FlashRelate: Extracting Relational Data from Semi-structured Spreadsheets Using Examples

Daniel W. Barowy  
University of Massachusetts Amherst, USA  
dbarowy@cs.umass.edu

Sumit Gulwani  Ted Hart  Benjamin Zorn  
Microsoft Research, USA  
{sumitg, ted.hart, ben.zorn}@microsoft.com

Abstract

With hundreds of millions of users, spreadsheets are one of the most important end-user applications. Spreadsheets are easy to use and allow users great flexibility in storing data. This flexibility comes at a price: users often treat spreadsheets as a poor man’s database, leading to creative solutions for storing high-dimensional data. The trouble arises when users need to answer queries with their data. Data manipulation tools make strong assumptions about data layouts and cannot read these ad-hoc databases. Converting data into the appropriate layout requires programming skills or a major investment in manual reformatting. The effect is that a vast amount of real-world data is “locked-in” to a proliferation of one-off formats.

We introduce FLASRELATE, a synthesis engine that lets ordinary users extract structured relational data from spreadsheets without programming. Instead, users extract data by supplying examples of output relational tuples. FLASRELATE uses these examples to synthesize a program in FLARE. FLARE is a novel extraction language that extends regular expressions with geometric constructs. An interactive user interface on top of FLASRELATE lets end users extract data by point-and-click. We demonstrate that correct FLARE programs can be synthesized in seconds from a small set of examples for 43 real-world scenarios. Finally, our case study demonstrates FLASRELATE’s usefulness addressing the widespread problem of data trapped in corporate and government formats.

Categories and Subject Descriptors  I.2.2 [Automatic Programming]; Program Synthesis; H.2.3 [Languages]: Data manipulation languages; H.4.1 [Office Automation]: Spreadsheets

Keywords  spreadsheets, relational data, data extraction, program synthesis

1. Introduction

There are an estimated 500 million Microsoft Excel users worldwide [26]. While many of these users use Excel to perform computation, a significant fraction use Excel solely as a simple form of data storage. A study of 5,606 Excel documents scraped from the web found that nearly 30% of the spreadsheets contained no formulas at all [10]. Furthermore, CSV-formatted files, which are essentially spreadsheets, are a widely-used format. For example, the U.S. Census Bureau distributes official government data in CSV form. It is thus not surprising that a tremendous amount of important data is stored in spreadsheets and spreadsheet-like formats.

Spreadsheets combine their data model and view. This combination gives spreadsheet creators a large degree of freedom when laying out their data. Although spreadsheets are tabular and ostensibly two-dimensional tables, end users may store any high-dimensional data in a spreadsheet as long as they can devise a spatial layout (using, e.g., headers, whitespace, and relative positioning) that projects that data into two dimensions. Thus while spreadsheets allow compact and intuitive visual representations of data, they are well suited for human understanding, their flexibility complicates the use of powerful data-manipulation tools (e.g., relational queries) that expect data in a certain form. We call these spreadsheets semi-structured because their data is in a regular format that is nonetheless inaccessible to data-processing tools. Unless semi-structured data can be decoded into the appropriate normal form expected by these tools, data is effectively “locked-in” to the user’s format.

This lock-in problem is widespread. Research suggests that few spreadsheets are trivially convertible to database relations. Only 22% of 200 randomly-chosen spreadsheets scraped from the web can be converted by performing “Export as CSV.” [5] For a user with data trapped in one of these formats, little hope is available in the form of off-the-shelf tools. Our experience reading Excel help forums suggests that many users do not realize their mistake until they have invested significant time entering and organizing their data.

The fact that non-experts are unlikely to embrace good data management practices combined with a need to use data computationally often motivates the development of domain-specific data extraction software: Perl and Awk are widely used because of their powerful capabilities for processing text files; XQuery, HTQL, XSLT, and SXPath are used to extract data from webpages. Recent research such as PADS [9, 30] and FlashExtract [19] aim to make text and...
web extraction technologies easier to use for end users. There are no such tools for extracting data from spreadsheets.

Contributions. We make the following contributions:

- We present FLARE, a domain-specific extraction language for spreadsheets that describes the domain of ad-hoc layouts. The FLARE interpreter takes a FLARE program and a spreadsheet as input and returns a set of tuples (a relational table) as output. FLARE is the first pattern-based spreadsheet query language explicitly designed to extract relational data from spreadsheets (see §5).
- We present FLASHRELATE, an algorithm that generates FLARE programs given a spreadsheet and a small set of positive/negative example tuples from the intended relational table. FLASHRELATE lets users extract data without programming (see §3).
- We empirically evaluate FLARE’s expressiveness and FLASHRELATE’s efficiency and run-time against a set of 43 real-world spreadsheets drawn from a research corpus and from Excel help forums. We show that the FLASHRELATE algorithm is able to synthesize correct FLARE programs from a small number of examples, typically in under 2 seconds (see §5).
- We present a novel interactive spreadsheet environment that takes a set of example tuples as input and produces a relational table as output. We describe how its design was influenced by a case study (see §5.4).

Example. To make the problem concrete, we refer to the running example in Fig. 1, a real spreadsheet taken from the EUSES corpus [10]. The spreadsheet shows timber harvests by country and year. It packs data about a particular country into value/year pairs within a row. The author probably structured the data this way to avoid data in a long thin column, which is harder to read.

While this representation is visually convenient, consider what needs to be done for a query as simple as computing the average harvest in 1950. The data layout poses a challenge because Excel’s `AVERAGE` function expects a reference to a spatially-contiguous range of values. By contrast, `AVERAGE` can trivially compute the result from the relational table shown in Fig. 2 because the values are always located in the same column. One solution is to convert the `AVERAGE` expression to union individual cells into a range (e.g., B2, D4, C6, ...), but this manual process is tedious and error-prone even for small spreadsheets. The other solution is to manually reformat the data. The user can then utilize Excel’s point-and-click `Sort & Filter` and `Subtotals` wizards to compute the result.

Studies examining Excel help forums suggest that users often ask for help reformattting their data in this situation [13]. Help is typically offered in the form of Visual Basic programs that perform the reformatting for them. Writing even simple Visual Basic programs is well beyond the capabilities of most Excel users.

Despite the difficulty of computing the result from the original structure, a typical person would have no trouble understanding the spreadsheet in Fig. 1. Numerous geometric cues guide a person’s eye to the right location. For example, timber harvest values are always located under the heading titled `value`. Year values are always located to the right of the paired timber value.

The key insight in this paper is that geometric constraints like direction and distance concisely express the spatial relationships underpinning the desired table extraction. The FLARE language can describe such geometric constraints, and the FLASHRELATE algorithm can automatically synthesize such FLARE programs from few examples of relational tuples.
1) \[ \text{Flatten}(\text{Map}(\lambda \gamma, \text{Map}(\lambda \gamma, \text{ToTuple}(I, \tau), \text{[<propl>](I, \gamma)), \text{AllCells}(I))) \]

2) \[ \text{[<pair>](I, \gamma)} = \{\text{[<spatial>]<cell>\[<spese]\]}(I, \gamma) \]

3) \[ \text{[<speseq>](I, \gamma)} = \text{Map}(\lambda \alpha, \text{Map}(\lambda \alpha, \text{[<speseq>]}(I, \text{snd}(\phi)), \text{Eval}(s, c, I, \gamma))) \]

4) \[ \text{where } s = \text{[<spatial>], and } c = \text{[<cell>]} \]

5) \[ \text{[<regx>](I, \gamma)} = \text{Map}(\lambda \alpha, \text{Map}(\lambda \alpha, \text{[<speseq>]}(I, \gamma), \text{Eval}(s, c, I, \gamma))) \]

6) \[ \text{where } s = \text{[<spatial>], and } c = \text{[<cell>]} \]

7) \[ \text{true iff regular expression } \sigma \text{ matches String } s. \]

8) \[ \text{EvalNamed}(\gamma, \text{[<speseq>](I, \gamma)} = \text{true iff regular expression } \sigma \text{ matches String } s. \]

9) \[ \text{evaluated as if it is an } \text{[<pair>]} \text{ with the column name } \gamma. \text{ “non-captured” columns named } \gamma \text{ are removed by ToTuple.} \]

2. Flare Language

The Flare interpreter takes a Flare program and a spreadsheet as input and returns a set of tuples (a relational table) as output.

The design of Flare is inspired by scripting languages with regular expression capabilities. While regular expressions provide powerful mechanisms for specifying classes of strings, they require additional general-purpose support code in order to capture relational information encoded in spreadsheets. Flare augments regular expressions with geometric constraints needed to extract this relational data without support code. Flare constructively enumerates all possible combinations of cells from the input spreadsheet that satisfy the set of constraints described by the program.

Flare has two kinds of constraints. 1) A cell constraint defines the class of valid strings for a column of an output table. 2) A spatial constraint defines the class of spatial relationships between two columns of an output table. Taken together, these two types of constraints form a constraint graph, specifically a tree. Output columns represent vertices in this graph, and constraint pairs represent the edges. Intuitively, one can think of this graph as a structure that, when it is geometrically translated over a spreadsheet, indicates which combination of cells form relational tuples.

The syntax for Flare is shown in Fig. 5 a simple abstract spreadsheet model is shown in Fig. 6 and the formal semantics are shown in Fig. 7. We discuss the language informally here to give the reader an intuitive sense for Flare. We describe Flare’s computational complexity in [4].

Cell Constraints. In Flare, cell constraints \( \text{[<cell>] } \) in Fig. 5 are based on regular expressions. Regular expressions are enclosed by a pair of slashes, /. For example, the expression \( /[0-9]+$/ matches a string that contains only numbers. From here on, we refer to this expression as \( \text{num} \).

\footnote{Our prototype implementation uses PCRE-like regular expressions.}
Following the same exercise, we can produce a program that extracts year entries. One such program is \(<\text{Date}, -u>: u+/year/\). We discuss the semantics of anchors in more detail below. Recall that we are attempting to produce a two column relational table containing all value, year pairs in order to compute the average timber harvest for 1950. To produce the 2-tuples needed to perform our task, we need a way to combine the above two extraction programs.

**Spatial Constraints.** Spatial constraints describe the relative spatial relationships between two output columns (\(<\text{spatial}c\) Fig. 5). Spatial constraints compose two subprograms. An \(n\)-tuple subprogram composed with an \(m\)-tuple subprogram yields an \(n + m\)-tuple subprogram.

The essential component of a spatial constraint is a geometric descriptor (\(<\text{dirdir}\) and \(<\text{hdir}\) in Fig. 5). There are four basic descriptors, \(up\), \(down\), \(left\), and \(right\), denoted by \(u\), \(d\), \(l\), and \(r\) respectively. A spatial constraint may contain up to two geometric descriptors, one for the vertical direction, and one for the horizontal direction. Spatial constraints are written as an infix operator between a pair of cell constraints. One may formally think of a spatial constraint as a boolean function that takes two cell locations as parameters and returns true iff those two cells satisfy the specified geometry in the spreadsheet. Line 11 in Fig. 7 describes the meaning of spatial constraints formally.

Spatial constraints are directed, indicating a functional relationship from the parent cell to the child cell. The child-of operator indicates a parent-child relationship between two subprograms (2nd rule for \(<\text{pro}\) Fig. 5). This operator, denoted by a pair of square brackets \([\ ]\), indicates that cells matching constraints listed inside the brackets are children of the parent cells matched by the constraint on the outside of the brackets. When separated by commas, multiple child subprograms are allowed inside the brackets. Each child defines a spatial constraint with its parent.

Which subprogram serves as the parent is not always important. In the running example, we arbitrarily decide that \(\text{Date}\) entries are relative to \(\text{Harvest}\) entries. Since \(\text{Date}\) entries are always to the right of \(\text{Harvest}\) entries, we compose the two programs with the spatial constraint and child-of brackets as in \(<\text{Harvest}, -u>: u+/value/ [r <\text{Date}, -u>].\) Note that in the combined program we omit the anchor for the \(\text{Date}\) subprogram since it is unnecessary in the combined context. This is the same program shown in Fig. 4 and it allows us to answer the user’s year-average query because it produces 2-tuples compatible with Excel’s point-and-click tools.

**Quantified and Indeterminate Geometric Descriptors.** Geometric descriptors may be appended with a constant quantity (\(<\text{quant}\) Fig. 5), for example, \(r[5]\), indicating that the child cell should be 5 cells to the right of the parent cell. In some cases, we need to express that a child cell has a spatial relationship of indeterminate distance from its parent (e.g., simply “to the right”).

To be concrete, suppose we need to obtain a listing of countries whose reported timber harvest for 1950 was greater than 2000 hectares. Clearly, we need to extract country name entries along with the value and year entries. But we have a problem: while the country name is always to the left of the value in Fig. 1, the distance is different for each value, year pair. A Kleene star signifies that a geometric descriptor may take any of a range of amounts \(<a>\) and \(<a>\). As with many regular expression implementations, the Kleene star can return a set of all matches or a single match. For example, the regular expression abc* with match-all semantics matches the set of strings \{ab, abc, abccc\}. In a single-match semantics, the user must specify which of the matches they prefer. Many regular expression implementations allow the user to select the last match or the first match, often referred to as greedy or non-greedy, respectively. With greedy single match semantics, the same regular expression matches the string abccc, and with non-greedy single match semantics, it matches the string ab.

Similarly, \(\text{FLARE}\) allows users to specify match-all, greedy, and non-greedy geometric descriptors as \(*\), \(*\), and \(*?\) respectively. In our modified example, any Kleene suffices, since there is only a single country name to the left (Fig. 4).

Match-all geometric descriptors form a one-to-many relationship between parent and child cells. When there is a one-to-one relationship between a parent and a child cell, parent and child cells are combined into a single tuple. A one-to-many relationship means that \(\text{FLARE}\) must form the cross-product of a singleton set containing the parent cell and the sets of matching child cells. When two child cells have a one-to-many relationship with a parent, they have a many-to-many relationship with each other. While \(\text{FLARE}\) is capable of expressing such layouts, we think that in practice they are rare. We correspondingly bias \(\text{FLASHRELATE}\)'s search algorithm against them (see [5.2.4 “Ranking”]).

**Anchors.** Anchors function as textual and spatial constraints, but they do not capture cells. When a child constraint is composed (via a spatial constraint) with a parent constraint employing an anchor, the matching child cells are relative to those cells matching the parent’s constraints, not relative to the cells matching the anchor. Finally, we do not allow anchors to have anchors since such constructions increase the search space of possible programs for \(\text{FLASHRELATE}\) to explore with no obvious benefit in practice.

3. **FlashRelate Synthesis Algorithm**

Programmable solutions to data extraction suffer an important limitation: they require programming knowledge. This fact puts these tools out of reach of typical end users. \(\text{FLASHRELATE}\) requires no programming knowledge.

The \(\text{FLASHRELATE}\) algorithm, shown in Fig. 8, automatically generates a \(\text{FLARE}\) program as output given a spreadsheet and a set of positive and negative examples of tuples as input. The synthesized \(\text{FLARE}\) program is guaranteed to extract all of the positive tuple examples and none of the negative tuple examples supplied by the user. We frame the problem of finding a satisfactory program as a search over all valid combinations of constraints that satisfy the positive and negative examples. Unlike prior synthesis work, \(\text{FLASHRELATE}\) does not use a version-space algebra to represent the set of candidate programs. Instead, \(\text{FLASHRELATE}\) reduces the search to the construction of a minimum-weight spanning tree.

Intuitively, a valid \(\text{FLARE}\) program is a spanning tree on a graph where each vertex represents a column in the end user’s desired output relation and each edge represents a (spatial constraint, cell constraint) pair. \(\text{FLASHRELATE}\)'s program search procedure, shown in Fig. 8, is modeled on a recursive formulation of Kruskal’s minimum-weight spanning tree algorithm [17]. By construction, all of the possible programs in the search space extract all of the positive examples. Not all of the programs exclude the negative examples. To quickly find a satisfactory program, our heuristics assign weights to edges such that the user’s intended program is likely to be a minimum-weight spanning tree. \(\text{FLASHRELATE}\) uses this information to greedily search for a satisfactory program. We describe \(\text{FLASHRELATE}\)'s complexity in [4].

3.1 Definitions

We use the following terms in Fig. 8. Let \(P\) be the set of user-provided tuples representative of the desired relational table, from this point on referred to as *positive examples*. Let \(N\) be the set of user-provided tuples representing counterexamples, from this
SYNTH($I,P,N$)
1. for each column index $i$
   2. $A_c[i] =$ Learn($I,P,i$)
3. for each pair of column names $i,j$ such that $i \neq j$
   4. $A_S[i,j] =$ LearnS($I,P,i,j$)
5. return SEARCH($\emptyset,N,A_c,A_S$)

(a) The top-level procedure. $I$ is the input spreadsheet, $P$ is the set of positive example tuples, and $N$ is the set of negative example tuples. The procedure precomputes all constraints satisfying $P$ and then calls the search routine.

LEARN($I,P,i$)
1. $A_c =$ set of predefined constraints
2. $A_c =$ $\emptyset$
3. for each constraint $\alpha \in A$
   4. if $\alpha \in P$: $\exists \alpha (I,p[i]) =$ true
   5. $A_c =$ $A_c \cup \{\text{Cell}(i,\alpha)\}$
   6. return $A_c$

(b) Learn$C$ learns cell constraints from positive examples; Cell$(i,\alpha)$ is a cell constraint constructor that takes a column name $i$ and a regular expression $\alpha$, and when indexed by a column name, yields a cell $(x,y)$.

LEARN($S(I,P,i,j)$)
1. $A_S =$ $\emptyset$
2. $V =$ LEARN$D$MOUNT($I,P,i,j$, true)
3. $H =$ LEARN$D$MOUNT($I,P,i,j$, false)
4. for each $\{v,h\}$ where $v \in V$, $h \in H$
   5. $A_S =$ $A_S \cup \{\text{Spatial}(i,j,v,h)\}$
   6. return $A_S$

(c) Learn$S$ learns spatial constraints from positive examples; Spatial$(i,j,v,h)$ is a spatial constraint constructor that takes two column names $i$ and $j$, a $<\text{vert}> v$, and a $<\text{horiz}> h$. LEARN$D$MOUNT is a function that enumerates geometric descriptors; see [3.2.4] “Pruning”.

SEARCH($C_P,N,A_c,A_s$)
1. if $|C_P| =$ NUMCOLS
2. if $\bigcup_{e \in C_P} \text{Negate}(e) \cup \text{Negate}(s) =$ N
   3. return $C_P$
4. else return FAILURE
5. else
   6. pairs =$ \{ (c,s) | c \in A_c[i], s \in A_s[i,j], s.t. C = \{V,C_P \cup \{c,s\}\} \}$
   7. pairs$' =$ $\text{RANK}\{\text{pairs}\}$
   8. $k =$ 0
9. while $k < |\text{pairs}'|$
   10. $C_P =$ SEARCH($C_P \cup \{\text{pairs}'[k]\}, N,A_c,A_S$)
11. if $C_P \neq$ FAILURE
   12. return $C_P$
13. $k =$ $k + 1$
14. return FAILURE

(d) The program search procedure. The output is a set of edges guaranteed to be a spanning tree. The routine that inserts child-of operators is omitted for brevity.

Negate($c$) $\equiv \{n \in N | \exists \gamma \in n \land \neg [c](I,\gamma)\}$
Negate($s$) $\equiv \{n \in N | \exists \gamma, \gamma' \in n \land \gamma' \notin [s](\gamma,\gamma')\}$

(e) The negative examples excluded by cell and spatial constraints, respectively.

Figure 8: FLASHERELATE’s program synthesis procedures.

point on referred to as negative examples. NUMCOLS is defined as the number of columns in a tuple in $P$. A tuple $p \in P$ indexed by a column $i$ yields a coordinate pair $\gamma = (x,y)$ representing a spreadsheet cell.

Let $V$ be a set of vertices, where $v \in V$ corresponds to a column in the end user’s desired output relation. Let $E$ be a directed set of edges, where $e \in E$ corresponds to a (cell constraint, spatial constraint) pair. $C = (V,E)$ is a complete, directed graph with $O(|V|^2)$ edges. The direction of an edge has a specific meaning. Cell constraints constrain the set of values that a source vertex can assume (e.g., a string “starting with a capital letter”). Spatial constraints constrain the possible spatial relationships between two vertices. The edge’s direction indicates the direction of the functional dependency between a source vertex and a target vertex (e.g., the target is “to the right” of the source).

We employ a number of heuristics in FLASHERELATE for the assignment of edge weights, described in [3.2.4]. Note that there may be many spanning trees that satisfy the examples given by the user. While all of these programs are correct with respect to the user’s examples, not all of them are what the user wants. Inferring user intent from incomplete specifications is a difficult problem. Our heuristics are thus employed to accomplish two goals: 1) to guide the search toward constraints most likely intended by users, reducing the number of examples needed, and 2) to speed the search by choosing the constraints known to execute faster (see [3.2.4]).

3.2 Algorithm

Informally, the synthesizer performs the following tasks, given $P$ and $N$:
1. For column $i$, learn the set of all cell constraints that satisfy the user’s positive examples (Fig. 8b). See [3.2.1].
2. For column pair $i,j$, learn the set of all spatial constraints that satisfy the user’s positive examples (Fig. 8c). See [3.2.2].
3. Any combination of learned cell and spatial constraints that form a spanning tree over all columns satisfies the user’s positive examples, but many of them include the user’s negative examples. Thus the last step is to find the set of constraints that excludes all negative examples (Fig. 8d). See [3.2.3].

We discuss implementation details in [3.2.4].

Example. We discuss a single round of synthesis using the algorithm shown in Fig. 8b by way of the running example in Fig. 8a. The desired relational output is shown in Fig. 10. Note that the column names in our example are for clarity; in actuality, FLASHERELATE uses a numbering scheme.

FLASHERELATE is used interactively:
1. The user calls FLASHERELATE with a sample tuple (a positive example) from the desired relation over data in the spreadsheet.
2. If the program extracts the relational table that the user wanted, the user is done. Otherwise, the user points out a discrepancy between the extracted table and the intended table with one of the following actions:
   (a) If the extracted table is missing some tuples, the user supplies at least one of the missing tuples as a positive example and calls FLASHERELATE again (step 3).
   (b) If the extracted table contains some unintended tuples, the user supplies at least one of the unintended tuples as a negative example and calls FLASHERELATE again (step 2).

While users must provide at least one new positive or negative example during each round of the synthesis procedure described above, they may also provide multiple examples. Suppose FLASHERELATE is given the following positive example: (Town: Hogsmeade, Street: 130 High St., Name: Hengist W.). This example also maps tuple column names to the $(x,y)$ spreadsheet coordinates: (Town: (1,2), Street: (2,2), Name: (3,2)), etc.
FLASH\textsc{RELATE} needs both representations. The first representation encodes the contents of a tuple; the second representation encodes the spatial relationships between columns in a tuple.

### 3.2.1 Step 1: Determine Cell Constraints

The FLASH\textsc{RELATE} synthesizer constructs cell constraints from regular expressions. A regular expression comes from one of two places: 1) it is constructed from a small set of standard character class tokens (EmptyCellTok, WhiteSpaceTok, AlphaTok, NumTok, and PunctTok), or 2) it comes from a small collection of commonly occurring string patterns that we identified while studying spreadsheets in the EUSES corpus. This strategy has been used by others \cite{1} in research for learning string programs.

Regular expression learning algorithms are outside the scope of this paper, but the topic is well-studied \cite{1}. Thus, the primary focus in this paper is in learning geometric patterns (see \S\ref{3.2.2}). FLASH\textsc{RELATE} is designed to work with any regex learning procedure that can learn from a set of positive string examples.

Since cell constraints may also include anchor constraints, we use the following simple anchor synthesis scheme. Given a set of positive example cells for column \(i\), the anchor synthesizer searches the spreadsheet for cells matching a common pattern (e.g., a header or whitespace) \(\rightarrow\) each positive column cell. If such a commonality is discovered, an \langle\text{anchor}\rangle\) is created. In practice, we find that empty cells are often used as anchors by the synthesizer, since whitespace is frequently utilized to draw attention to a related set of values (e.g., a column adjacent to whitespace).

Line \(1\) in Fig. \ref{fig:fig9} calls LearnC (Fig. \ref{fig:fig2}) for each column name \(i\) in the set of positive examples. LearnC eliminates those constraints that do not match all of the strings associated with column \(i\).

For pedagogical reasons, we limit the set of regular expressions used in the example. If the synthesizer chooses the program shown in Fig.\ref{fig:fig14}, the \(\rightarrow\) syntax indicates that the lhs is the parent of the rhs.

\begin{equation}
<\text{Town},\ [[a-zA-Z'].]\>$/>  
<\text{Street},\ [[0-9]\ [a-zA-Z'].]\>$/>  
<\text{Name},\ [\text{a-zA-Z'].}\>$/>  
\end{equation}

Figure 11: The example uses the above set of regular expressions.

### 3.2.2 Step 2: Determine Spatial Constraints

Line \(2\) in Fig. \ref{fig:fig4d} calls LearnS (Fig. \ref{fig:fig8c}) for each pair of column names \((i, j)\) in \(P\). LearnS finds all possible spatial constraints that satisfy the observed spatial layout between columns \(i\) and \(j\) from positive examples. The set of spatial constraints inferred for column name pair \((1, 2)\) for our single positive example is shown in Fig.\ref{fig:fig13}. We discuss pruning strategies to keep the search space small in \S\ref{3.2.4} “Pruning”.

### 3.2.3 Step 3: Find a Satisfying Set of Constraints

Next FLASH\textsc{RELATE} searches for a set of constraints that exclude the negative examples. The algorithm in Fig. \ref{fig:fig8d} performs such a search. There are five essential steps in this recursive procedure:

1. Exclude (cell, spatial) constraint pairs that would introduce a loop into the program graph given the current set of chosen constraints (line \(6\)).
2. Call Rank\(P\) to rank constraint pairs (line \(7\)). See \S\ref{3.2.4}.
3. Choose a constraint pair (line \(9\)).
4. Recursively choose the next pair of constraints (line \(10\)).
5. If constraints have been found for all the columns in the relation, ensure that the program excludes all of the negative examples (line \(12\)) and return the program. If not, backtrack (line \(16\)).

We define some additional terms used in Fig. \ref{fig:fig8d}. Lines \(6\) and \(14\) represent the algorithm’s implementation of nondeterministic choice. Each iteration of the \text{while} loop represents a choice point. \text{C\_\_p} denotes the set of chosen constraint pairs \((c, s)\) at a choice point where \(c\) is a cell constraint and \(s\) is a spatial constraint. \text{pairs}\_\_c denotes the set of all valid constraint pairs at the same choice point.

The search space may contain numerous solutions. The algorithm is free to choose any correct program that excludes all of the negative examples. The implementation of Rank\(P\) determines which pair of constraints the search ultimately chooses. We discuss these implementation choices in \S\ref{3.2.4}.

Suppose the synthesizer chooses the program shown in Fig.\ref{fig:fig14}. Since, in our example, the user has provided no negative examples, the chosen program trivially satisfies the criteria on line \(2\) in Fig. \ref{fig:fig4d}. If the program did not exclude any negative examples, the search would backtrack and consider a different constraint pair.
We omit these additional user interactions for brevity. We want, which we believe to be more natural. This choice also allows vertical or horizontal geometric descriptors. The naive approach one of more of the following classes.

1. All the deltas represent a fixed distance, and the set of deltas is a bijective relation from \( p \) to \( j \). This class produces match-all Kleene amounts. This class produces match-single (non-greedy \(*?\) and greedy \(*\)) Kleene amounts.

2. Pruning of the search space, for efficient search.

3. Choice of data structures.

**Ranking.** FLASHRELATE must choose between alternative constraints. The choice of more specific vs more general constraints impacts both the speed of the synthesizer and the number of examples required by the user. Favoring specific constraints may make the search fast, because they are more likely to rule out negative examples. Specific constraints may also mean that the user must provide more positive examples before the correct program is found. Conversely, favoring general constraints may fail to exclude negative examples, causing the search to backtrack frequently, slowing search. General constraints may require a user to provide more negative examples.

We tend to favor more specific over more general programs. This enables users to focus on what they want, instead of what they don’t want, which we believe to be more natural. This choice also allows for faster search. We rank constraint pairs by the following heuristics, in this order:

1. **H1** A constraint that excludes large numbers of negative examples is favored over a constraint that excludes few.

2. **H2** Specific spatial constraints are favored over general spatial constraints. This implies that multiple positive examples are required to learn non-constant-length spatial constraints.

3. **H3** Programs that encode simpler geometric layouts (i.e., fewer “turns”) are favored over ones with more complex layouts. We believe that users devise complex layouts only when needed.

**Pruning.** LEARNDIRAMOUNT is a function that takes a spreadsheet \( I \), a set of positive examples \( P \), a pair of column names \( i \) and \( j \), and a boolean that says whether the function should return vertical or horizontal geometric descriptors. The naive approach of enumerating all possible geometric descriptors fails when one considers that there are an infinite number of constant-length descriptors. Instead, LEARNDIRAMOUNT returns the largest set of descriptors consistent with the positive examples. Many descriptors can be ruled out based on the geometric layout of columns \( i \) and \( j \) from the positive examples.

A delta is the distance in either the horizontal or vertical direction, in terms of the number of cells, between \( p[i] \) and \( p[j] \). We use delta class information to prune the search space. Given the set of deltas derived from the positive examples for \( i \) and \( j \), examples fall into one of more of the following classes:

1. All the deltas represent a fixed distance, and the set of deltas is a bijective relation from \( p[i] \) to \( p[j] \). This class produces a single constant-amount for a set of deltas.

2. Delta distances vary, but the set of deltas is still a bijective relation from \( p[i] \) to \( p[j] \). This class produces match-single (non-greedy \(*?\) and greedy \(*\)) Kleene amounts.

3. Delta distances vary, and the set of deltas is not a bijective relation from \( p[i] \) to \( p[j] \). Specifically, one \( p[i] \) value maps to two or more \( p[j] \) values. This class produces match-all Kleene \((*)\) amounts.

4. Complexity Analysis

Let \( n \) denote the number of cells and let \( t \) denote the “tuple width”, i.e., the number of output columns. FLASH has a best-case run-time of \( O(n) \) when sufficient parallelism is available. FLASHRELATE has a best-case run-time of \( O(t \log t) \) when satisfactory programs can be found greedily. FLASHRELATE’s worst-case complexity is \( O(n^{21}) \) and is dictated by the size of the output. FLASHRELATE’s worst-case complexity is \( O(t^{t-2}) \) and is dictated by the number of spanning trees on a connected graph having \( t \) vertices. FLASH and FLASHRELATE were designed so that the worst-case run-times do not typically arise in practice. We discuss these results in more detail in the following sections.

4.1 **FLASH Run-Time**

A Flash program is evaluated in three phases: 1) cell constraint evaluation, 2) spatial constraint evaluation, and 3) reduction of cell and spatial constraint matches to output tuples.

1. In the worst case, all \( n \) cells satisfy all \( t \) cell constraints. Thus FLASH must inspect every cell once for every cell constraint. The string matching implementation also has a cost that we denote \( k \) and is a function of the length of the string. Since there are \( t \) cell constraints, this cost is \( O(nkt) \). If the cost of string matching is linear in the length of the string as in traditional DFAs, the cost is dominated by the other terms.

2. In the worst case, all \( n^2 \) cell pairs satisfy all \( t-1 \) spatial constraints. Thus FLASH must evaluate every cell pair once for every spatial constraint. This cost is \( O(n^2t) \).

3. The remaining cost is proportional to the size of the output, since the algorithm constructively enumerates all tuples that satisfy the FLASH program. In the worst case, if every cell matches...
every cell constraint and every spatial constraint is satisfied by every pair, then the number of tuples in the output can be $O(n^3)$. Thus the total worst-case cost is $O(nkt) + O(n^2t) + O(n^3) = O(n^3)$. $t$ is often smaller than the total number of cores on a modern machine, and we make use of this fact in FLARE’s implementation, which is parallelized.

Note that FLARE never re-evaluates a constraint in $\mathcal{H}_1$ and $\mathcal{H}_2$ with respect to a given input because of a caching mechanism that requires $O(n^2t)$ space in the worst case. To have an output as large as the worst case, FLARE needs to extract every combination of input cells $t$ times—an unusual and not particularly useful extraction. As shown by our benchmarks, such programs do not arise in typical scenarios, where the cost is typically closer to linear in the size of the spreadsheet.

4.2 FlashRelate Run-Time

The worst-case time complexity of FlashRelate depends on the size of the space of possible programs times the cost of running each FLARE program. All valid FLARE programs are spanning trees. Thus the number of possible programs equals the number of possible spanning trees for a complete graph on $t$ vertices, or $O(\binom{n}{t})$ programs. The worst case occurs when there is exactly one valid program, found only after evaluating all other programs.

The size of the search space means that exhaustively searching through all permutations of possible constraints is impractical. Instead, we use heuristic search and a weighting scheme such that the minimal (or close to minimal) spanning trees are likely to be the user’s intended program. Synthesis cost is thus be dictated by the cost of a minimal spanning tree algorithm. Kruskal’s algorithm has a cost of $O(t \log t)$.

5. Evaluation

In this section, we evaluate the design of FLARE and FlashRelate on a variety of real-world spreadsheets. We answer the following questions:

1. Is it possible to manually write FLARE programs to perform a diverse set of extraction tasks?
2. Can FlashRelate automatically infer equivalent programs for the same set of tasks as in question 1?
3. How effective are our heuristics at reducing the time and number of examples required by the synthesizer?
4. Is FlashRelate usable in the real world?

5.1 Benchmark Spreadsheets and Tasks

To evaluate FlashRelate, we assembled a collection of 43 benchmarks using spreadsheets taken from other work on reorganizing spreadsheet tables, from our own microbenchmarks for testing purposes, and from a large spreadsheet corpus created for research purposes.

Benchmark Selection. Our evaluation considers two sets of benchmarks. The first set of benchmarks were borrowed from related work [13] that examined 51 table-transformation programs from Excel help forums. Despite the apparent complexity of these tasks, we found to our surprise that nearly half (22) of the transformations were straightforward relational extraction tasks that could be expressed in FLARE. The remainder performed computation (e.g., arithmetic) outside the scope of extraction programs. To round out our evaluation with more difficult tasks, we assembled a second set of benchmarks by searching the EUSES spreadsheet corpus [10] for spreadsheets with complex ad-hoc layouts. This second set of benchmarks was chosen specifically to test our synthesis algorithm against challenging extraction tasks. We supplemented this second set with synthetic benchmarks known to present challenges to our synthesizer. As a measure of the complexity of the extraction task for each benchmark, we note the number of indeterminate geometric descriptors that appear in our ground truth programs ("k*" in Fig. 17). Expressiveness. To evaluate the expressiveness of FLARE, we manually wrote a correct FLARE program for each benchmark and extracted the resulting output table. In the course of this effort, we found that the FLARE language is expressive enough to extract the desired tuples from all of the multi-dimensional data patterns we observed. We conclude that FLARE is an effective tool for performing extraction tasks.

Synthesizer Experiments. Using our ground truth programs, we compare the results of the FlashRelate synthesis algorithm to evaluate its effectiveness. Recall that synthesis depends on providing a set of positive and negative examples to the synthesis algorithm as described in [3].

The following method is intended to simulate a user interacting with FlashRelate. The relational table extracted by the ground truth program represents the user’s desired extraction output; we call this the oracle. After each invocation of the FlashRelate algorithm by our simulated user with a set of examples, we determine whether the synthesized program is correct by comparing its output against the oracle. When the FlashRelate output differs, the simulated user finds the first tuple that deviates from the oracle by scanning the extracted table from top to bottom. Deviant tuples come in two forms: 1) if a tuple from the ground truth is missing from the program output, it is a positive example; 2) if a tuple from the program output does not appear in the ground truth, it is a negative example. We repeat this process until the synthesizer either finds a program whose output matches the oracle’s or it times out. 10 minutes was chosen as the maximum total duration of the task as we felt few users would wait longer.

In our experiments, we consider 6 algorithm configurations to understand the benefit of our ranking choices. In all cases, regular expressions come from a small corpus (< 100) of common patterns combined with the regular expression generator described in §3. Each configuration examines the effect of adding a heuristic to the FlashRelate algorithm. Experiment configurations are: 1) $H_1$, $H_2$, $H_3$, $H_4$, 2) $H_1$, $H_2$, and $H_3$, 3) $H_1$ and $H_2$, 4) $H_1$, 5) no ranking, and 6) $R$. $H#$ refers to the heuristic described in §3.2.4.

Configuration R changes the oracle model, choosing examples randomly. It tests whether FlashRelate is robust to a wide variety of user example-selection behaviors. Example order may matter because FlashRelate learns geometric differences from one example to the next. Anecdotally, examples located near each other in a spreadsheet are geometrically more similar than examples further from each other. When examples are chosen randomly, they are unlikely to be repeatedly drawn from the same neighborhood. We enable all heuristics, run the experiment 30 times, and average the results.

5.2 Experimental Setup

We evaluated synthesis on typical end-user hardware. Our test machine was an AMD Phenom II X4 940 quad-core desktop machine running at 3GHz with 4GB of RAM. FlashRelate was written in a mix of F# and C# and can be used both as a VSTO.NET plugin for Microsoft Excel and with a web UI we built to experiment with user interfaces.

5.3 Results

Fig. [15] shows the total time it takes to synthesize our 43 benchmarks. The y-axis shows the running time of the synthesis algorithm in cases where it succeeded. The axis is truncated at 120 seconds because, with ranking, most benchmarks succeed well before that time.
In the best case (all rankings), the algorithm failed to find a solution within 10 minutes for only 1 out of 43 benchmarks. When synthesis found a correct solution, more than 80% of the benchmarks completed in less than 10 seconds total. Per-iteration time is extremely fast: typically a user only has to wait 1.6 seconds (median: 0.6 seconds).

In the failing case, Appen4-5, many of the columns in the same tuple have the same basic numeric type. Furthermore, all of the spatial constraints need to be of indeterminate length. The way that the hand-written ground truth program adds sufficient discriminating power is with an anchor that specifically looked for a 1-2 digit number to the left of a capture. Our naive anchor synthesizer was not designed to generate this pattern, since it only looked for empty cells and strings shared by multiple cells in the same output column. We modified the code to generate numeric patterns and we were able to synthesize this last benchmark in 69.52 seconds, but we found that this change decreased the performance of the other benchmarks. Designing a smarter anchor synthesizer is a subject for future work.

Fig. 16 shows the number of examples (iterations) required to synthesize the correct result. Simulated users provided an average of 3.5 positive examples (median: 3 positive examples) and 2.0 negative examples (median: 1 negative example). Without ranking, the algorithm is significantly slower, and fails to find a solution more often before a timeout occurs (13 total timeouts). Without ranking, the algorithm also requires many more examples: an average of 3.5 positive examples (median: 3 positive examples) and 2.0 negative examples (median: 3 positive; 9 negative). We conclude that without ranking, sometimes very general solutions are generated (matching too many cells) and these numerous negative examples are required to narrow the selection.

Random example selection (config, R in Fig. 16a & b) does have a small effect on the speed and number examples needed by the synthesizer, but our heuristics still perform well even in this pathological case. For Fig. 16b, the standard error of the mean (SE)
ranges from 0.002 seconds when $x$ is small to 35.5 seconds when $x$ is large. For Fig [16b], SE ranges from 0 when $x$ is small to 7.62 when $x$ is large. Not surprisingly, this means that the benchmarks exhibit more variation with regard to time and examples as we make claims about increasingly large proportions of the suite.

Table [17] summarizes the time and number of examples required with all heuristics enabled as well as benchmark complexity.

**Summary.** We find that FLASHRELATE synthesizes correct programs quickly and with little user effort. While one ranking scheme sacrifices higher speed for lower effort (R4), generally our ranking schemes reduce both the wait and the effort.

5.4 Case Study

What is the FLASHRELATE user experience like? We were given the opportunity to test the tool in an unplanned real-world scenario brought to our attention by a separate group of researchers.

Economists and psychologists studying food perception curated a large collection of photographs of school lunches for the purpose of studying participants’ perceptions about food healthiness. Participant responses would then be compared against ground-truth nutritional data in an analysis written in the R language. But the procedure ran into difficulty: the school supplied researchers with PDF printouts of the nutrients for each meal. Consequently, the information needed to be extracted into a normalized CSV form before it could be used. A sample is shown in Fig. [16].

Graduate students assigned to the data-extraction task resorted to manual data entry due to several obstacles: 1) the data could not be reliably copy-and-pasted because it was in PDF form, 2) the data was in a nested tabular form that did not conform to their desired input format, and 3) the data contained many blank entries, requiring post-processing. Given the level of effort required to manually enter the data (hundreds of pages), we were asked if we could help.

We solved the first problem using Tabula [24]. Tabula extracts tables from PDFs into CSVs using a vision algorithm. The tool was developed by the non-profit organization ProPublica to address the needs of journalists who often need to extract data from government and corporate documents [29].

After conversion to a CSV the data was still unnormalized. Nutrients were grouped by each food item, and to save vertical space, were presented side-by-side. Using our FLASHRELATE web UI, we synthesized an extraction program by giving examples in our desired format. While the tool occasionally produced odd extractions, requiring additional positive and negative examples, we were able to quickly find a satisfactory program. In total, the task took about 10 minutes and required us to supply 5 positive and 5 negative examples. For comparison we also hand-wrote a FLARE program, which took more than an hour to produce.

Finally, we performed post-processing (primarily the removal of asterisks) using FLASHFILL [11]. FLASHRELATE was specifically designed to work in spreadsheet-manipulating toolchains with tools like FLASHFILL, thus validating our choice.

Mentally mapping output tuples back to their original location in the spreadsheet is difficult; knowing this mapping helps inform which examples to give next. For large spreadsheets, it is hard to know when a synthesized program extracts an entire spreadsheet.

Discussion. Our case study gave us insights into how a user interface influences data extraction for users, and we refined our interface to better support the task. For example, when a user clicks on an output tuple, the set of cells that produced it are highlighted, allowing the user to quickly determine whether the inferred FLARE program extracted the intended table in its entirety. Additionally, like WRANGLER [16], we added a histogram to show the number of distinct string matches for each column. Observing that the counts match one’s own intuition helps to track down useful positive examples.

6. Related Work

**Extracting Data from the Web.** An important related body of work focuses on extracting relational data from data on the web. Like FLARE, SXPATCH [24] includes spatial primitives in its queries, but does not attempt to synthesize programs in the query language from examples, as we do. While SILA [23] defines spatial abstractions like FLASHRELATE, it attempts to extract records from spatially structured data algorithmically, and not from examples. [8] and [2] provide good overviews for the range of approaches taken.

Wrappers are procedures to extract data from Internet resources. Wrapper induction is the method to automatically construct wrappers [16]. There has been a wide variety of work in this area, ranging from supervised systems [15][18][22], semi-supervised systems [3], to unsupervised systems [6]. Our work enables users to induce wrappers interactively using examples.

**Extracting Data from Spreadsheets.** SENAIZU [3] attempts to automatically infer hierarchical structure in spreadsheets using a set of classifiers. By contrast, FLASHRELATE can be used to perform arbitrary extraction tasks from arbitrary spreadsheets.

HAEXCEL [7] focuses on recovering the true relational schema from the spreadsheet data. We believe that the appropriate schema is task-dependent. FLASHRELATE returns the set of tuples that the user wants based a set of user-supplied examples. OPENREFINE [27] and WRANGLER [16] help users clean and transform their spreadsheet data into relational form. While OPENREFINE typically requires users to program, WRANGLER automatically infers likely transformation rules and presents them in natural language. Two pitfalls with WRANGLER are that users must be understanding of the available transforms, and must be capable of finding alternate transforms in the event that the inference is wrong. FLASHRELATE sidesteps this problem by allowing the user to work exclusively with output examples. Users need not understand the semantics of transformations, only their effect.

GYRO [14] expresses spatial constraints in the form of geometric regions. FLARE instead represents spatial constraints as vectors in 2D space since some patterns are inexpressible as regions. GYRO’s inference algorithm is based on searching a database of known programs. Since FLASHRELATE can be used to generate arbitrary transforms, a different approach was needed.

**Programming by Example.** The area of programming by example (including FLASHRELATE) promises to enhance productivity for end users [12][20]. The most closely related work is PROGFROMEX [13], which performs tabular transforms for spreadsheets in tabular format. FLASHRELATE addresses arbitrary transformations, employs heuristic search (PROGFROMEX uses version-space algebra), and needs only output examples (PROGFROMEX needs input-output examples). Another recent technology, QUICKSILVER [21], synthesizes relational algebra queries over normalized spreadsheet tables. QUICKSILVER cannot handle any of the transformation tasks in our benchmarks.

**Extracting Data from Text Files.** The PADS project simplifies ad hoc data-processing tasks for programmers by developing DSLs and learning algorithms to extract data from textual formats [9].

FLASHEXTRACT supports a by-example framework for extracting data from text files [19]. FLASHEXTRACT uses relative string positions, thus it cannot synthesize extraction programs that make use of 2D spatial information (including the motivating example in Fig. [1]). FLASHEXTRACT programs are also strictly hierarchical, so multiple constraints cannot refer to “overlapping” regions. FLASHRELATE has no such restriction.
We gratefully thank Gustavo Soares for his assistance in designing The flexibility of spreadsheets allows users to specify ad-hoc formats, providing flexibility at the expense of automated processing. We present FLARE, the first language that allows users to express relational extraction queries against spreadsheets. We also present FLASHRELATE, an algorithm that automatically synthesizes FLARE programs from user-provided examples. We designed the interface to be simple, fast, and efficient. Users need only point and click to obtain the extractions that they want. Notably, users need no knowledge of programming to liberate their data.

7. Conclusion

The flexibility of spreadsheets allows users to specify ad-hoc formats, providing flexibility at the expense of automated processing. We present FLARE, the first language that allows users to express relational extraction queries against spreadsheets. We also present FLASHRELATE, an algorithm that automatically synthesizes FLARE programs from user-provided examples. We designed the interface to be simple, fast, and efficient. Users need only point and click to obtain the extractions that they want. Notably, users need no knowledge of programming to liberate their data.

Acknowledgements

We gratefully thank Gustavo Soares for his assistance in designing and building the web IDE user interface for FlashRelate. Daniel Barowy was supported by a Microsoft Research internship and NSF grant #1349784 during this work.

References


![Figure 18: A sample entry from the nutritional dataset given to us for extraction using FLASHRELATE. Output tuples needed to be in the form (Description, Nutrient, Value, Units) such as (ORANGE CHICKEN, Sodium, 453.8952*, mg).](image)