FLASHRELATE: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples

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 in a two dimensional grid. 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 vast amounts of real-world data is “locked-in” to a proliferation of one-off formats.

We introduce FLASHRELATE, a synthesis engine that lets ordinary users extract structured data from spreadsheets without programming. Instead, users drive the extraction process by specifying output examples, which FLASHRELATE uses to synthesize a program in FLARE. FLARE is a novel extraction language that extends regular expressions with geometric constructs. We built an interactive user interface on top of FLASHRELATE that lets end-users generate FLARE programs by point-and-click. We demonstrate that correct extraction programs can be synthesized in seconds from a small number of examples for 43 real-world scenarios. Finally, our case study shows that FLASHRELATE addresses the widespread problem of data trapped in corporate and government formats.

1. Introduction

There are an estimated 500 million Microsoft Excel users worldwide [Register 2009]. 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 [Fisher and Rothermel 2005]. Furthermore, CSV-formatted files, which are isomorphic to spreadsheets, are a widely-used format. For example, the U.S. Census Bureau distributes official government data in this 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 gives the spreadsheet creator a large degree of freedom when encoding their data. Furthermore, 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 an encoding that projects that data into two dimensions. Thus while spreadsheets allow compact and intuitive visual representations of data 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 problem is widespread. Research suggests that the vast majority of spreadsheets available on the web cannot be trivially converted to database relations [Chen et al. 2013]. 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 user-help forums suggests that many users do not realize their mistake until they have invested significant time entering and organizing their data.

Other domains have encountered this problem and addressed it with new software tools. The fact that non-experts are unlikely to embrace appropriate data management tools motivated the development of numerous domain-specific data extraction features: 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 [Zhu et al. 2012; Fisher and Walker 2011] and FlashExtract [Le and Gulwani 2014] aim to make text and web extraction technologies easier to use for end-users. There are no such tools for extracting data from spreadsheets.

A video demonstration is available at: http://tinyurl.com/mh3bo3a
Figure 1: A semi-structured spreadsheet excerpt with two example tuples highlighted. The top row and leftmost column represent $x$ and $y$ coordinates respectively. The first tuple (red) represents the timber harvest (per 1000 hectares) for Albania in 1950. The second tuple (blue) represents the timber harvest for Austria in 1950. The FLARE program in Fig. 4 performs this extraction.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>value</td>
<td>year</td>
<td>value</td>
<td>year</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Albania</td>
<td>1000</td>
<td>1950</td>
<td>930</td>
<td>1981</td>
</tr>
<tr>
<td>3</td>
<td>Austria</td>
<td>3139</td>
<td>1951</td>
<td>3177</td>
<td>1955</td>
</tr>
<tr>
<td>4</td>
<td>Belgium</td>
<td>541</td>
<td>1947</td>
<td>601</td>
<td>1950</td>
</tr>
<tr>
<td>5</td>
<td>Bulgaria</td>
<td>2964</td>
<td>1947</td>
<td>1959</td>
<td>1958</td>
</tr>
<tr>
<td>6</td>
<td>Czech</td>
<td>2416</td>
<td>1950</td>
<td>2503</td>
<td>1960</td>
</tr>
</tbody>
</table>

Figure 2: An extracted relational table with the same two tuples highlighted as shown in Fig.1.

**Contributions** We make the following contributions:

- We describe a domain-specific extraction language, FLARE, that describes the domain of ad-hoc encodings (see §2). FLARE is the first pattern-based spatial query language explicitly designed to extract relational data from spreadsheets.

- We present an algorithm, FLASHRELATE, that synthesizes FLARE programs given a small number of user-provided positive and negative examples so that users can extract data without programming (see §3).

- We empirically evaluate FLARE’s expressiveness and the effectiveness of FLASHRELATE against a set of 43 real-world spreadsheets drawn from a research corpus and from Excel user help forums. We show that the FLASHRELATE algorithm is able to synthesize correct FLARE programs from a small number of examples, typically under 2 seconds (see §4).

- We describe a novel interactive environment whose design was influenced by a case study (see §4.4).

**Example** To make the problem concrete, we refer to the running example in Fig. 1, a real spreadsheet taken from the EUSES corpus [Fisher and Rothermel 2005]. The spreadsheet shows timber harvests by country and year. It packs data about a particular country into value/year pairs within a row, appending comments to the far right. The original author probably structured the data in this form to avoid data in a long thin column, which would be harder to read.

While this representation is visually convenient, consider what needs to be done in a simple scenario: computing the average harvest in 1950. The encoding 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 all in the same column. One solution is to convert the AVERAGE expression to use a union of single cells (a logical range of values like 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 user help forums suggest that users often ask for help reformatting their data in this situation [Harris and Gulwani 2011]. Help is typically offered in the form of Visual Basic programs that perform the reformatting for them2. Writing even such a simple program 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 a heading titled value. Year values are always located to the right of the paired timber value. The key insight in FLARE is that we can use intuitive geometric relationships like relative direction and distance to concisely express the desired extraction. We motivate our discussion of the FLARE language (Section 2) using this example.

2. **Flare Language**

FLARE converts ad hoc formats into relational forms that can be used by relational tools. 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 general-purpose support code to capture relational information encoded in spreadsheets. FLARE augments regular expressions with geometric constraints needed to extract this relational data without support code.

FLARE has two kinds of constraints. 1) A cell constraint defines a class of valid strings for a field of an output tuple.

Figure 3: A simple FLARE program for the example extraction task. $\sim e$ stands for a regular expression that matches anything but whitespace. We describe FLARE’s syntax in Section 2.

Figure 4: A FLARE program with an additional constraint to extract country names, which are an indeterminate distance to the left of each matched year, value pair. own stands for the regular expression that matches alphabetic characters and whitespace. See §2 “Geometric descriptors”.

- We provide a 25-line VBA program for this example in the appendix.
2) A **spatial constraint** define a class of valid geometric relationships between fields of a tuple. Taken together, these two types of contraints form a constraint graph, specifically a tree. Cell constraints represent vertices in the graph while spatial constraints represent edges in the graph. Intuitively, one can think of this graph as a geometric structure that, when translated over a spreadsheet, indicates which cells should form a relational tuple.

We provide the syntax for **FLARE** in Fig. 5. The language's formal semantics are provided in an appendix. We discuss the language informally here to give the reader an intuitive sense for **FLARE**.

**Cell constraints.** In **FLARE**, cell constraints are based on regular expressions\(^3\). Regular expressions are delimited by a pair of slashes, `/`. The expression `/\^[0-9]+$/` matches a string that contains only numbers. Here on, we refer to this expression as `Num`.

Informally, a cell constraint is a boolean function that takes a spreadsheet and a cell location as parameters and returns `true` if the constraint is satisfied by the string in the spreadsheet at that location. We refer to the cell location parameter as the `root` of a search since **FLARE** programs must always start at a particular location. **FLARE** achieves the effect of searching an entire spreadsheet by evaluating the program at all of the roots in a spreadsheet.

As with many regular expression implementations, **FLARE** allows a cell constraint to `capture` data. Capturing means that data is returned as a field in a tuple in addition to being used as a constraint. Capture is denoted by enclosing a cell constraint within a pair of angle brackets, `<.`. A valid **FLARE** program must contain at least one capturing cell constraint.

In order to create a tuple, captured cells must be associated with field names (see `<cell>` in Fig. 5). Our working expression can be named “Harvest” by changing the expression to `<Harvest,Num>`, the smallest complete **FLARE** program that both matches numeric strings and extracts them. This program yields a set of 1-tuples containing all of the numeric strings from the spreadsheet.

Observe that `<Harvest,Num>` is not sufficiently discriminator to match *only* timber harvest entries in Fig. 1. Year values also have this format. Cell constraints allow for a restriction called an `anchor` that provides extra context. In this case, we want entries to match only when the header value is present in the same column.

Anchors are appended to the end of a constraint, starting with a `:` character `<anchor>`, Fig. 5). In this case, the anchor `u+/value/` states that the text “value” must be in an entry at least one cell above the entry matched by the preceding constraint. The **FLARE** program `<Harvest,Num>:u+/value/` is sufficient to match only numeric strings under the header value. In fact, we have over-specified the program: `<Harvest,:-?u+/value/`, where `-?` stands for the regular expression that matches non-whitespace strings, is sufficient to match harvest values. We describe the syntax of anchors in more detail below.

Following the same exercise, we can produce a program that extracts year entries. One such program is `<Date,:-?u+/year/`. To produce the 2-tuples needed to perform our task (recall that we want to compute the average timber harvest for 1950) we need a way to combine these extraction programs.

**Spatial constraints.** Spatial constraints describe the relative spatial relationships between fields `<spatial>`, Fig. 5). In our example, observe that year entries are always to the right of `value` entries. Spatial constraints compose **FLARE** subprograms to produce larger output tuples. There are four basic geometric descriptors, `up`, `down`, `left`, and `right`, denoted by `u`, `d`, `1`, and `r` respectively `<vert>` and `<horiz>`, Fig. 5). 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 infix operators for a pair of cell constraints. One may informally think of a spatial constraint as a boolean function that takes two cell locations as parameters and returns `true` if those two cells satisfy the specified geometry in the spreadsheet.

As mentioned earlier, cell constraints have a cell location parameter referred to as the `root`. When composing two subprograms, we must decide which to execute first, as spatial constraints are directed. In practice this means that the set of cell locations matched by a `parent` subprogram are paired with all possible `destination` cells in the spreadsheet and evaluated against the compositional spatial con-
constraint. Those destinations that satisfy the spatial constraint are passed to the child constraints as a set of roots. In some cases, which subprogram should be the parent is arbitrary; in other cases, the choice is not. Here, we arbitrarily choose to match values first.

There is one more bit of notation related to parent-child relationships, and that is the child-of operator (2nd rule for <prog>, Fig. 5). This operator, denoted by a pair of square brackets, indicates that cells matching constraints listed inside the brackets are children of cells matched by the constraint to the left of the brackets. Multiple child constraints are allowed inside the brackets. This allows us to create multiple spatial constraints for the same set of parent constraints. This is an important construction in FLARE, and its presence dramatically affects the speed of our synthesis algorithm because it allows us to represent the program as a tree (see §3).

In our example, we have (arbitrarily) decided that Date entries should be relative to Harvest entries. Since Date entries are to the right of Harvest entries, we compose the two programs with the r constraint as in <Harvest, ¬r: w=value/〈r<Date, ¬r]]. Note that the combined program allows us to omit the anchor in the child Date constraint. This is the same program shown in Fig. 3, and it produces 2-tuples sufficient to answer the user’s year-average query using Excel’s point-and-click capabilities.

**Geometric descriptors.** FLARE’s spatial constraints are very expressive. Geometric descriptors may be appended with a constant amount (<amount>, 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 to its parent (e.g., “to the right”) but that the distance is indeterminate.

To be concrete, suppose we modify our example such that 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. We need to leave the distance undetermined. We use Kleene star (•) for this purpose (<k•> and <k•>, Fig. 5).

As with other regular expression implementations, the Kleene star can be interpreted in a number of different ways. A Kleene star can return a set of all matches or a single match. 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. For example, the regular expression abc• with match-all semantics matches the set of strings {ab, abc, abcc, abccc}. With greedy single match semantics, the same regular expression matches the string abcc, and with non-greedy single match semantics, it matches the string ab.

Similarly, 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 have the interesting consequence of forming a one-to-many relationship between the parent cell and the matched child cells, unlike other constraints in FLARE. When there is a one-to-one relationship between the parent and its children, the matching child cells for a given parent cell can be combined together into a single tuple, containing one field for a given parent cell and one field for each child. A one-to-many relationship means that we must produce multiple tuples for a given parent cell. This is accomplished by performing a cross-product between the parent and each set of child matches.

**Anchors** Anchors function purely as predicates indicating whether a common neighbor is present, thus they do not capture cells. This means that 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 limit anchor expressiveness (no recursive anchors) since unbounded anchor subprograms increase the search space of possible programs for FLASHRELATE to explore, with no obvious benefit.

### 3. FlashRelate Synthesis Algorithm

Programmatic solutions to data extraction suffer two key limitations. First, the expertise required to use programming tools is often particular to specific ad-hoc formats. Second, and more significantly, they require a knowledge of programming. The first aspect creates challenges even for programmers, while the second aspect puts these solutions out of reach of the vast majority of end users.

The goal of the FLASHRELATE algorithm, shown in Fig. 6a-e, is to automatically generate a FLARE program that is consistent with the positive and negative examples supplied by the user. We frame the problem of finding a correct program as a search over all valid combinations of constraints that satisfy the positive and negative examples. As noted earlier, cell and spatial constraints in FLARE represent vertices and edges in a constraint graph, respectively. Taken together, a valid program must form a tree. If we encode our search problem such that each column is represented as a vertex, and all possible edges form a complete digraph, then the problem of finding a satisfying program reduces to that of finding the cell and spatial constraints that form a spanning tree. Thus FLASHRELATE’s program search procedure, shown in Fig. 6b, is modeled on a recursive formulation of Kruskal’s spanning tree algorithm.

We also employ a number of heuristics in FLASHRELATE. Note that there may be many constraint graphs 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 de-
SYNTH(I, P, N)
1 for each column index i
2 \( A_C[i] = \text{LearnC}(I, P, i) \)
3 for each pair of field names \( i, j \) such that \( i \neq j \)
4 \( A_S[i, j] = \text{LearnS}(I, P, i, j) \)
5 return \( \text{Search}(\emptyset, N, A_C, A_S) \)

(a)

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

(b)

LEARN(S(I, P, i, j))
1 \( A_S = \emptyset \)
2 \( V = \text{LearnDirAmount}(I, P, i, j, \text{TRUE}) \)
3 \( H = \text{LearnDirAmount}(I, P, i, j, \text{FALSE}) \)
4 for each \( [v, h] | v \in V, h \in H \)
5 \( A_S = A_S \cup \{ \text{Spatial}(i, j, v, h) \} \)
6 return \( A_S \)

(c)

Figure 6: FLASHRELATE’s program synthesis procedures. (a) is a top-level procedure that first precomputes all possible cell and spatial constraints satisfying the positive examples, \( P \), and then calls the search routine. \( p \in P \) is a tuple, and when indexed by a field name, yields a cell coordinate pair \((x, y)\). (d) is the program search procedure. The result is a set of column pairs guaranteed to be a tree. Not shown is the routine that inserts child-of operators \([\text{CELL}]\). (b) learns cell constraints from positive examples; \( \text{Cell}(i, \alpha) \) is a cell constraint constructor that takes a field name \( i \) and a regular expression \( \alpha \). (d) learns spatial constraints from positive examples; \( \text{Spatial}(i, j, v, h) \) is a spatial constraint constructor that takes two field names \( i \) and \( j \), a vertical geometric descriptor \( v \), and a horizontal geometric descriptor \( h \). \( \text{LearnDirAmount} \) is a function that enumerates geometric descriptors; see \( \S \)3.2.4 “Pruning” for details. (e) The two expressions signify the negative examples excluded cell and spatial constraints, respectively.

Sierks from incomplete specifications is a difficult problem. Our heuristics are thus employed to accomplish two goals: 1) they guide the search toward constraints more likely to be wanted by users, thus reducing the number of examples needed, and 2) they speed the search by choosing the constraints known to execute faster.

3.1 Definitions

We use the following terms in Fig. 6. Let \( P \) be a set of user-provided tuples representing desired program outputs, from this point on referred to as positive examples. Let \( N \) be a set of user-provided tuples representing counterexamples, from this point on referred to as negative examples. \text{NUMCOLS} is defined as the number of fields in a tuple in \( P \). A tuple \( p \in P \) indexed by a field \( i \) yields a coordinate pair \( \gamma = (x, y) \).

Let \( V \) be the set of field names in \( P \). Let \( E \) be a set of directed edges, each edge referred to by a pair of field names \( (i, j) \). Let \( G = (V, E) \) be a complete digraph.

3.2 Algorithm

Informally, the synthesizer must perform the following tasks, given \( P \) and \( N \):

1. Determine a set of cell constraints for each field specified in the user’s positive examples (Fig. 6c). See \( \S \)3.2.1.
2. Determine a set of spatial constraints for each field pair specified in the user’s positive examples (Fig. 6d). See \( \S \)3.2.2.
3. Any combination of the remaining cell and spatial constraints that form a tree over all fields satisfies the user’s positive examples, but many of them include the user’s negative examples. Thus the last step is to identify a combination of cell and spatial constraints that excludes all negative examples (Fig. 6b). See \( \S \)3.2.3.

We discuss a number of implementation details that make FLASHRELATE’s search procedure efficient in \( \S \)3.2.4.

Example We discuss a single round of synthesis using the algorithm shown in Fig. 6 by way of the running example in Fig. 7. The desired relational output is shown in Fig. 8. FLASHRELATE is intended to be used in the following interactive manner:

1. The user calls FLASHRELATE with a sample tuple (a positive example) from the desired relation over data in the spreadsheet. FLASHRELATE returns a program to the user.
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:
- If the extracted table is missing a tuple, the user provides a positive example of the missing tuple and calls FLASHRELATE again (step 2).
- If the extracted table contains an unwanted tuple, the user provides a negative example of the unwanted tuple and calls FLASHRELATE 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 FLASHRELATE is given the following positive example: \((1: \text{Hogsmeade}, 2: 130 \text{ High St.}, 3: \text{Hengist W.})\). This example also maps tuple field names to spreadsheet coordinates: \((1: (1, 2), 2: (2, 2), 3: (3, 2))\). FLASHRELATE needs both representations. The first representation encodes the contents of a tuple; the second representation encodes the spatial relationships between fields in a tuple.

### 3.2.1 Step 1: Determine cell constraints

The FLASHRELATE synthesizer constructs cell constraints from regular expressions. A regular expression comes from one of two places: 1) it is dynamically 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 [Gulwani 2011] in research for learning string programs.

Regular expression learning algorithms are outside the scope of this paper, but the topic is well-studied [Angluin 1987]. Thus, the primary focus in this paper is in learning geometric patterns (see §3.2.2). FLASHRELATE 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 field \(i\), the anchor synthesizer searches each cell’s neighbors for a similarly-located common string. If such a common neighbor is discovered, an Anchor is added as a possibility for the appropriate cell constraint. In practice, we find that empty cells are frequently used as anchors by the synthesizer.

Line 1 in Fig. 6a calls LearnC (Fig. 6c) for each field name \(i\) in the set of positive examples. LearnC eliminates those constraints that do not match all of the strings associated with field \(i\).

For pedagogy, we limit the set of regular expressions to those shown in Fig. 9; in practice, the set is larger. Returning to our example, if the synthesizer is given this set, LearnC returns the constraints shown in Fig. 10.

### 3.2.2 Step 2: Determine spatial constraints

Line 3 in Fig. 6a calls LearnS (Fig. 6d) for each pair of field names \((i, j)\) in \(P\). LearnS finds all possible spatial constraints that satisfy the observed spatial layout between fields \(i\) and \(j\) from 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 \(p[j]\). For each positive example \(p\), LearnS calculates the delta for the cells (lines 2-3) \(p[i]\) and \(p[j]\). We use delta class information to prune the search space (see §3.2.4, “Pruning”).

The set of spatial constraints inferred for field name pair \((1, 2)\) for our single positive example is shown in Fig. 11.
3.2.3 Step 3: Find a satisfying set of constraints

Next FlashRelate searches for a set of constraints that satisfy its negative examples. As shown in Fig. 6b, 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 7).
2. Call RankP to rank constraint pairs (line 8). See §3.2.4.
3. Choose a constraint pair (line 10).
4. Recursively choose the next pair of constraints (line 11).
5. If constraints have been found for all the fields in the relation, ensure that the program excludes all of the negative examples (line 2) and return the program. If not, backtrack (line 4).

We define some additional terms used in Fig. 6b. Lines 6-15 represent the algorithm’s implementation of nondeterministic choice. Each iteration of the while loop represents a choice point. \( C_S \) denotes the a set of chosen edges at a particular choice point. \( C_P \) denotes the set of chosen constraint pairs \((\alpha, \beta)\) at the same choice point where \( \alpha \) is a cell constraint and \( \beta \) is a spatial constraint. \( \text{pairs} \) 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 RankP determines which pair of constraints the search ultimately chooses. We discuss these implementation choices in §3.2.4.

In our example, suppose the synthesizer chooses the program shown in Fig. 12. Since, in our example, the user has provided no negative examples, the chosen program trivially satisfies the criteria on line 2 in Fig. 6. Were this not the case, the search would backtrack and consider a different constraint pair.

The output of the program in 12 is shown in Fig. 13. While all of the extracted tuples are correct, the output is missing rows with omitted towns in the original spreadsheet (Fig. 7). The user’s next step is to add another positive example (e.g., the 2nd tuple in Fig. 8) and call the synthesis algorithm again (Fig. 6a). This procedure continues until the user is satisfied with the output of the algorithm. We omit these additional user-interactions for brevity.

3.2.4 Implementation Details

The efficient operation of the synthesis algorithm depends on three key implementation ideas:

1. Ranking schemes for synthesizer search choices.
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. But 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** Constraints that exclude large numbers of negative examples are favored over constraints that exclude few.
2. **H2** Specific spatial constraints are favored over general spatial constraints. This scheme implies that multiple positive examples are required to learn non-constant-length spatial constraints.
3. **H3** Straighter programs are favored over ones with more bends. This simplifies programs.

**Pruning** LearnDirAmount is a function that takes a spreadsheet, \( f \), a set of positive examples, \( P \), a pair of field 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. We provide pseudocode in the appendix.

Many descriptors can be ruled out when the geometric form of positive examples based on class membership. Given the set of deltas derived from the positive examples for \( i \) and \( j \), deltas 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.

Constant-amount descriptors can be omitted when deltas belong to classes 2 or 3. Single-match Kleene descriptors can be omitted when deltas belong to class 3.

We also employ a fourth heuristic, **H4**, that considers the cells above and to the left of positive examples to be implicit negative examples. We abandon these implicit negatives whenever a user tells us otherwise.

**Data Structures** The effect of a constraint on a spreadsheet is a set of cells. We represent node matches as a bit vector.
We also represent edge matches as a mapping from cell coordinates to bit vectors. In both cases, the length of the bit vector corresponds to the number of cells in the spreadsheet, and the coordinate of a given index in the bit vector is determined by a mapping function, \( f \). In the bit vector, a “1” at index \( i \) means that the constraint matched a cell at coordinate \( f(i) \) in the spreadsheet; a “0” indicates no match.

This representation is efficient for three reasons: 1) Matches are stored efficiently. For cell constraints, only one bit per cell is required. 2) Questions such as “does this constraint satisfy all positive examples?” can be answered in constant time using bit vector arithmetic. For this purpose, we bitwise-AND the effect of a constraint with the bit vector representing the positive examples. If the resulting bit vector equals the positive example bit vector, then answer is yes, otherwise no. 3) Operations requiring bit-counting can be done in \( O(#\text{bits set}) \) [Wegner 1960]. Bit-counting is used often in our heuristics.

We are able to evaluate partial programs efficiently during search using a dynamic programming scheme. Lastly, partial program evaluation can largely happen in parallel and before the synthesizer runs (i.e., offline). With a large set of constraints, this is especially beneficial, as the system can process constraints in the background before users invoke the program synthesizer.

4. Evaluation

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

1. It is possible to manually write \textsc{Flare} programs to perform a diverse set of extraction tasks?
2. Can \textsc{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 \textsc{FlashRelate} usable in the real world?

4.1 Benchmark Spreadsheets and Tasks

To evaluate \textsc{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 [Harris and Gulwani 2011] that examined 51 table-transformation programs from Excel user-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 in \textsc{Flare}. To round out our evaluation with more difficult tasks, we assembled a second set of benchmarks by searching the EUSES spreadsheet corpus [Fisher and Rothermel 2005] for spreadsheets with complex ad-hoc encodings. 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 variable-length spatial constraints that appear in our ground truth programs (“k*” in Fig. 15).

Expressiveness To evaluate the expressiveness of \textsc{Flare}, we manually wrote a correct \textsc{Flare} program for each benchmark and extracted the resulting output table. In the course of this effort, we found that the \textsc{Flare} language is expressive enough to extract the desired tuple from all of multidimensional data patterns we observed. We conclude that \textsc{Flare} is an effective tool for performing extraction tasks.

Synthesizer Experiments Using our ground truth programs, we compare the results of the \textsc{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 \textsc{FlashRelate}. The relational table extracted by the ground truth query represents the user’s desired extraction output; we call this the oracle. After each invocation of the \textsc{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 \textsc{FlashRelate} output differs, the 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 oracle is missing from the program output, it is a positive example; 2) if a tuple from the program output does not appear in the oracle, it is a negative example. We repeat this process until the synthesizer either finds a program whose output matches the oracle or we time-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 an heuristic to the \textsc{FlashRelate} algorithm. Experiment configurations are: 1) \textsc{H1}, \textsc{H2}, \textsc{H3}, \textsc{H4}, 2) \textsc{H1}, \textsc{H2}, and \textsc{H3}, 3) \textsc{H1} and \textsc{H2}, 4) \textsc{H1}, and 5) no ranking. For each ranking scheme, \textsc{H#} refers to the heuristic described in §3.2.4. We also run the experiment 30 times with all heuristics enabled where we also select examples randomly and average over the results, configuration \textsc{R}. The purpose of \textsc{R} is to test whether our oracle model of choosing examples sequentially is robust to user variation.
Figure 14: Performance across configurations. (a) Total benchmark synthesis times. Fewer seconds and a greater percentage is better. For example, none (no heuristics) succeeded on only 60% of the benchmarks before the timeout whereas H1, H2, H3, H4 succeeded on all but one. (b) Number of examples required. Fewer examples and a greater percentage is better. Generally, H1, H2, H3, H4 requires fewer examples than the other configurations. R is a variation of H1, H2, H3, H4 where examples are chosen randomly.

<table>
<thead>
<tr>
<th>pre (sec)</th>
<th>worst (sec)</th>
<th>pos + neg</th>
<th>k*</th>
<th>col</th>
<th>rec</th>
<th>cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>hg_ex2</td>
<td>0.1</td>
<td>5.7</td>
<td>3+2</td>
<td>0</td>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>hg_ex3</td>
<td>0.0</td>
<td>0.2</td>
<td>2+2</td>
<td>0</td>
<td>2</td>
<td>15</td>
</tr>
<tr>
<td>hg_ex4</td>
<td>0.0</td>
<td>0.2</td>
<td>2+1</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>hg_ex5</td>
<td>0.0</td>
<td>0.2</td>
<td>2+1</td>
<td>0</td>
<td>2</td>
<td>92</td>
</tr>
<tr>
<td>hg_ex6</td>
<td>0.0</td>
<td>0.3</td>
<td>5+2</td>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>hg_ex7</td>
<td>0.0</td>
<td>19.9</td>
<td>3+1</td>
<td>0</td>
<td>17</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex8</td>
<td>0.0</td>
<td>1.1</td>
<td>5+4</td>
<td>3</td>
<td>5</td>
<td>250</td>
</tr>
<tr>
<td>hg_ex9</td>
<td>0.0</td>
<td>0.1</td>
<td>1+1</td>
<td>0</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>hg_ex10</td>
<td>0.0</td>
<td>0.2</td>
<td>2+1</td>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>hg_ex11</td>
<td>0.0</td>
<td>0.3</td>
<td>3+3</td>
<td>2</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>hg_ex12</td>
<td>0.1</td>
<td>1.9</td>
<td>3+1</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex13</td>
<td>0.0</td>
<td>0.5</td>
<td>4+2</td>
<td>4</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>hg_ex14</td>
<td>0.0</td>
<td>0.4</td>
<td>1+1</td>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>hg_ex15</td>
<td>0.0</td>
<td>2.0</td>
<td>3+2</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex16</td>
<td>0.0</td>
<td>0.4</td>
<td>1+1</td>
<td>0</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>hg_ex17</td>
<td>0.0</td>
<td>4.0</td>
<td>1+1</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex18</td>
<td>0.6</td>
<td>6.4</td>
<td>4+1</td>
<td>0</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td>hg_ex19</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex20</td>
<td>0.0</td>
<td>4.0</td>
<td>1+1</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex21</td>
<td>0.0</td>
<td>4.0</td>
<td>1+1</td>
<td>0</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex22</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex23</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex24</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex25</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex26</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex27</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex28</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex29</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex30</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex31</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex32</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex33</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex34</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex35</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex36</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex37</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex38</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex39</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>hg_ex40</td>
<td>0.0</td>
<td>0.8</td>
<td>1+1</td>
<td>0</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 15: Benchmarks from [Harris and Gulwani 2011] are shown on the left while benchmarks from [Fisher and Rothermel 2005] are shown on the right. pre (sec) is the precomputation time; worst (sec) is the duration of the longest iteration; pos + neg is the number of positive and negative examples required for synthesis; # k* is the number of Kleene stars in the ground truth program, a measure of its complexity; # col is the number of columns in the output table; # rec is the number of records in the output table; # cells is the number of cells in the used range of the input spreadsheet. TO indicates that the synthesis algorithm timed-out after 10 minutes.

4.2 Experimental Setup

We evaluated synthesis on typical end-user hardware. Our test machine was a 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.

4.3 Results

Fig. 14a 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 since, 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 fields 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 common strings. 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. 14b shows the number of examples (iterations) required to synthesize the correct result. Users only need to provide 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 (for 13 total timeouts). Without ranking, the algorithm also requires many more examples: an average of 3.0 positive and 13.7 negative examples (medians: 3 positive; 9 negative). We conclude that without ranking, sometimes very general solutions are generated (matching too many cells) and thus numerous negative examples are required to sufficiently narrow the selection.

Random example selection (config. R in Fig. 14a & 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. 14a, 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. 14b, 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 statements about increasingly large proportions of the suite.

Table 15 summarizes the time and number of examples required by the algorithm with all rankings enabled as well as benchmark complexity.

Summary. We find that FlashRELATE is successful at synthesizing the appropriate program quickly and with a small amount of user effort. Furthermore, while one ranking scheme sacrifices a slightly higher speed for fewer iterations (R4), generally our ranking schemes reduce both the number of examples required and the time the user has to wait.

4.4 Case Study

What is the FlashRELATE user experience like? By a happy coincidence, 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. Their experiment planned to use photos to study participants’ perception of food healthiness. Participant responses would then be compared to ground-truth nutritional data. But the procedure ran into a roadblock: 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 in an analysis written in R. 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 [ProPublica 2014]. 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 [Weiss 2013].

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 of 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. The equivalent hand-written program took more than an hour to produce.

Finally, we performed post-processing (primarily the removal of asterisks) using FlashFill [Gulwani 2011]. FlashRELATE was specifically designed to work in spreadsheet-manipulating toolchains with tools like FlashFill, and this experience shows the utility of such a choice.

Summary. We learned two lessons from our case study. 1) Mentally mapping output tuples back to their original location in the spreadsheet is difficult; knowing this mapping helps inform which examples to give next. 2) For large spreadsheets, it is hard to know when a synthesized program extracts an entire spreadsheet. To address these concerns, we refined our user interface. If a user clicks on an output tuple, the set of cells that produced it will be highlighted. The user can quickly determine whether the inferred FLARE program extracts all of the input by clicking on a button that selects all of the output tuples. Finally, for each column, a histogram shows the counts for each distinct string matched for that field. Non-numeric strings in spreadsheets are often categor-
ical. Seeing whether the counts match one’s own intuition helps to track down the most useful positive examples.

5. 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, SXPath [Oro et al. 2010] includes spatial primitives in its queries, but does not attempt to synthesize programs in the query language from examples, as we do. While SILA [Oro and Ruffolo 2011] defines spatial abstractions like FLASHRELATE, it attempts to extract records from spatially structured data algorithmically, and not from examples. [Ferrara et al. 2012] and [Cafarella et al. 2011] 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 [Kushmerick et al. 1997]. There has been a wide variety of work in this area, ranging from supervised systems [Hsu and Dung 1998; Kushmerick et al. 1997; Muslea et al. 1999], semi-supervised systems [Chang and Lui 2001], to unsupervised systems [Crescenzi et al. 2001]. By contrast, our work enables users to induce wrappers interactively using examples.

Extracting Data from Spreadsheets  SENBAZURU [Chen and Cafarella 2013] 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.

[Cunha et al. 2009] focus on recovering the true relational schema from the spreadsheet data. We believe that the appropriate schema is task-dependent. FLASHRELATE crafts an extraction program that returns precisely those tuples that the user wants based a set of user-supplied examples.

OpenRefine [Verborgh and De Wilde 2013] and Wrangler [Kandel et al. 2011] 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. These tools are limited in their extraction capabilities with regard to spreadsheets.

Programming by Examples  The area of programming by examples [Lieberman 2001], which includes FLASHRELATE, is a large and growing area [Gulwani 2012] because of its potential to enhance productivity for end-users. Of these, the most closely related work is that of ProgFromEx [Harris and Gulwani 2011], which addresses tabular transforms for spreadsheets already in tabular format. FLASHRELATE addresses arbitrary transformations, employs heuristic search (ProgFromEx uses version-space algebras), and it needs only output examples (ProgFromEx needs input-output examples). Another recent PBE technology, Quicksilver [Lu et al. 2013], 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 domain-specific languages for describing data formats in text files and learning algorithms for inferring such formats using annotations [Fisher and Walker 2011]. The FlashExtract project supports a by-example framework for extracting data from text files [Le and Gulwani 2014]. The FlashExtract project showed how some of the linear abstractions for extracting data from text files can also be used for extracting data from other semi-structured documents including webpages and spreadsheets. By contrast, FLASHRELATE focuses on extracting data from spreadsheets and provides novel and more powerful two-dimensional geometric abstractions and learning algorithms for that purpose. FlashExtract cannot extract the motivating example in Fig. 1.

6. Conclusion

The flexibility of spreadsheets allows users to specify ad-hoc formats, providing flexibility at the expense 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.

A video demonstration of FLASHRELATE is available at: http://tinyurl.com/mh3bo3a
References


<table>
<thead>
<tr>
<th>Subject expertise?</th>
<th>Passing knowledge</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant?</td>
<td>Uninteresting</td>
<td>Compelling</td>
</tr>
<tr>
<td>Sound?</td>
<td>Flawed</td>
<td>Sound</td>
</tr>
<tr>
<td>Accept?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Strength of conviction?</td>
<td>Ambivalent</td>
<td>Adamant</td>
</tr>
</tbody>
</table>

Points for

Points against

Questions for authors

Other notes