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

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

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

Abstract

Spreadsheets store a tremendous amount of important data. One reason spreadsheets are so successful is that they are both easy to use and allow users great expressiveness in storing and manipulating their data. This flexibility comes at a price, as presentation elements are often combined with the underlying data model. As a result, many spreadsheets contain data in ad-hoc formats. These formats complicate the use of traditional relational tools which require data in a normalized form. Normalizing data from these formats is often tedious or requires programming, and often, a user may prefer the original presentation.

We describe an approach that allows users to easily extract structured data from spreadsheets without programming. We make two contributions. First, we describe a novel domain specific language called Flare that extends traditional regular expressions with spatial constraints. Second, we describe an algorithm called FlashRelate that can synthesize Flare programs from user-provided positive and negative examples. Using 43 benchmarks drawn both from a standard spreadsheet corpus and from Excel user-help forums, we demonstrate that correct extraction programs can be synthesized quickly from a small number of examples. Our approach generalizes to many data-cleaning tasks on semi-structured spreadsheets.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords Program Synthesis, Domain Specific Languages, Heuristic Search, Spreadsheets

1. Introduction

There are an estimated 500 million Microsoft Excel users worldwide [24]. It is thus not surprising that a tremendous amount of important data is stored in spreadsheets.

As with other document types such as text files and web pages, spreadsheets combine their data model and view. This flexibility gives the spreadsheet creator a large degree of freedom in encoding their data. It also complicates the use of powerful database tools (e.g., relational queries) which expect data in a particular form. Although spreadsheets are tabular and ostensibly *tables*, end-users may organize their data in *any* two-dimensional form that they find convenient.

This problem is widespread. Recent research suggests that the vast majority of spreadsheets available on the web cannot be trivially converted to database relations [6]. We argue that the cause is due to the multidimensionality of data. Representing multidimensional data in a two-dimensional spreadsheet presents a fundamental problem for users, since additional dimensions need to be projected into a 2D grid. Spreadsheet authors often solve this problem by crafting clever encodings of their data.

While expert users might recognize when database technologies might be more appropriate, they do so because these technologies facilitate automatic processing. In contrast, spreadsheets allow compact and intuitive visual representations of data better suited for human understanding. Since these representations vary from spreadsheet to spreadsheet, each unique encoding must be decoded if they are to be used with more powerful data tools.

The conflating of presentation information with data is commonplace in many domains. The recognition that non-experts are unlikely to embrace data management tools designed for automatic processing has lead to the development of numerous domain-specific technologies for data extraction. Scripting languages like Perl, Awk, and Python have been designed to support string processing in text files. Web technologies like XQuery, HTQL, XSLT, and SXPath can be used to extract data from webpages. *There are no such tools for spreadsheets.*

Example To make our contributions more concrete, we refer to the example showing in Fig. 1(a), a real spreadsheet taken from the EUSES corpus [11]. The spreadsheet shows timber harvests by country and year. It also 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 having the data in a long thin column, which would be harder to read.

Consider the steps that a programmer must undertake to compute the average harvest in 1950 from this representa-

	A	B	C	D	E	...	R
1		value	year	value	year		Comments
2	Albania	1,000	1950	930	1981		FRA 1
3	Austria	3,139	1951	3,177	1955		FRA 3
4	Belgium	541	1947	601	1950		
5	Bulgaria	2,964	1947	3,259	1958		FRA 1
6	Czech ...	2,416	1950	2,503	1960		NC

...

(a)

	A	B	C	D
1	Albania	1,000	1950	FRA 1
2	Albania	930	1981	FRA 1
...				
5	Austria	3,139	1951	FRA 3
6	Austria	3,177	1955	FRA 3
...				
9	Belgium	541	1947	
10	Belgium	601	1950	

...

(b)

Figure 1: (a) A semi-structured spreadsheet with two example tuples highlighted. 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. (b) An extracted relational table with the same two tuples highlighted as in Fig. 1a

tion. The spreadsheet author's encoding, while perhaps easier to read, makes this task a challenge. The first thought of such a programmer might be to convert the data into a form better suited to performing the computation, like the relational table shown in Fig. 1(b). One approach would be to use a scripting language like Perl or Visual Basic to define regular expressions that would match the elements in each column. For example, one could match country names with an alphanumeric pattern, years with 4-digit patterns, etc. Having done this, matches would be spatially collated to form tuples, and tuples collated to form a table. We show one such program in Fig. **??**.

Problem Statement The problem that we address in this paper is how to enable ordinary spreadsheet users, who do not have programming skills, to convert ad hoc data formats into relational ones. The strategy we use to achieve this goal has two parts. 1) *We present a domain-specific language (DSL) called* FLARE *for extracting relevant data from spreadsheets in a relational format.* 2) *We also present an algorithm called* FLASHRELATE *that automatically generates* FLARE *programs from a small number of user-provided positive and negative examples of tuples in the desired output table, greatly simplifying the process of automated data extraction.*

FLARE *Query Language* FLARE describes the regular geometric structure of ad-hoc encodings to map spreadsheet data into relational tables. The design of FLARE is inspired by scripting languages with regular expression capabilities, which have enabled developers with to extract relational data from text files, such as server logs. Spreadsheets represent another major source of semi-structured data. Without supporting code, regular expressions and other string-matching tools are not expressive enough to capture relational information encoded in spreadsheets. These structures, such as hierarchical data, become complex geometric structures when projected into a two-dimensional grid.

FLARE *converts ad hoc stuctures into relational ones so that relational tools can use them.* FLARE declaratively specifies the structure of ad-hoc encodings in the form of constraints. Data that matches these constraints are automatically converted into relational tables.

FLARE has two kinds of constraints. Constraints over the text *within* a spreadsheet cell are referred to as *cell constraints* and are composed of regular expressions. Cell constraints represent valid classes of values in a relational table, i.e., a relational column. Constraints *between* cells are referred to as *spatial constraints* and are composed of geometric relationships. Spatial constraints represent valid geometric relationships between values that belong in the same relational tuple.

Taken together, these two kinds of constraints form a directed constraint graph over the spreadsheet. Vertices consist of cell constraints while edges consist of spatial constraints. The cells matched by a traversal of this graph produce a set of relational tuples in the desired output table. We show a sample FLARE query and corresponding constraint graph in Fig. 3. Intuitively, one can think of this graph as an invariant geometric structure that, when translated over a spreadsheet, indicates which cells form a relational tuple.

FLASHRELATE *Synthesis Algorithm* Programmatic solutions to data extraction suffer two key limitations. First, the expertise required to use these tools is often particular to specific document types. Second, and more significantly, they require 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. As a result, users are either unable to leverage access to rich data or have to resort to manual copy-and-paste, which is both time-consuming and error-prone.

FLASHRELATE is a complementary technology that bridges the expertise gap between ordinary end-users and sophisticated new query languages like FLARE. FLASHRELATE infers the correct set of constraints from a small number of examples given by the user and outputs an appropriate FLARE program.

Example 1: Returning to our example in Fig. 1, we show how a FLARE program can concisely extract the necessary relational table. Recall that the user would like to

```
Function Extract() As Collection
  ...

  Set Tuples = New Collection     'Results
  rYear.Pattern = "^19[0-9]{2}$"  'Year Patt.
  rValue.Pattern = "^[0-9]+$"     'Value Patt.

  'Search for Tuples
  For Each ws In Worksheets
    For Each cell In ws.UsedRange
      x = cell.Column
      y = cell.Row
      x_rt = x + 1

      If rYear.Test(cell.Value) _
        And rValue.Test(_
          ws.Cells(y, x_rt).Value) Then
        Dim tupleCoords
        tupleCoords = Array(ws.Index, x, _
                            y, x_rt, y)
        Tuples.Add (cellCoords)
      End If
    Next
  Next

  Extract = Tuples
End Function
```

Figure 2: A Visual Basic program that performs the desired extraction in Example 1. Variable declarations were omitted for space reasons. The equivalent FLARE program, shown in Fig. 3a, declaratively specifies the extraction and is much simpler.

compute the average timber harvest for 1950. The FLARE program shown in Figure 3 performs this extraction. Once the data is in the desired form, the desired result can be computed trivially with the following SQL query: `SELECT AVG(column_1) FROM FlareTable WHERE column_2 = 1950`.

Despite the difficulty of computing the result from the original structure, a typical person would have no trouble understanding the spreadsheet. 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 timber value. Country names are always located to the far left. Although other geometric information may be used to perform the same task, these three invariants are sufficient to extract the desired tuple. FLARE makes use of these intuitive geometric concepts to concisely express the desired extraction.

In Figure 3(a), `Node` constraints identify the two columns in the desired relation, (1) the value, and (2) the year, labeled with column IDs 1 and 2, respectively. `Node` constraints are based on regular expressions. The first `Node` constraint includes an extra contextual constraint, called an *anchor*, that requires the presence of a second, contextual cell. In this program, that additional cell must contain the string "value" somewhere above `Node` 1 in the input spreadsheet.

Note that `Node` 2 is a strict superset of `Node` 1. How then, are "value" cells to be distinguished from "year" cells? `Edge`

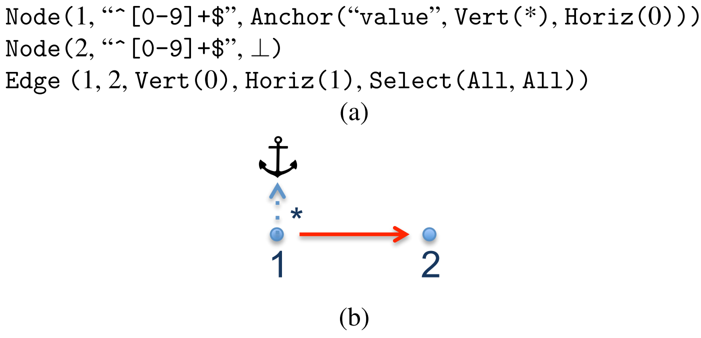
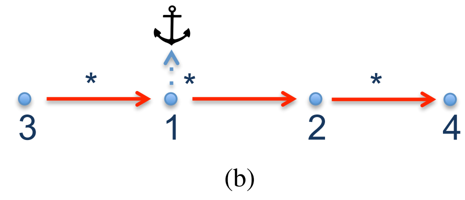


Figure 3: (a) FLARE program for first example extraction task with (b) schematic illustration. `Node` constraints are shown as dots, `Edge` constraints are shown as solid arrows, and `Anchor` constraints are shown with a dashed arrow and anchor symbol. `Edge` s and `Anchors` of non-constant length are labeled with a Kleene star. Node numbers correspond to attribute IDs in the desired relational tuple.

```
Node(3, "^[a-zA-Z ]+$", ⊥)
Node(4, "^[a-zA-Z ]+$", ⊥)
Edge (3, 1, Vert(0), Horiz(*), Select(All, All))
Edge (2, 4, Vert(0), Horiz(*), Select(All, All))
```
(a)



(b)

Figure 4: (a) A set of additional constraints added to Figure 3a; (b) schematic illustration.

constraints provide the additional context required by describing the spatial relationships between `Node`s. Year values must be one cell to the right of harvest values.

Example 2: Another task might be to find comments for countries reporting harvests after 1960. This new extraction, displayed as a set of additional constraints appended to our original program, is shown in Figure 4. The extracted table corresponds to the one shown in Fig. 1b. The task can be completed by running the following SQL query against the extracted table: `SELECT DISTINCT column_3,column_4 FROM FlareTable WHERE column_2 > 1960`.

Contributions This paper makes the following contributions:

- We describe a domain-specific query language, FLARE, that allows extraction of relational data from two-dimensional semi-structured spreadsheets (see §3). 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 (see §4). FLASHRELATE's effectiveness (in speed and number of required examples) is dictated by its efficient constraint search.
- We empirically evaluate the expressiveness of the FLARE and the effectiveness of FLASHRELATE against a set of 43 real-world spreadsheets drawn from the EUSES corpus and from Excel user help forums [11, 16]. We show that the FLASHRELATE algorithm is able to synthesize correct FLARE programs from examples in all but one case. Only a small number of examples are required (see §5). FLASHRELATE rarely takes longer than 2 seconds.

2. Related Work

Much recent research considers the problem of extracting structured data from unstructured or semi-structured sources, including documents on the web (e.g, [4, 9, 21, 22]).

Extracting Data from the Web Another important related body of work focuses on extracting relational data from data on the web. SXPath [22] is a query language, like FLARE, that uses a combination of spatial relations and path expressions to extract data from complex web documents. SXPath uses intuitive spatial relations in describing patterns to avoid queries that involve complex, non-intuitive deep path expressions. SXPath, like FLARE, includes spatial primitives in its queries, but does not attempt to synthesize programs in the query language from examples, as we do. SILA [21] defines a spatial DOM abstraction (PDOM) and proposes an algorithm for automatically extracting data records from deep web pages based on the PDOM. While, like FLASHRELATE, SILA attempts to extract records from spatially structured data, it does so algorithmically, and not by defining a domain-specific query language. For overviews of the range of approaches taken to web data extraction, see Ferrera et al. [9] and Cafarella et al. [4]

Extracting Data from Spreadsheets The research most closely related to FLASHRELATE is the concurrently-developed SENBAZURU project, which, like FLASHRELATE, seeks to extract relational data from spreadsheets [5, 6]. While the project goals are similar, the approach taken by FLASHRELATE is very different. SENBAZURU attempts to automatically infer *hierarchical structure* in spreadsheets by creating a classifier that identifies data frames in the document and another classifier that infers the intended hierarchy based on a set of predefined features. In contrast, FLASHRELATE uses positive and negative output examples to synthesize a program in a domain-specific language of our own design. FLASHRELATE can be used to perform *arbitrary* extraction tasks from *arbitrary* spreadsheets, provided that regular syntactic and spatial structure is present.

Cunha et al. [7] also consider extracting relational data from spreadsheets, but their focus is on recovering the true relational schema from the spreadsheet data. We believe that a user might have in mind many different task-dependent schemas for a single spreadsheet, as the example in the introduction illustrates. Instead, FLASHRELATE crafts an extraction program that returns precisely those tuples that the user *wants* based a set of user-supplied examples.

Query Synthesis by Examples The *view synthesis problem* [8, 26] aims to find the most succinct and accurate query for a given database view. There are two key differences with our work: (i) View synthesis techniques infer a relation from multiple single-dimensional relational tables, while we infer a relation from a single, two-dimensional semi-structured spreadsheet. Such spreadsheets are often used to encode multiple dimensions in ad-hoc ways. (ii) View synthesis techniques infer a relation from a large representative example view, while we infer a transformation from a set of few example rows, an important usability aspect for end-users.

Programming by Examples The area of programming by examples [18] (PBE) is gaining renewed interest [14] because of its revolutionary potential to enhance productivity of millions of end-users. Gulwani et al. have developed programming-by-example techniques for automating repetitive data manipulation tasks related to structured spreadsheet tables [15]. These include syntactic string transformations [13], semantic string transformations [25], and table layout transformations [16]. Of these, the most closely related work is that of [16]. The following are some key differences with our work: (i) We address a different class of spreadsheet tasks, namely transforming semi-structured data into structured relational data. (This facilitates application of prior work on manipulating structured relational data using input-output examples.) For instance, [16] cannot handle any of the transformation tasks associated with our new benchmark examples. (ii) The synthesis techniques used are completely different. Prior work uses a class of techniques called *version-space algebras*, while we perform heuristic search. (iii) The user interaction is quite different. Prior work takes as input multiple input-output examples, while our technique takes as input multiple positive/negative examples of tuples in the desired output table.

Quicksilver [19] is another recent PBE technology for structured relational data. It synthesizes relational algebra queries over strictly relational tables, while we focus on a different class of spreadsheet tasks, namely extracting relational tuples from semi-structured spreadsheets. QuickSilver cannot handle any of the transformation tasks associated with our benchmark examples.

Manipulation of Semi-Structured Data 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 [10]. The learned format can then be used by programmers to implement custom data-analysis tools. PADS focuses on parsing and manipulating semi-structured data in text files or log files while we focus on semi-structured data in spreadsheets.

Learning Theory Angluin presented a classical algorithm for learning regular expressions or finite state automatas from positive and negative examples [2]—an *equivalence oracle* produces a counterexample that exhibits disequivalence of the current hypothesis with the intended one, and a *membership oracle* labels that counterexample as either positive or negative. Our work learns a more sophisticated concept, namely a two-dimensional query involving regular expressions from positive/negative examples. In our case, it is the end-user who plays the role of both equivalence and membership oracles, providing a positive or negative tuple row(s) in each interaction.

Two-Dimensional Grammars The formal methods community has done a lot of work on defining *picture grammars* [12, 23], which extend classical grammars for generating strings into two-dimensional space, using concepts like two-dimensional regular expressions [3, 20] and automata [17]. These works focus on theoretical analysis of two-dimensional grammars, specifically decidability and complexity of classical problems for these novel languages. In contrast, a FLARE program is specifically designed for matching relational tuples. Furthermore, we address the problem of synthesizing the desired program from few positive/negative examples.

Header Inference [1] describes a system that automatically infers header information in spreadsheets by exploiting special layout or formatting attributes, such as sub-headers, footers, filler cells (blank cells or cells with some special characters to aid visual readability of table content). Our work also leverages the presence of such layout or formatting constraints in spreadsheets; however our use of this information is not heuristic in nature—users provide examples for tuple extraction and we learn a specific pattern-based logical query for accomplishing the intended task.

3. Flare Language

The goal of a FLARE program is to transform a semi-structured spreadsheet I into an n-ary relational table. The syntax and formal semantics of FLARE are shown in Fig. 5. A spreadsheet I is a two-dimensional collection of strings. A *cell* c is a pair of x and y coordinates that can be used to index I (as in $I[c]$). We use the term $\texttt{Cells}(I)$ to denote all cells in the used range of the spreadsheet I. An n-ary relational table is a set of n-ary tuples of strings. Each such tuple can also be represented as a map from tuple attribute indices $\{1, \ldots, n\}$ to spreadsheet strings (as in $\{1 \mapsto s_1, 2 \mapsto s_2, \ldots, n \mapsto s_n\}$).

A FLARE program ρ consists of a set of n nodes and a set of directed edges that form a tree shape. These nodes and edges are labeled with descriptions that form the criteria for extracting relational tuples from the input sheet. Each node in P corresponds to a unique tuple attribute index i from the output relational table. It is associated with a *cell constraint* α (denoted by $\texttt{CellConstraint}(i)$) that is a Boolean constraint over spreadsheet cells. Each directed edge in P corresponds to an ordered pair of tuple attribute indices j and k from the output relational table. Each such edge is associated with two kinds of constraints: (a) a *spatial constraint* β (denoted by $\texttt{SpatialConstraint}(j, k)$), which is a Boolean constraint over an ordered pair of spreadsheet cells and checks horizontal/vertical orientation of the two cells. (b) a *select constraint* γ (denoted by $\texttt{SelectConstraint}(j, k)$), which filters a set of cells with respect to another cell based on some distance related tags. We discuss the precise syntax and semantics of these different kinds of constraints later in this section. We use the term $\texttt{Root}(\rho)$ to denote the tuple attribute index corresponding to the root node in the tree structure formed by the nodes and edges in ρ.

In order to define a constructive semantics for a FLARE program ρ, we introduce a recursive function $F(j, k, c)$ that takes as input an edge identifier (i.e., a pair of tuple attribute indices j and k) and a cell c and returns a set of m-tuples, where m is the number of nodes reachable from the corresponding edge.

$F(j, k, c)$ is computed in three steps: (i) Compute the set C' of all cells that simultaneously satisfy both the spatial relationship to c with $\texttt{SpatialConstraint}(j, k)$ and the cell pattern $\texttt{CellConstraint}(k)$. (ii) Filter C' to that subset C'' that satisfies the $\texttt{SelectConstraint}(j, k)$ relationship with c. (iii) For each $c'' \in C''$, compute the cross product of singleton c'' with the result of the recursive invocation of F along each outgoing edge from nodes with tuple attribute index k and cell c'' (i.e., $F(k, k_i, c'')$, where k_i belongs to the set of children of node k (denoted by $\texttt{Children}(\rho, k)$)), and return their union.

The execution of a program ρ on an input spreadsheet I proceeds as follows: (i) Compute the set C of all cells that satisfy the cell constraint $\texttt{CellConstraint}(\ell)$, where ℓ is the tuple attribute index associated with the root node of program- ρ. (ii) For each $c \in C$, compute the cross product of singleton c with the result of recursive invocations of F along the outgoing edges from the root node and cell c (i.e., $F(\ell, k_i, c)$), and return their union.

A cell constraint $\texttt{Cell}(r, \text{\textanchor})$ consists of a regular expression r, namely a Boolean constraint over strings, and an *anchor constraint* ⚓, namely a Boolean constraint over cells. An anchor constraint $\texttt{Anchor}(r, \beta)$, which consists of a regular expression r and a spatial constraint β, asserts that there exists a cell c' whose content matches r and that is related to the argument cell using the spatial constraint β. Anchor constraints can be thought of as a special case of the spatial

$$
\begin{array}{rcl}
\text{Program } \rho & := & \{\mathcal{N}_i\}_i \cup \{\mathcal{E}_i\}_i \\
\text{Node } \mathcal{N} & := & \texttt{Node}(j, \alpha) \\
\text{Edge } \mathcal{E} & := & \texttt{Edge}(j, k, \beta, \gamma) \\
\text{Cell Constraint } \alpha & := & \texttt{Cell}(r, ⫰) \\
\text{Spatial Constraint } \beta & := & \texttt{Spatial}(v, h) \\
\text{Vertical Constraint } v & := & \texttt{Vert}(q) \mid \top \\
\text{Horiz. Constraint } h & := & \texttt{Horiz}(q) \mid \top \\
\text{Quantity } q & := & \ldots \mid \text{-2} \mid \text{-1} \mid 0 \mid 1 \mid 2 \mid \ldots \\
 & \mid & \texttt{-}* \mid * \\
\text{Select Constraint } \gamma & := & \texttt{Select}(T_1, T_2) \\
\text{Type } T & := & \texttt{All} \mid \texttt{NearX} \mid \texttt{NearY} \\
 & \mid & \texttt{FarX} \mid \texttt{FarY} \\
\text{Anchor Constraint } ⫰ & := & \texttt{Anchor}(r, \beta) \mid \bot
\end{array}
$$

(a)

Cell:	$(\mathbb{N}, \mathbb{N})$
Spreadsheet:	Cell $\rightarrow$ String
Program:	Spreadsheet $\rightarrow$ n-tuple set
Cell/Anchor Constraint:	Cell $\rightarrow$ Bool
Spatial/Vert/Horiz Constraint:	(Cell, Cell) $\rightarrow$ Bool
Select Constraint:	(Cell, Cell set) $\rightarrow$ Cell set

(b)

$$
\begin{array}{rcl}
\texttt{Node}(j, r, ⫰) & \equiv & \texttt{Node}(j, \texttt{Cell}(r, ⫰)) \\
\texttt{Edge}(j, k, v, h, \gamma) & \equiv & \texttt{Edge}(j, k, \texttt{Spatial}(v, h), \gamma) \\
\texttt{Anchor}(r, v, h) & \equiv & \texttt{Anchor}(r, \texttt{Spatial}(v, h)) \\
\texttt{All} & \equiv & \texttt{Select(All, All)}
\end{array}
$$

(c)

$$
\begin{array}{rcl}
[\![\rho]\!]\, I & = & \{\{(\ell \mapsto I[c])\} \cup \sigma_1 \cup \ldots \cup \sigma_d \mid c \in C, \sigma_i \in F(\ell, k_i, c)\} \\
 & & \text{where } C = \{c \mid [\![\alpha]\!]\, c,\ c \in \texttt{Cells}(I)\},\ \ell = \texttt{Root}(\rho) \\
 & & \alpha = \texttt{CellConstraint}(\ell),\ \{k_1, \ldots, k_d\} = \texttt{Children}(\rho, \ell) \\
F(j, k, c) & = & \{\{(k \mapsto I[c''])\} \cup \sigma_1 \cup \ldots \cup \sigma_d \mid c'' \in C'', \sigma_i \in F(k, k_i, c'')\} \\
 & & \text{where } C'' = [\![\gamma]\!]\,(c, C'),\ C' = \{c' \mid [\![\beta]\!]\,(c, c') \wedge [\![\alpha]\!]\, c'\}, \\
 & & \alpha = \texttt{CellConstraint}(k),\ \beta = \texttt{SpatialConstraint}(j, k) \\
 & & \gamma = \texttt{SelectConstraint}(j, k),\ \{k_1, \ldots, k_d\} = \texttt{Children}(\rho, k)
\end{array}
$$

$$
\begin{array}{rcl}
[\![\texttt{Cell}(r, ⫰)]\!]\,(I, c) & = & [\![r]\!]\, I[c] \ \wedge\ [\![⫰]\!]\,(I, c) \\
[\![\texttt{Spatial}(v, h)]\!]\,(c, c') & = & [\![v]\!]\,(c, c') \wedge [\![h]\!]\,(c, c') \\
[\![\texttt{Horiz}(q)]\!]\,(c, c') & = & (\text{fst}(c) - \text{fst}(c')) \in \texttt{Range}(q) \\
[\![\texttt{Vert}(q)]\!]\,(c, c') & = & (\text{snd}(c) - \text{snd}(c')) \in \texttt{Range}(q) \\
 & & \text{where } \texttt{Range}(\texttt{-}*) = \{-1, -2, -3, \ldots\} \\
 & & \qquad \texttt{Range}(*) = \{1, 2, 3, \ldots\} \\
 & & \qquad \texttt{Range}(i) = \{i\} \\
[\![\top]\!]\,(c, c') & = & \mathit{true} \\
[\![\texttt{Select}(T_1, T_2)]\!]\,(c, C) & = & [\![T_2]\!]\,(c, [\![T_1]\!]\,(c, C)) \\
[\![\texttt{All}]\!]\,(c, C) & = & C \\
[\![\texttt{NearX}]\!]\,(c, C) & = & \underset{c' \in C}{\text{argmin}}\, |\text{fst}(c) - \text{fst}(c')| \\
[\![\texttt{Anchor}(r, \beta)]\!]\, I, c & = & \exists c'\ ([\![r]\!]\, I[c'] \wedge [\![\beta]\!]\,(c, c')) \\
[\![\bot]\!]\,(I, c) & = & \mathit{true} \\
[\![r]\!]\, s & = & \mathit{true} \text{ iff regular expression } r \\
 & & \text{matches string } s
\end{array}
$$

(d)

Figure 5: (a) Syntax and (b) types of a FLARE program ρ. We refer to $*$ as a Kleene star. (c) For notational convenience, we use the shorthand syntax on the left to denote the expressions on the right. (d) Semantics of a FLARE program ρ on spreadsheet I. x, y, and z are Coordinates. c is a Cell. C is a set of Cells. a and b are integers, and v is a Boolean.

and cell constraints. Anchor constraints serve as a Boolean predicate but do not extract a column in the output tuple.

A spatial constraint $\texttt{Spatial}(v, h)$ consists of a pair of constraints: a vertical constraint v, and a horizontal constraint h. Both forms are Boolean constraints over a pair of cells and check whether the two cells have the specified vertical and horizontal orientation, respectively. Both forms are parameterized by a quantity argument q. Kleene ("*") quantities can match *any* cell in a particular direction, whereas constant quantities only match cells in a particular location.

The select constraint γ consists of a pair of enumerated `Type` tags that control the behavior of Kleene quantities. A tag `Type` is either `All`, `NearX`, `NearY`, `FarX`, or `FarY`, each of which is a filter over a set of cells C with respect to another cell c. In particular, the tag `NearX` selects those cells from C that have the shortest horizontal distance from the cell c. Other tags are defined similarly. A select constraint will typically contain at most one constraint in each direction (i.e., at most one of `NearX` and `FarX`, and at most one of `NearY` and `FarY`).

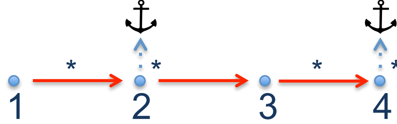
3.1 Program Execution

In this section, we walk through the execution of a FLARE program and discuss our early experience writing programs for extracting data from spreadsheets. Because FLARE builds on regular expressions, the additional concepts we present are relatively easy to learn.

For our example, we return to the input spreadsheet shown in Fig. 1a and the desired relational table output

```
Node(1,"^[a-zA-Z ]+$",⊥)
Node(2,"^[0-9]+$",Anchor("value",Vert(∗),⊤))
Node(3,"^19[0-9]2$",⊥)
Node(4,"^(FRA [1-3]|NC|)$",
      Anchor("Comments",Vert(∗),⊤))
Edge(1,2,Vert(0),Horiz(∗),Select(All,All))
Edge(2,3,Vert(0),Horiz(1),Select(All,All))
Edge(3,4,Vert(0),Horiz(∗),Select(All,All))
```

(a) Shown as a FLARE program.



(b) Shown as a tree.

Figure 6: The program for the running example.

shown in Fig. 1b. A FLARE program that accomplishes this is shown in Fig. 6a and is illustrated schematically in Fig. 6b.

The output of a FLARE program should be thought of as the simultaneous solution of its set of constraints on a spreadsheet, however an operational description of the interpreter helps to further understanding.

FLARE programs must be tree-shaped and all nodes must be reachable via directed links from the root. This formalism makes the recursive execution procedure described earlier well defined. Every node except the root node is paired with one edge for which that node is the destination. Each node corresponds a constraint over the contents of cells in a column of the desired relational table. Each edge corresponds to a spatial relationship between the cells of two output columns. The interpreter loop processes each of these constraint pairs starting from the tree root's cell constraint. When evaluating the root constraint, all cells in the spreadsheet are considered. Cells *filtered* by the root constraint are candidates for inclusion in the root column (which is not necessarily the first column). Note that inclusion of a cell in the final output is predicated on all subsequent constraints for each cell being met.

For subsequent nodes and edges, the interpreter recursively returns a new set of cells satisfying each additional constraint pair. The result become candidates for the attribute named by the constraint pair's ID.

In our example, the following root node is first evaluated.

```
Node(1,"^[a-zA-Z ]+$",⊥)
```

The cell constraint filters the entire input spreadsheet by applying the regular expression, `^[a-zA-Z ]+$`, and then by ensuring that all cells that match the expression also have neighbors that satisfy the anchor expression, $\perp$. The $\perp$ anchor is a special case meaning that no anchor expression is given, thus evaluating the $\perp$ anchor for any cell in any spreadsheet always returns true. After evaluating the root

	A	B	C	D	E	...	R
1		value	year	value	year		Comments
2	Albania	1,000	1950	930	1981		FRA 1
3	Austria	3,139	1951	3,177	1955		FRA 3
4	Belgium	541	1947	601	1950		
5	Bulgaria	2,964	1947	3,259	1958		FRA 1
6	Czech ...	2,416	1950	2,503	1960		NC

...

Figure 7: The set of cells (highlighted) from I that satisfy the root constraint in the running example.

node, the output column with attribute ID 1 contains the cells highlighted in Fig. 7. Our intent was to select the cells belonging to the first column, i.e., country names. Clearly, this constraint is insufficiently precise; however, additional constraints will eliminate these unwanted cells.

The cell constraint for column 2 includes an anchor in addition to its regular expression:

```
Node(2,"^[0-9]+$",Anchor("value",Vert(∗),⊤))
```

First, cells that do not match "`^[0-9]+$`" are eliminated (note that FLARE matches regular expressions against the raw text values in a spreadsheet before Excel's formatting rules are applied). Next, only the cells that satisfy the anchor predicate are returned. This anchor states that a cell containing the text "`value`" must appear somewhere above the match.

The following edge joins columns 1 and 2:

```
Edge(1,2,Vert(0),Horiz(∗),All)
```

The spatial constraint states that cells identified by the cell constraint for attribute 2 must be *anywhere to the right* and *not above or below* the cells identified by the cell constraint for attribute 1. This eliminates the unwanted cells for attribute 1 described earlier.

The interpreter processes all remaining constraints in the same manner. Every parent cell in the program tree is *joined* with each of its child cells, thus yielding a relational n-tuple. This program yields the output shown in Fig. 1b.

The worst-case complexity of a FLARE program, in terms of the number of cells, n, and number of columns, c, is $O(n^c)$, when all edges use Kleene stars. In practice, complexity is low. Tuples with very large numbers of columns are rare. Encodings rarely require *only* variable-length components. Lastly, the FLARE interpreter can search all alternatives efficiently in parallel, since extraction operations can be expressed as pure functions on an immutable grid. As a point of reference, the FLARE synthesizer may call the FLARE runtime many thousands of times during its operation. As we show in §5 , FLASHRELATE is nonetheless quite fast.

4. FlashRelate Synthesis Algorithm

The goal of the FLASHRELATE algorithm, shown in Fig. 8a-e, is to generate a FLARE program that is consistent with the positive and negative examples supplied by

SYNTH(I, P, N)

1 **for each** column index i
2 $\quad$ A$\mathcal{N}[i]$ = Learn$\mathcal{N}(I, P, i)$
3 **for each** pair of column indices i, j such that $i \neq j$
4 $\quad$ A$\mathcal{E}[i, j]$ = Learn$\mathcal{E}(I, P, i, j)$
5 **return** SEARCH$(\emptyset, N, \mathrm{A}\mathcal{N}, \mathrm{A}\mathcal{E})$

(a)

SEARCH$(C_{\mathcal{P}}, N, \mathrm{A}\mathcal{N}, \mathrm{A}\mathcal{E})$

1 **if** $|C_{\mathcal{P}}|$ = NUMCOLS
2 $\quad$ **if** $\bigcup_{(\alpha,\beta) \in C_{\mathcal{P}}}$ Negate$(\alpha) \cup$ Negate$(\beta) = N$
3 $\quad\quad$ **return** $C_{\mathcal{P}}$
4 $\quad$ **else return** FAILURE
5 **else**
6 $\quad$ $C_{\mathcal{E}} = \{\mathcal{E} \mid \mathcal{E} \in C_{\mathcal{P}}\}$
7 $\quad$ $pairs = \{(\mathcal{N}, \mathcal{E}) \mid \mathcal{N} \in \mathrm{A}\mathcal{N}[i], \mathcal{E} \in \mathrm{A}\mathcal{E}[i, j]$
$\quad\quad$ s.t. $G' = (V, C_{\mathcal{E}} \cup \{\mathcal{E}\})$ is loop-free
$\quad\quad$ and $i, j \in [1 \ldots$ NUMCOLS$]\}$
8 $\quad$ $pairs'$ = RANK$\mathcal{P}(pairs)$
9 $\quad$ $k = 0$
10 $\quad$ **while** $k < |pairs'|$
11 $\quad\quad$ $C'_{\mathcal{P}}$ = SEARCH$(C_{\mathcal{P}} \cup \{pairs'[k]\}, N)$
12 $\quad\quad$ **if** $C'_{\mathcal{P}}$ = FAILURE
13 $\quad\quad\quad$ $k = k + 1$
14 $\quad\quad$ **else**
15 $\quad\quad\quad$ **return** $C'_{\mathcal{P}}$
16 $\quad$ **return** FAILURE

(b)

LEARN$\mathcal{N}(I, P, i)$

1 A = set of predefined constraints
2 A$\mathcal{N} = \emptyset$
3 **for each** constraint $\alpha \in A$
4 $\quad$ **if** $\forall$ tuples $p \in P, [\![\alpha]\!]\ (I, p[i]) = true$
5 $\quad\quad$ A$\mathcal{N}$ = A$\mathcal{N} \cup \{\mathtt{Node}(i, \alpha)\}$
6 **return** A$\mathcal{N}$

(c)

LEARN$\mathcal{E}(I, P, i, j)$

1 A$\mathcal{E} = \emptyset$
2 V = LEARNQUANT$(I, P, i, j, \textsc{true})$
3 H = LEARNQUANT$(I, P, i, j, \textsc{false})$
4 **for each** $\{v, h \mid v \in V, h \in H\}$
5 $\quad$ $\beta = \mathtt{Spatial}(v, h)$
6 $\quad$ **for each** $\gamma \in$ ENUMSELECT
7 $\quad\quad$ A$\mathcal{E}$ = A$\mathcal{E} \cup \{\mathtt{Edge}(i, j, \beta, \gamma)\}$
8 **return** A$\mathcal{E}$

(d)

LEARNQUANT$(I, P, i, j, vert)$

1 $\Delta = \emptyset$
2 $Q = \emptyset$
3 $\lambda = \mathtt{Vert}$ **if** $vert$ **otherwise** $\mathtt{Horiz}$
4 ϕ = snd **if** $vert$ **otherwise** fst
5 **for each** $p \in P$
6 $\quad$ $\Delta = \Delta \cup \{\lambda(\phi(p[j]) - \phi(p[i]))\}$
7 **if** $\forall \delta_1, \delta_2 \in \Delta, \delta_1 = \delta_2$
8 $\quad$ Q = any $\delta \in \Delta$
9 **if** $\forall \delta \in \Delta, \delta \geq 0$
10 $\quad$ $Q = Q \cup \{\lambda(*)\}$
11 **else if** $\forall \delta \in \Delta, \delta \leq 0$
12 $\quad$ $Q = Q \cup \{\lambda(-*)\}$
13 **else return** FAILURE
14 **return** Q

(e)

Figure 8: FLASHRELATE's program synthesis procedures. (a) is the top-level procedure. (b) is the program search procedure. (c) is a subroutine for learning node constraints from positive examples. (d) is a subroutine for learning edge constraints from positive examples. ENUMSELECT is a macro that enumerates the set of all possible select constraints. (e) learns the direction and amount of spacing between examples.

the user. A FLARE program is a set of cell and spatial constraints, which as we described earlier, represent vertices and edges in a constraint graph. We thus frame the problem of finding a correct program as a search over all valid sets of these constraints that satisfy the positive and negative examples.

While FLARE can produce programs that represent arbitrary disconnected directed acyclic graphs (DAGs), we limit the FLASHRELATE's capabilities to producing connected directed trees. We do this for two reasons: 1) all of the encodings that we have examined thus far can be encoded in the tree-shaped subset of FLARE, and 2) this formulation allows us to reduce the problem of finding a satisfying program to that of finding a spanning tree. Thus FLASHRELATE's program search procedure, shown in Fig. 8b, 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 desires 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 created by users, and 2) speed the search by

4.1 Definitions

We use the following terms in Fig. 8. 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 undesired program outputs, from this point on referred to as *negative examples*. NUMCOLS is defined as the number of attributes in a tuple in P.

Let V be the set of attribute IDs, where each ID represents a position in the desired relational output tuple in P. Let E be the complete set of directed edges, each edge referred to by a pair of attribute IDs (i, j). Let $G = (V, E)$ be the complete digraph over the attributes in P. The following two expressions also signify a shorthand for the negative tuples excluded by a cell constraint and spatial constraint, respectively:

$$\text{Negate}(\alpha) \equiv \{n \in N \mid n \notin [\![\alpha]\!]\,(I, c), \forall c \in I\}$$
$$\text{Negate}(\beta) \equiv \{n \in N \mid n \notin [\![\beta]\!]\,(c, c'), \forall c, c' \in I, c \neq c'\}$$

4.2 Algorithm

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

1. Using positive examples, determine which cell constraints are possible (Fig. 8c). See §4.2.1 .
2. Using positive examples, determine which spatial constraints are possible (Fig. 8d). See §4.2.2 .
3. Identify a combination of cell and spatial constraints that excludes all negative examples (Fig. 8b). See §4.2.3 .

We discuss a number of implementation details that make FLASHRELATE's search procdure efficient in §4.2.4 .

Example We discuss a single round of synthesis using the algorithm shown in Fig. 8 by way of a running example, the spreadsheet shown in Fig. 9a. The desired relational output is shown in Fig. 9b.

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 new positive example and calls FLASHRELATE again (step 2).
   - If the extracted table contains an undesired tuple, the user provides a new negative example and calls FLASHRELATE again (step 2).

Note that while users must provide at least one new positive or negative example during each round of the procedure described above for the algorithm to synthesize an improved FLARE program, they are not limited to providing only a single new example.

Suppose our user starts the process by providing the following positive example to the synthesizer, the first tuple in the desired table:

Deerfield	130 Central St.	Joe M.

Note that this positive example also encodes the following information, a map from each tuple attribute index to spreadsheet coordinates.

$$1 \longrightarrow (1,2) \quad 2 \longrightarrow (2,2) \quad 3 \longrightarrow (3,2)$$

FLASHRELATE uses both representations. The first representation describes the contents of a tuple; the second representation describes the spatial relationships between attributes in a tuple.

4.2.1 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 [13] in research for learning string programs.

Regular expression learning algorithms are outside the scope of this paper, but the topic is well-studied [2]. Thus, the primary focus in this paper is in learning geometric patterns (see §4.2.2). FLASHRELATE is designed to work with any 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 attribute 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. 8a calls Learn$\mathcal{N}$ (Fig. 8c) for each the set of strings corresponding to each attribute ID i in the set of positive examples. Learn$\mathcal{N}$ eliminates those constraints that do not match all of the strings associated with i.

Returning to our example, if the synthesizer is given the set of regular expressions shown in Fig. 10, Learn$\mathcal{N}$ returns the constraints shown in Fig. 11.

4.2.2 Step 2: Determine spatial constraints

Line 3 in Fig. 8a calls Learn$\mathcal{E}$ (Fig. 8d) for each pair of attribute indices (i, j) in P. Learn$\mathcal{E}$ finds all possible spatial constraints that satisfy the observed spatial layout between columns from positive examples.

For each positive example p, Learn$\mathcal{E}$ calculates the delta for the cells (lines 2-3) $p[i]$ and $p[j]$. Each *delta* is the distance in either the horizontal or vertical direction, expressed as a number of cells, between $p[i]$ and $p[j]$.

	1	2	3
1	**Town**	**Address**	**Owner**
2	Deerfield	130 Central St.	Joe M.
3		10 Whately Ave.	Mary L.
4		TOTAL	2
5	Amherst	10 North Pleasant St.	Bob O.
6		55 Westman Lane	Claudia S.
7		2 Rectangle Ct.	Alyssa B.
8		TOTAL	3
9	Hadley	5 Rocky Rd.	Greg S.
10		34 Godell Rd.	Omar L.
11		TOTAL	2

	1	2	3
1	Deerfield	130 Central St.	Joe M.
2	Deerfield	10 Whately Ave.	Mary L.
3	Amherst	10 North Pleasant St.	Bob O.
4	Amherst	55 Westman Lane	Claudia S.
5	Amherst	2 Rectangle Ct.	Alyssa B.
6	Hadley	5 Rocky Rd.	Greg S.
7	Hadley	34 Godell Rd.	Omar L.

Figure 9: (a) An input spreadsheet. (b) The desired relational table. Note that missing fields have been inserted and headers and summary rows have been removed.

```
^[0-9]+$                 ^[a-z]+$
^[A-Z]+$                 ^[A-Z][a-z]+$
^[0-9]+ [a-zA-Z. ]+$     ^[a-zA-Z. ]+$
```

Figure 10: The example uses the above set of regular expressions.

{ `Node(1,^[A-Z][a-z]+$,`$\bot$`)` }
{ `Node(2,^[0-9] [a-zA-Z. ]+$,`$\bot$`)` }
{ `Node(3,^[a-zA-Z. ]+$,`$\bot$`)` }

Figure 11: Candidate cell constraints after calling Learn$\mathcal{N}$.

```
Edge(1,2,Vert(0),Horiz(1)),Select(All,All))
Edge(1,2,Vert(0),Horiz(*)),Select(All,x))
Edge(1,2,Vert(*),Horiz(1)),Select(y,All))
Edge(1,2,Vert(*),Horiz(*)),Select(y,x))
...
```

Figure 12: Candidate `Edge` constraints from the running example. Only the constraints for attribute pair $(1, 2)$ are shown, $x \in \{$`NearX`, `FarX`, `All`$\}$, and $y \in \{$`NearY`, `FarY`, `All`$\}$ for space reasons.

```
Node(1,^[A-Z][a-z]+$,⊥)
Node(2,^[0-9] [a-zA-Z. ]+$,⊥)
Node(3,^[a-zA-Z. ]+$,⊥)
Edge(1,2,Vert(0),Horiz(1)),Select(All,All))
Edge(2,3,Vert(0),Horiz(1)),Select(All,All))
```

Figure 13: The program synthesized given a single positive example.

Deerfield	130 Central St.	Joe M.
Amherst	10 North Pleasant St.	Bob O.
Hadley	5 Rocky Rd.	Greg S.

Figure 14: Program output after running the synthesis algorithm with a single positive example.

Given the set of deltas derived from all the positive examples for i and j, the set falls into one of more of the following classes. We use delta class information when calling `Edge` constructors to generate spatial constraints.

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 constant-length quantity values for `Vert()` or `Horiz()` constructors.
2. Delta distances vary, and the set of deltas is a bijective relation from $p[i]$ to $p[j]$. This class produces Kleene ("*") quantities for `Vert()` and `Horiz()` constructors and "match-single" select type tags (`NearX`, `FarX`; `NearY`, `FarY`).
3. Delta distances vary, and the set of deltas is not a bijective relation from $p[i]$ to $p[j]$. Specifically, one or more distinct $p[i]$ maps to two or more $p[j]$. This class produces Kleene ("*") quantities for `Vert()` and `Horiz()` constructor and "match-all" select type tag (`All`).

We also use delta class information when pruning for efficiency reasons (see 4.2.4, "Pruning"). The set of possible spatial constraints inferred for ID pair $(1, 2)$ for our single positive example is shown in Fig. 12.

4.2.3 Step 3: Find a satisfying set of constraints

Next FlashRelate searches for a set of constraints that satisfy its negative examples (Fig. 8a). As shown in Fig. 8b, there are five essential steps in this recursive procedure:

1. Exclude (`Node`, `Edge`) constraint pairs that would introduce a loop into the program graph given the current set of chosen constraints (line 7).
2. Call Rank$\mathcal{P}$ to rank constraint pairs (line 8). We discuss Rank$\mathcal{P}$ in §4.2.4 .
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 attributes 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 here some additional terms used in Fig. 8. Lines 6-16 represent the algorithm's implementation of nondeterministic choice. Each iteration of the `while` loop represents a choice point. $C_{\mathcal{E}}$ denotes the a set of chosen edges at a particular choice point. $C_{\mathcal{P}}$ denotes the set of chosen constraint pairs (α, β) at the same choice point where α is

a `Node(`$j, r,$`⊥)` constraint and β is an `Edge(`j, k, v, h, γ`)` constraint. $pairs$ 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 Rank$\mathcal{P}$ determines which pair of constraints the search ultimately chooses. We discuss these implementation choices in §4.2.4 .

In our example, suppose the synthesizer chooses the program shown in Fig.13. Since, in our example, the user has provided no negative examples, the chosen program trivially satisfies the criteria on line 2 in Fig. 8. Were this not the case, the search would backtrack and consider a different constraint pair.

The output of this program is shown in Fig. 14. While all of the extracted tuples are correct, the output is missing rows with omitted towns in the original spreadsheet (Fig. 9a). The user's next step is to add another positive example (e.g., the 2nd tuple in Fig. 9b) and call the synthesis algorithm again (Fig. 8a). This procedure continues until the user is satisfied with the output of the algorithm. We omit these additional user-interactions for brevity.

4.2.4 Implementation Details

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

1. Pruning of the search space, for efficient search.
2. Ranking schemes for synthesizer search choices.
3. Choice of data structures.

Pruning Constant-length spatial constraints can be ruled out when the geometric form has been shown to vary. In our example in Fig. 9, all `Edge` constraints containing `Vert(0)` for $(1, 2)$ can be ruled out because the cell in the second column in the second tuple ("10 Whately Ave.") is not a constant-length distance vertically from the first cell ("Deerfield"). In the first tuple, the vertical distance is zero; in the second the distance is 1.

Match-single spatial constraints can also be ruled out in some cases. In the same example mentioned above, it is not possible for the `Edge` constraint for $(1, 2)$ to contain a `Select()` with the tags `NearY` or `FarY`. This is because it is possible to match two different cells ("130 Central St." and "10 Whately Ave") for attribute 2 from the starting cell for attribute 1 ("Deerfield").

Ranking FLASHRELATE often has a number of alternative constraints to choose. The balance of specific vs general constraints is important because it 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. However, 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, resulting in a slow search. General constraints may require a user to provide more negative examples.

We tend to favor more specific programs 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 design choice also allows for faster search. We rank constraint pairs by the following schemes, in this order:

1. **R1** Constraints that exclude large numbers of negative examples are favored over constraints that exclude few. This scheme ensures that the search favors constraints that exclude negative examples.
2. **R2** 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. **R3** Straighter programs are favored over ones with more bends. This simplifies programs.
4. **R4** We consider the cells above and to the left of positive examples to be implicit negative examples.

Data Structures The effect of a constraint on a spreadsheet is a set of cells. We represent node matches as a bit vector (using .NET's `BigInteger` struct). Likewise, we 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 `Node` 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 $O(\#\,\text{bits set})$ using Wegner's bit-counting algorithm [27]. Bit-counting is frequently used in our ranking schemes to count the number cells matching a particular criteria.

We are able to evaluate partial programs efficiently during search using a dynamic programming scheme by caching the effect of each constraint on the input spreadsheet. 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.

5. Evaluation

In this section, we evaluate the design of FLARE and FLASHRELATE on a variety of real-world spreadsheets. The purpose of this evaluation is to answer the following questions:

1. It is 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 execution time and number of examples required by the synthesizer?

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[1]. The first set of benchmarks were borrowed from prior work [16] 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 FLARE. To round out our evaluation with more difficult tasks, we assembled a second set of benchmarks by searching the EUSES spreadsheet corpus [11] 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 also supplemented this second set with synthetic benchmarks known to present challenges to our synthesizer. As a measure of the complexity of the extractoin task for each benchmark, we note the number of variable-length `Edge` constraints that appear in our ground truth programs ("k*" in 16).

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 tuple from all of multi-dimensional data patterns we observed. We conclude that, independent of our ability to synthesize a FLARE program, FLARE is an effective tool for expressing data extraction tasks against semi-structured data projected into a two-dimensional grid.

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 §4 .

[1] All benchmarks will be made available online.

The following method is intended to simulate a user interacting with 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 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 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 that typical users longer wait times intolerable.

In our experiments, we consider 5 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 § 4. Each configuration examines the effect of adding a ranking scheme to the FLASHRELATE algorithm. Experiment configurations are: 1) **all rankings**, 2) **R1, R2, and R3**, 3) **R1 and R2**, 4) **R1**, and 5) **no ranking**. For each ranking scheme, **R#** refers to the procedure described in §4.2.4 (Ranking).

5.2 Results

Fig. 15a 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 60 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).

Fig. 15b 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.

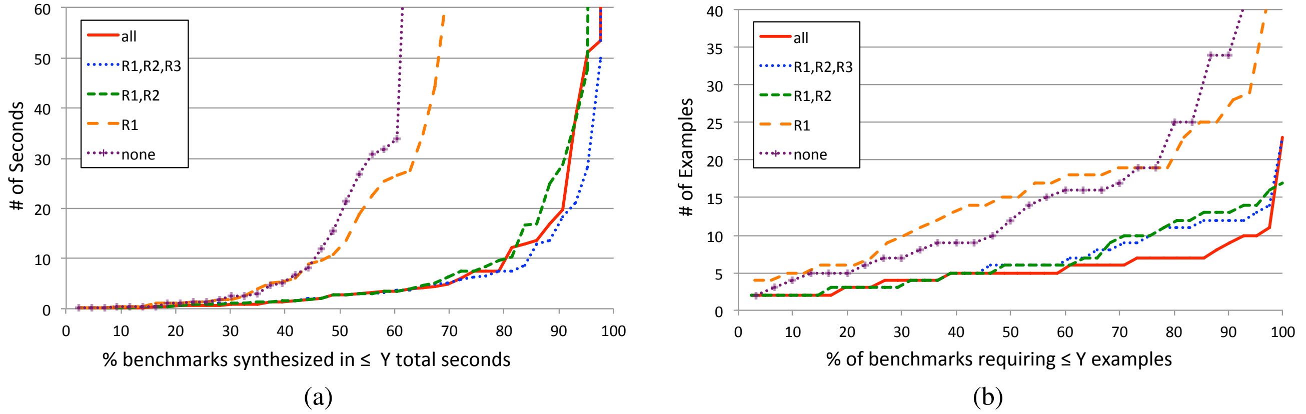


Figure 15: (a) *Total* benchmark synthesis times with ranking and without ranking schemes. Lower numbers are better. For example, **none** (no heuristics) succeeded on roughly 60% of the benchmarks before the timeout whereas **all** succeeded on all but one. (b) Number of examples required to successfully synthesize a program with and without ranking schemes. Lower numbers are better. Generally, **all** required fewer examples than the other heuristics.

	pre (sec)	worst (sec)	pos + neg	# k*	# col	# rec	# cells		pre (sec)	worst (sec)	pos + neg	# k*	# col	# rec	# cells
hg_ex2	0.1	5.7	3+2	0	9	14	270	_h8d62ck	0.2	2.7	5+5	2	8	62	864
hg_ex3	0.0	0.2	2+2	0	2	15	168	02rise	0.0	5.0	4+1	0	12	6	175
hg_ex4	0.0	0.2	2+1	1	2	6	9	03-1-report	0.1	0.4	6+1	4	4	177	702
hg_ex5	0.0	0.2	2+1	0	2	92	128	03PFMJOUR	0.1	2.0	3+2	1	7	10	288
hg_ex6	0.0	0.3	5+2	2	3	9	68	1213	0.1	1.0	3+2	4	4	116	350
hg_ex7	0.0	19.9	3+1	0	17	3	171	1214	1.7	0.9	5+2	3	3	62	162
hg_ex8	0.0	1.1	5+4	3	5	250	240	2003FinalPop	0.5	0.2	3+2	1	2	191	1113
hg_ex9	0.0	0.1	1+1	0	1	8	8	3_4	0.2	1.3	3+4	1	5	68	522
hg_ex10	0.0	0.2	2+1	1	2	20	30	3q2000	0.9	19.3	4+1	0	19	5	3780
hg_ex11	0.0	0.3	3+3	2	3	19	42	3yrsegment	0.1	1.7	3+1	2	5	55	750
hg_ex12	0.1	1.9	3+1	0	9	3	252	4q03fax	0.2	0.9	7+3	2	3	186	1394
hg_ex13	0.0	0.5	4+2	4	4	5	63	act3_23	0.0	0.7	3+4	3	3	24	175
hg_ex17	0.0	0.4	1+1	0	4	16	160	act4_023	0.1	0.6	4+3	2	4	222	312
hg_ex18	0.0	2.0	3+2	0	6	3	28	Appen4-5	TO	TO	NA	3	4	99	1020
hg_ex26	0.0	0.4	1+1	0	4	4	16	ascap_2000	0.3	0.8	5+1	0	6	168	252
hg_ex29	0.0	4.0	1+1	0	9	3	60	ascap_2001	0.1	0.8	5+1	2	6	39	1037
hg_ex35	0.6	6.4	4+1	0	14	36	2786	e-learning	0.2	2.2	8+15	4	6	127	522
hg_ex36	0.0	0.8	1+1	0	6	3	36	Favorite	0.0	0.3	3+3	2	3	36	68
hg_ex37	0.0	2.0	1+1	0	10	5	55	historical	0.3	1.3	10+1	3	4	251	1911
hg_ex39	0.1	0.2	2+2	0	2	49	146	Staircase	0.0	0.3	3+2	2	3	3	9
hg_ex40	0.0	8.9	1+1	0	12	1	12	Table_201	0.2	4.9	7+1	4	9	45	540
hg_ex41	0.1	3.2	7+1	2	10	171	840								

Figure 16: Benchmarks from [16] are shown on the left while benchmarks from [11] 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. We abbreviate file names for compactness.

Table 16 summarizes the time and number of examples required by the algorithm with all rankings enabled as well as benchmark complexity.

We find that FLASHRELATE is quite 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), which we consider to be an acceptable tradeoff, generally our ranking schemes reduce both the number of examples required and the time the user has to wait.

5.3 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# as a Microsoft Excel 2010 VSTO.NET plugin for Windows 8.

5.4 Future Work

Users sometimes use other encodings, like color or typeface formatting. While FLARE does not currently support constraints based on these attributes, they are a straightforward extension to our work. Combining FLASHRELATE with other program synthesis techniques would allow end users to handle ad-hoc in-cell encodings [15].

6. Conclusion

The flexibility of spreadsheets allows users to combine data definitions and data views, providing expressiveness at the expense of utility. The ad-hoc structure of such spreadsheets makes it challenging to leverage existing tools (for data analysis, etc.) that require data to be present in a standard relational format. We present FLARE, the first language to allow users to express relational extraction queries against semi-structured, two-dimensional spreadsheet data. FLARE is much like a two-dimensional AWK. We also present FLASHRELATE, an algorithm for the automatic synthesis of FLARE programs from user-provided positive and negative examples. Using EUSES and other spreadsheet benchmarks, we show that we can often synthesize programs that users want, making FLARE accessible to non-experts.

A video demonstration of FLASHRELATE is available at: `http://tinyurl.com/oz72quh`

References

[1] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *VL/HCC*, pages 165–172, 2004.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[3] M. Anselmo, D. Giammarresi, and M. Madonia. Regular expressions for two-dimensional languages over one-letter alphabet. In *Developments in Language Theory*, Lecture Notes in Computer Science. 2005.

[4] M. J. Cafarella, A. Halevy, and J. Madhavan. Structured data on the web. *CACM*, 54(2):72–79, 2011.

[5] Z. Chen and M. Cafarella. Automatic web spreadsheet dat extraction. In *SSW'13*, 2013.

[6] Z. Chen, M. Cafarella, J. Chen, D. Prevo, and J. Zhuang. Senbazuru: a prototype spreadsheet database management system. *PVLDB*, 6(12):1202–1205, 2013.

[7] J. Cunha, J. Saraiva, and J. Visser. From spreadsheets to relational databases and back. In *PEPM 2009*, pages 179–188. ACM, 2009.

[8] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, 2010.

[9] E. Ferrara, P. De Meo, G. Fiumara, and R. Baumgartner. Web data extraction, applications and techniques: a survey. *arXiv preprint arXiv:1207.0246*, 2012.

[10] K. Fisher and D. Walker. The PADS project: an overview. In *ICDT*, 2011.

[11] M. I. Fisher and G. Rothermel. The EUSES Spreadsheet Corpus: A shared resource for supporting experimentation with spreadsheet dependability mechanisms. In *1st WEUSE*, pages 47–51, 2005.

[12] D. Giammarresi, F. Venezia, and A. Restivo. Two-dimensional languages, 1997.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[14] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 2012.

[15] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *CACM*, Aug 2012.

[16] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[17] J. Kari and C. Moore. Rectangles and squares recognized by two-dimensional automata. *Theory Is Forever*, 2004.

[18] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[19] E. Lu, R. Bodik, and B. Hartmann. Quicksilver: Automatic Synthesis of Relational Queries. Technical Report UCB/EECS-2013-68, UC-Berkeley, May 2013. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-68.html`.

[20] O. Matz. Regular expressions and context-free grammars for picture languages. In *STACS*, 1997.

[21] E. Oro and M. Ruffolo. Sila: a spatial instance learning approach for deep webpages. In *CIKM 2011*, pages 2329–2332. ACM, 2011.

[22] E. Oro, M. Ruffolo, and S. Staab. Sxpath: extending xpath towards spatial querying on web documents. *PVLDB*, 4(2): 129–140, 2010.

[23] M. Pradella, A. Cherubini, and S. C. Reghizzi. A unifying approach to picture grammars. *Information and Computation*, 209(9), 2011.

[24] T. Register. Microsoft feeds excel to supercomputer, Nov. 2009. URL `http://www.theregister.co.uk/2009/11/18/sc09_microsoft_excel_hpc/`.

[25] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012.

[26] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, 2009.

[27] P. Wegner. A technique for counting ones in a binary computer. *Commun. ACM*, 3(5):322–, May 1960. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/367236.367286`.