too strong, when users are interested in aggregates for only some of the keys that are present in the data.

V. Applications

In this section, we illustrate how various motivating applications could be realized using our system.

A. Autocompletion

SERGEY: I think the time discussion is irrelevant, should it be cut out completely?

Autocompletion can be seen as using knowledge derived from data at time t_0 and new input data at time $t_1 > t_0$ to predict data that the user has not yet written. Therefore, constraints are learned at time t_0 and combined with data from a new row i (column j) that the user has added to a table T by time t_1. Constraints that do not conflict with the added data are considered *viable*. Viable constraints that have enough input data to compute the result for a blank cell $T[i, j]$ in the new row (column) are *active*, which means they can be used to predict the outcome for that cell. *Active* constraints can be used to autocomplete blank cells such that the user does not have to fill in the values manually. Once the user has stopped editing the new row i (column j), the table T is updated to include i (j) and constraints are learned once again.

Let us illustrate this on one of the spreadsheets from the gathered dataset. In Figure 4b, we see two tables the upper one is completed and in the lower one values in the last raw are not yet filled, then the system finds a satisfying constraint (*SUMIF*) and based on it proposes filling values.

SERGEY: there should be a discussion on what to do if multiple constraints satisfy the data and propose predictions, just a sketch here for now If multiple constraints satisfy the data and allow autocompletion, then there is a number of ways to handle it. Firstly, autocompletion can be disabled. Secondly, the most specific constrain can be applied. Thirdly, if one is not more specific than the other, one needs to introduce an autocompletion order, which would resolve the problem by picking only one constraint to suggest the values.

B. Error detection

SERGEY: Either we need to elaborate on the online detection or remove offline-online and simplify the rest, otherwise it is not clear what is going on; again I don't see any point in time here

We consider two cases of error detection. The first setting is the *online* detection of errors which is similar to the autocompletion setting. However, instead of considering viable constraints, we look at *conflict* constraints, i.e. constraints that were learned at time t_0 but do not agree with the newly added data at time t_1. Such conflict constraints can indicate errors in the input. Since conflict constraints could also be accidental constraints that are correctly invalidated by the new data, the size of the data at time t_0 should be large enough.

The second type of error detection is *offline* and attempts to detect errors in a spreadsheet. Consider, for example, a use case concerning crime statistics provided by the FBI. An extract of the spreadsheet is depicted in Figure 4a. When run on this sheet, our system is able to detect the second sum constraint, however, a missing value prevents the first sum constraint from being learned. A possible approach to detect such missing or wrong values is to look at a sample of the table that excludes the erroneous row and compare the constraints learned on this sample with the constraints learned by adding the row containing the mistake. The sample needs to large enough such that we have enough confidence that the constraints are correct. For example, in Figure 4a all but one row satisfy the constraint, which is a strong indication that there is a constraint violation. If we define the sample to be our data at time t_0 and the added row to be the additional input at t_1, this task can be formulated as online error detection and dealt with in the same way.

VI. Related Work

TaCLe combines ideas from several existing approaches from multiple different fields of research.

First, it borrows techniques from logical and relational learning [15], as it finds constraints in multiple relations (or tables). However, unlike typical logical and relational learning approaches, it focuses on data in spreadsheets and on constraints that can be column or row-wise. Furthermore, it derives a set of simple atomic constraints rather than a set of rules that each consist of multiple literals as in clausal discovery [16?]. Still several ideas from logical and relational learning proved to be useful, such as the use of a canonical form, a refinement graph, and pruning for redundancy. Also related to this line of research is the work on deriving constraints in databases such as functional and multi-valued dependencies [17?] although this line of research has focused on more specialized techniques for specific types of constraints. Furthermore, the constraints used in spreadsheets are often quite different than those in databases.

Second, there exist several algorithms in the constraint programming community that induce constraint (programs) from one or more examples and questions. Two well-known lines of research include ModelSeeker [1] and Quacq [2]. The former starts from a single example in the form of a vector of values and then exhaustively looks for the most specific constraints from a large constraint library that hold in the vector. To this aim, it cleverly enumerates different

no, that is a design decision, present as such: "spurious constraints may lead to wrongly detected conflicts. One could add a threshold on the size of vects so only if data large enough"

good, but not referenced to

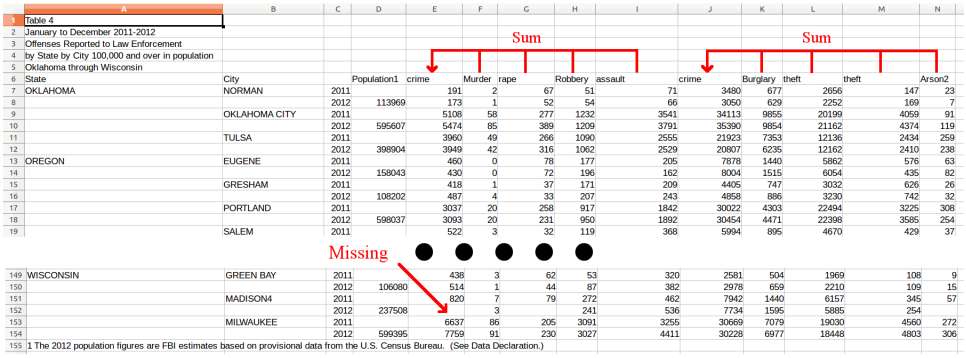
so not by default?

add exp 'without dependency' and with?

snapshots

too vague: sample the data? try leaving out each row? make a concrete proposal

what we discussed sounded cleaner: learn constraints on snapshot; keep track of what vectors (of blocks) change, if for a functional cons, all its right-hand side vectors changed, then evaluate it to see whether left-hand side can be computed (or is satisfied, for error correction?)



(a) Real world tabular constraint reconstruction: FBI crime statistics

Date	Region	Units	Total Amt.
24/06/2012	North	186	$50,592.00
01/06/12	East	356	$96,832.00
09/09/12	West	907	$246,704.00
26/06/2012	South	190	$51,680.00
22/04/2012	North	717	$195,024.00
22/03/2012	West	550	$149,600.00
19/12/2011	East	942	$256,224.00
31/10/2011	North	901	$245,072.00
02/10/11	West	117	$31,824.00

Region	Units per region	Amt. per region
North	1804	490688
East	1298	353056
South	190	51680
West		

(b) SERGEY: Samuel, Tias have a look at it, should be corrected and extended, like with groups indications and formulas?

which is all over integers, we also over strings and real-valued, AND, not one vector but we have many indep. blocks

tables from the vector. For instance, if the initial vector was of dimension 1×20, it would consider rearrangements of size 2×10, 4×5, 5×4, and Modelseeker would then look for both column, row and even diagonal constraints. Key differences with our technique are that we assume the tables are given (a reasonable assumption if one starts from a spreadsheet), and also that ModelSeekers constraints are the typical ones from the constraint programming community, which are aimed at combinatorial optimization and scheduling rather than what is typically used in spreadsheets. Quacq [2] and its predecessor Conacq [3] acquire a conjunction of constraints using techniques inspired on Mitchells version spaces to process examples, some variants also asking informative questions to the user. As the conjunction for typical constraint programming problems (such as Sudoku) can be quite large a key problem is to cope with redundant constraints. As Modelseeker, these systems focus on the types of constraints in combinatorial optimization but unlike ModelSeeker and our approach, they do not assume the variables are organized in tabular form.

Third, our work borrows from the seminal work of Gulwani et al. [7] on program synthesis for tabular data that is incorporated in Microsofts Excel. In FlashFill, the end-user might work on a table containing the surnames (column A) and names (column B) of politicians. The first row might contain the values A1 = "Blair" and B1 = "Tony", and the user might then enter in cell C1 the value "T.B.". At that point FlashFill would generate a program that produces the output "T.B." for the inputs "Blair" and "Tony", would also apply it to automatically generate values for the other politicians (rows) in the table. While it might need an extra example to deal with names such as "Van Rompuy, Herman" and determine whether it should output "H.V." or "H.V.R.", FlashFill learns correct programs from very few examples. Flashfill has been extended to e.g. FlashExtract [10] to produce a set of normalized database tables starting from naively filled out spreadsheets. However, unlike our approach, Flashfill requires the user to identify explicitly the desired inputs/outputs from the function as well as to provide explicitly some positive and sometimes also negative examples. In contrast, our approach is unsupervised, although it would be interesting to study whether it could be improved through user interaction and through user examples. Another interesting open issue (for all of these techniques with the exception of logical and relational learning) is how to deal with possibly incomplete or noisy data.

Finally, there are the works on BayesDB [**?**] and Tabular [6], two recent probabilistic modeling languages that have been specifically designed for dealing with relational data in tabular form and that can, as FlashFill, also automatically fill out missing values in a table. However, unlike the other mentioned approaches, it is based on an underlying probabilistic graphical model that performs probabilistic inference rather than identify hard constraints.

it looks only at pairwise constraints, so an important factor is how fast the algorithm can converget to the entire constraint set. Redundant and spur also issue for these metho

VII. Conclusions

Our goal is to automatically identify constraints in a spreadsheet. We have presented and evaluated our approach, implemented as the system *TaCLe*, that is able to learn a large set of constraints from CSV data. Moreover, the amount of redundant constraints found by the system is limited despite the lack of more sophisticated post-processing steps.

a) Future work: There are multiple directions for future work on this topic. Integrating the system in an interactive setting would offer users the possibility to easily receive and provide feedback. Moreover, the system can be adapted to deal with additional types produce less redundant constraints through the use of (heuristic) filtering or post-processing.

Currently, the system only learns single constraints, however, extending our approach to nested constraints would allow more expressive concepts to be learned. This shift would bring the approach in line with programming by example.

Aside from finding errors, the system can be extended to deal with noise and the ability to learn soft constraints. Soft constraints are the (potentially weighted) constraints that hold only on some of the data. This would extend the approach to new application domains as well as provide more native error detection.

only strings?

write limitations: single cons, complete rows/cols only, (what else?) and deal with that as futurework...

References

[1] Nicolas Beldiceanu and Helmut Simonis. *Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, chapter A Model Seeker:

can be more precise, does it add formulae?
or this is real DB database, e.g a schema and PK/FK constraints?

Extracting Global Constraint Models from Positive Examples, pages 141–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013.

[3] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. A sat-based version space algorithm for acquiring constraint satisfaction problems. In *Machine Learning: ECML 2005, 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005, Proceedings*, pages 23–34, 2005.

[4] Jing Fang, Prasenjit Mitra, Zhi Tang, and C. Lee Giles. Table header detection and classification. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada.*, 2012.

[5] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume arXiv:1405.3694v1, 2014. Theory and Practice of Logic Programming, Online Supplement.

[6] Andrew D. Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. Tabular: A schema-driven probabilistic programming language. Technical Report MSR-TR-2013-118, December 2013.

[7] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[8] Thomas Herndon, Michael Ash, and Robert Pollin. Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. *Cambridge Journal of Economics*, 2013.

[9] Arnaud Lallouet, Matthieu Lopez, Lionel Martin, and Christel Vrain. On learning constraint problems. *Proceedings - International Conference on Tools with Artificial Intelligence, ICTAI*, 1(section II):45–52, 2010.

[10] Vu Le and Sumit Gulwani. Flashextract: a framework for data extraction by examples. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 55, 2014.

[11] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597, 2008.

[12] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. *Principles and Practice of Constraint Programming – CP 2007: 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007. Proceedings*, chapter MiniZinc: Towards a Standard CP Modelling Language, pages 529–543. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[13] Gustavo Niemeyer. https://labix.org/python-constraint.

[14] Oleksandr Polozov and Sumit Gulwani. Flashmeta: a framework for inductive program synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 107–126, 2015.

[15] Luc De Raedt. *Logical and Relational Learning: From ILP to MRDM (Cognitive Technologies)*. Springer-Verlag New York, Inc., 2008.

[16] Luc De Raedt and Luc Dehaspe. Clausal discovery. *Machine Learning*, 26(2-3):99–146, 1997.

[17] Iztok Savnik and Peter A. Flach. Discovery of multivalued dependencies from relations. *Intell. Data Anal.*, 4(3-4):195–211, 2000.

[18] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *SIGOPS Oper. Syst. Rev.*, 40(5):404–415, October 2006.