Foofah: Transforming Data By Example

Zhongjun Jin Michael R. Anderson Michael Cafarella H. V. Jagadish
University of Michigan, Ann Arbor
{markjin,mrander,michjc,jag}@umich.edu

ABSTRACT

Data transformation is a critical first step in modern data analysis: before any analysis can be done, data from a variety of sources must be wrangled into a uniform format that is amenable to the intended analysis and analytical software package. This data transformation task is tedious, time-consuming, and often requires programming skills beyond the expertise of data analysts. In this paper, we develop a technique to synthesize data transformation programs by example, reducing this burden by allowing the analyst to describe the transformation with a small input-output example pair, without being concerned with the transformation steps required to get there. We implemented our technique in a system, FOOFAH, that efficiently searches the space of possible data transformation operations to generate a program that will perform the desired transformation. We experimentally show that data transformation programs can be created quickly with FOOFAH for a wide variety of cases, with 60% less user effort than the well-known WRANGLER system.

Keywords

Data Transformation; Program Synthesis; Programming By Example; A* algorithm; Heuristic

1. INTRODUCTION

The many fields that depend on data for decision making have at least one thing in common: raw data is often in a non-relational or poorly structured form, possibly with extraneous information, and cannot be directly used by a downstream information system, like a database or visualization system. Figure 1 from [16] is a good example of such raw data. In modern data analytics, data transformation (or data wrangling) is usually a crucial first step that reorganizes raw data into a more desirable format that can be easily consumed by other systems. Figure 2 showcases a relational form obtained by transforming Figure 1.

Traditionally, domain experts handwrite task specific scripts to transform unstructured data—a task that is often labor-

Bureau of I.A.	
Regional Director	Numbers
Niles C.	Tel: (800)645-8397
	Fax: (907)586-7252
Jean H.	Tel: (918)781-4600
	Fax: (918)781-4604
Frank K.	Tel: (615)564-6500
	Fax: (615)564-6701

...

Figure 1: A spreadsheet of business contact information

	Tel	Fax
Niles C.	(800)645-8397	(907)586-7252
Jean H.	(918)781-4600	(918)781-4604
Frank K.	(615)564-6500	(615)564-6701

...

Figure 2: A relational form of Figure 1

intensive and tedious. The requirement for programming hamstrings data users that are capable analysts but have limited coding skills. Even worse, these scripts are tailored to particular data sources and cannot adapt when new sources are acquired. People normally spend more time preparing data than analyzing it; up to 80% of a data scientist's time can be spent on transforming data into a usable state [28].

Recent research into automated and assisted data transformation systems have tried to reduce the need of a programming background for users, with some success [19, 22, 41]. These tools help users generate reusable data transformation programs, but they still require users to know which data transformation operations are needed and in what order they should be applied. Current tools still require some level of imperative programming, placing a significant burden on data users. Take WRANGLER [22], for example, where a user must select the correct operators and parameters to complete a data transformation task. This is often challenging if the user has no experience in data transformation or programming.

In general, existing data transformation tools are difficult to use due to two usability issues:

- *High Skill*: Users must be familiar with the often complicated transformation operations and then decide which operations to use and in what order.
- *High Effort*: The amount of user effort increases as the data transformation program gets lengthy.

To resolve the above usability issues, we envision a data transformation program synthesizer that can be successfully used by people without a programming background and that requires minimal user effort. Unlike WRANGLER, which asks

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD'17, May 14-19, 2017, Chicago, IL, USA
© 2017 ACM. ISBN 978-1-4503-4197-4/17/05...$15.00
DOI: http://dx.doi.org/10.1145/3035918.3064034

the user for procedural hints, this system should allow the user to specify a desired transformation simply by providing an input-output example: the user only needs to know how to *describe the transformed data*, as opposed to knowing any particular transformation operation that must be performed.

Our Approach — In this paper, we solve the data transformation program synthesis problem using a Programming By Example (PBE) approach. Our proposed technique aims to help an unsophisticated user easily generate a quality data transformation program using purely input-output examples. The synthesized program is designed to be easy-to-understand (it is a straight-line program comprised of simple primitives), so an unsophisticated user can understand the semantics of the program and validate it. Because it is often infeasible to examine and approve a very large transformed dataset synthesizing a readable transformation program is preferred over performing an opaque transformation.

We model program synthesis as a search problem in a state space graph and use a heuristic search approach based on the classic A* algorithm to synthesize the program. A major challenge in applying A* to program synthesis is to create a heuristic function estimating the cost of any proposed partial solution. Unlike robotic path planning, where a metric like Euclidean distance naturally serves as a good heuristic function, there is no straightforward heuristic for data transformation. In this work, we define an effective A* heuristic for data transformation, as well as lossless pruning rules that significantly reduce the size of the search space. We have implemented our methods in a prototype data transformation program synthesizer called FOOFAH.

Organization — After motivating our problem with an example in Section 2 and formally defining the problem in Section 3, we discuss the following contributions:

- We present a PBE data transformation program synthesis technique backed by an efficient heuristic-search-based algorithm inspired by the A* algorithm. It has a novel, operator-independent heuristic, Table Edit Distance Batch, along with pruning rules designed specifically for data transformation (Section 4).
- We prototype our method in a system, FOOFAH, and evaluate it with a comprehensive set of benchmark test scenarios that show it is both effective and efficient in synthesizing data transformation programs. We also present a user study that shows FOOFAH requires about 60% less user effort than WRANGLER(Section 5).

We explore Related Work in Section 6 and finish with a discussion of future work in Section 7

2. MOTIVATING EXAMPLE

Data transformation can be a tedious task involving the application of complex operations that may be difficult for a naïve user to understand, as illustrated by the following simple but realistic scenario:

Example 1. *Bob wants to load a spreadsheet of business contact information (Figure 1) into a database system. Unfortunately, the raw data cannot be loaded in its original format, so Bob hopes to transform it into a relational format (Figure 2). Manually transforming the data record-by-record would be tedious and error-prone, so he uses the interactive data cleaning tool* WRANGLER *[22].*

Niles C.	Tel	(800)645-8397
	Fax	(907)586-7252
Jean H.	Tel	(918)781-4600
	Fax	(918)781-4604
Frank K.	Tel	(615)564-6500
	Fax	(615)564-6701

Figure 3: Intermediate table state

	Tel	Fax
Niles C.	(800)645-8397	
		(615)564-6701
Jean H.	(918)781-4600	
Frank K.	(615)564-6500	

Figure 4: Perform Unfold before Fill

Bob first removes the rows of irrelevant data (rows 1 and 2) and empty rows (rows 5, 8, and more). He then splits the cells containing phone numbers on ":", extracting the phone numbers into a new column. Now that almost all the cells from the desired table exist in the intermediate table (Figure 3), Bob intends to perform a cross-tabulation operation that tabulates phone numbers of each category against the human names. He looks through WRANGLER*'s provided operations and finally decides that* Unfold *should be used. But* Unfold *does not transform the intermediate table correctly, since there are missing values in the column of names, resulting in "null" being the unique identifier for all rows without a human name (Figure 4). Bob backtracks and performs a* Fill *operation to fill in the empty cells with the appropriate names before finally performing the* Unfold *operation. The final data transformation program is shown in Figure 5.*

The usability issues described in Section 1 have occurred in this example. Lines 1–3 in Figure 5 are lengthy and repetitive (*High Effort*). Lines 5–6 require a good understanding of the Unfold operation, causing difficulty for the naïve user (*High Skill*). Note that Deletes in Lines 1–2 are different from the Delete in Line 3 in that the latter could apply to the entire file. Non-savvy users may find such conditional usage of Delete difficult to discover, further illustrating the *High Skill* issue.

Consider another scenario where the same task becomes much easier for Bob, our data analyst:

Example 2. *Bob decides to use an alternative data transformation system,* FOOFAH*. To use* FOOFAH*, Bob simply needs to choose a small sample of the raw data (Figure 1) and describe what this sample should be after being transformed (Figure 2).* FOOFAH *automatically infers the data transformation program in Figure 6 (which is semantically the same as Figure 5, and even more succinct). Bob takes this inferred program and executes it on the entire raw dataset and finds that raw data are transformed exactly as desired.*

The motivating example above gives an idea of the real-world data transformation tasks our proposed technique is designed to address. In general, we aim to transform a poorly-structured grid of values (e.g., a spreadsheet table) to a relational table with coherent rows and columns. Such a transformation can be a combination of the following chores:

1. changing the structure of the table
2. removing unnecessary data fields
3. filling in missing values
4. extracting values from cells
5. creating new cell values out of several cell values

```
1 Delete row 1
2 Delete row 2
3 Delete rows where column 2 is null
4 Split column 2 on ':'
5 Fill split with values from above
6 Unfold column 2 on column 3
```

Figure 5: Program created with WRANGLER

```
1 t = split(t, 1, ':')
2 t = delete(t, 2)
3 t = fill(t, 0)
4 t = unfold(t, 1)
```

Figure 6: Program synthesized with FOOFAH

We assume that the input data should be transformed without any extra semantic information, so, for example, transforming "NY" to "New York" is not possible (previous projects [1,9,37] have addressed such semantic transformations). Transformations should not add new information that is not in the input table, such as adding a column header. We provide another example use case in Appendix B.

3. PROBLEM DEFINITION

To help the user synthesize a correct data transformation program, we take a Programming By Example (PBE) approach: the user provides an input-output example pair, and the system generates a program satisfying the example pair and hopefully can correctly transform the full dataset $\mathcal{R}$.

3.1 Problem Definition

With all notations summarized in Table 1, we define this problem formally:

Problem *Given a user's set of input-output examples* $\mathcal{E} = (e_i, e_o)$*, where* e_i *is drawn from raw dataset* $\mathcal{R}$ *and* e_o *is the desired transformed form of* e_i*, synthesize a data transformation program* $\mathcal{P}$*, parameterized with a library of data transformation operators, that will transform* e_i *to* e_o*.*

Like previous work in data transformation [17, 22], we assume the raw data $\mathcal{R}$ is a grid of values. $\mathcal{R}$ might not be relational but must have some regular structure (and thus may have been programmatically generated). Further, $\mathcal{R}$ may contain schematic information (e.g., column or row headers) as table values, and even some extraneous information (e.g., "Bureau of I.A." in Figure 1).

Once the raw data and the desired transformation meet the above criteria, the user must choose the input sample and specify the corresponding output example. More issues with creating quality input-output examples will be discussed in detail in Section 4.5.

3.2 Data Transformation Programs

Transforming tabular data into a relational table usually require two types of transformations: *syntactic transformations* and *layout transformations* [13]. Syntactic transformations reformat cell contents (e.g., split a cell of "mm/dd/yyyy" into three cells containing month, day, year). Layout transformations do not modify cell contents, but instead change how the cells are arranged in the table (e.g., relocating cells containing month information to be column headers).

We find that the data transformation operators shown in Table 2 (defined in Potter's Wheel project [33, 34] and used by state-of-art data transformation tool WRANGLER [22]) are

Notation	Description
$\mathcal{P} = \{p_1, \ldots, p_n\}$	Data transformation program
$p_i = (op_i, par_1, \ldots)$	Transformation operation with operator op_i and parameters par_1, par_2, etc.
$\mathcal{R}$	Raw dataset to be transformed
$e_i \in \mathcal{R}$	Example input sampled from $\mathcal{R}$ by user
$e_o = \mathcal{P}(e_i)$	Example output provided by user, transformed from e_i
$\mathcal{E} = (e_i, e_o)$	Input-output example table pair, provided as input to the system by user

Table 1: Frequently used notation

Operator	Description
Drop	Deletes a column in the table
Move	Relocates a column from one position to another in the table
Copy	Duplicates a column and append the copied column to the end of the table
Merge	Concatenates two columns and append the merged column to the end of the table
Split	Separates a column into two or more halves at the occurrences of the delimiter
Fold	Collapses all columns after a specific column into one column in the output table
Unfold	"Unflatten" tables and move information from data values to column names
Fill	Fill empty cells with the value from above
Divide	Divide is used to divide one column into two columns based on some predicate
Delete	Delete rows or columns that match a given predicate
Extract	Extract first match of a given regular expression each cell of a designated column
Transpose	Transpose the rows and columns of the table
Wrap (added)	Concatenate multiple rows conditionally

Table 2: Data transformation operators used by FOOFAH

expressive enough to describe these two types of transformations. We use these operations in FOOFAH: operators like Split and Merge are syntactic transformations and operators like Fold, and Unfold are layout transformations. To illustrate the type of operations in our library, consider Split. When applying Split parameterized by ':' to the data in Figure 7, we get Figure 8 as the output. Detailed definitions for each operator are shown in Appendix A.

Our proposed technique is not limited to supporting Potter's Wheel operations; users are able to add new operators as needed to improve the expressiveness of the program synthesis system. We assume that new operators will match our system's focus on syntactic and layout transformations (as described in Section 2); if an operator attempts a semantic transformation, our system may not correctly synthesize programs that use it. As we describe below, the synthesized programs do not contain loops, so novel operators must be useful outside a loop's body.

We have tuned the system to work especially effectively when operators make "conventional" transformations that apply to an entire row or column at a time. If operators were to do otherwise — such as an operator for "Removing the cell values at odd numbered rows in a certain column", or for "Splitting the cell values on Space in cells whose values start with 'Math"' — the system will run more slowly. Experimental results in Section 5.5 show evidence that adding operators can enhance the expressiveness of our synthesis technique without hurting efficiency.

Numbers
Tel:(800)645-8397
Fax:(907)586-7252

Figure 7: Pre-Split data

Numbers	
Tel	(800)645-8397
Fax	(907)586-7252

Figure 8: After Split on ':'

Program Structure — All data transformation operators we use take in a whole table and output a new table. A reasonable model for most data transformation tasks is to sequentially transform the original input into a state closer to the output example until we finally reach that goal state. This linear process of data transformation results in a loop-free or straight-line program, a simple control structure successfully applied in many previous data transformation projects [17, 22, 23, 42]. We use the operators mentioned above as base components. Synthesizing loops is usually unnecessary in our case because the operators in our operator library are defined to potentially apply to multiple values. Nevertheless, the loop-free program structure could restrict us from synthesizing programs that require an undetermined number of iterations of a data transformation operation, or could lead to verbose programs with "unrolled loops." For example, if the user wants to "Drop column 1 to column $\lfloor k/2 \rfloor$ where k is the number of columns in the table" our system will be unable to synthesize a loop-based implementation and instead will simply repeat Drop many times.

Motivated by the above considerations, we formally define the data transformation to be synthesized as follows:

Definition 3.1 (Data transformation program $\mathcal{P}$)**.** *$\mathcal{P}$ is a loop-free series of operations $(p_1, p_2, ..., p_k)$ such that: **1**. Each operation $p_i = (op_i, par_1, \dots) : t_{in} \rightarrow t_{out}$. p_i includes operator op_i with corresponding parameter(s) and transforms an input data table t_{in} to an output data table t_{out}. **2**. The output of operation p_i is the input of p_{i+1}.*

4. PROGRAM SYNTHESIS

We formulate data transformation program synthesis as a search problem. Other program synthesis approaches are not efficient enough given the huge search space in our problem setting (Section 4.1). We thus propose an efficient heuristic search method, inspired by the classic A* algorithm. In Section 4.2, we introduce a straw man heuristic and then present our novel operator-independent heuristic, *Table Edit Distance Batch* (TED Batch), based on a a novel metric, *Table Edit Distance* (TED), which measures the dissimilarity between tables. In addition, we propose a set of pruning rules for data transformation problems to boost search speed (Section 4.3). We compare the time complexity of our technique with other previous projects (Section 4.4). Finally, we discuss issues about creating examples and validation (Section 4.5).

4.1 Program Synthesis Techniques

In Section 3.2, we described the structure of our desired data transformation program to be component-based and loop-free. Gulwani et al. proposed a *constraint-based* program synthesis technique to synthesize loop-free bit-manipulation programs [15, 21] using logic solvers, like the SMT solver. However, the constraint-based technique is impractical for our interactive PBE system because the number of constraints dramatically increases as the size of data increases, scaling the problem beyond the capabilities of modern logic solvers.

Other methods for synthesizing component programs include *sketching* and *version space algebra*. Solar-Lezama's work with sketching [40] attempts to formulate certain types of program automatically through clever formulation of SAT solving methods. This approach focuses on programs that are "difficult and important" for humans to write by hand, such for thread locking or decoding compressed data streams, so it is acceptable for the solver to run for long periods. In contrast, our aims to improve productivity on tasks that are "easy but boring" for humans. To preserve interactivity for the user, our system must find a solution quickly.

Version space algebra requires a complete search space of programs between two states, which make it more suitable for a Programming By Demonstration problem where the user explicitly provides intermediate states and the search space between these states is small [27] or for PBE problems that can be easily divided into independent sub-problems [12]. In our problem, the search space of the synthesized programs is exponential, and thus version space algebra is not practical.

Search-based techniques are another common approach used by previous program synthesis projects [25, 30, 32, 35]. For our problem, we formulate program synthesis as a search problem in a *state space graph* defined as follows:

Definition 4.1 (Program synthesis as a search problem)**.** *Given input-output examples $\mathcal{E} = (e_i, e_o)$, we construct a state space graph $G(V, A)$ where arcs A represent candidate data transformation operations, vertices V represent intermediate states of the data as transformed by the operation on previously traversed arcs, e_i is the initial state v_0, and e_o is the goal state v_n. Synthesizing a data transformation program is finding a path that is a sequence of operations leading from v_0 to v_n in G.*

Graph Construction — To build a state space graph G, we first expand the graph from v_0 by adding out-going edges corresponding to data transformation operators (e.g., Drop, Fold) with all possible parameterizations (parameters and their domains for each operator are defined both in [34] and Appendix A). The resulting intermediate tables become the vertices in G. Since the domain for all parameters of our operator set is restricted, the number of arcs is still tractable. More importantly, in practice, the pruning rules introduced in Section 4.3 trim away many obviously incorrect operations and states, making the actual number of arcs added for each state reasonably small (e.g., the initial state e_i in Figure 10 has 15 child states, after 161 are pruned).

If no child of v_0 happens to be the goal state v_n, we recursively expand the most promising child state (evaluated using the method introduced in Section 4.2) until we finally reach v_n. When the search terminates, the path from v_0 to v_n is the sequence of operations that comprise the synthesized data transformation program.

4.2 Search-based Program Synthesis

Due to our formulation of the synthesis problem, the search space is exponential in the number of the operations in the program. Searching for a program in a space of this size can be difficult. *Brute-force* search quickly becomes intractable. As a PBE solution needs to be responsive to preserve interactivity, we are exposed to a challenging search problem with a tight time constraint. To solve the problem, we develop a heuristic search algorithm for synthesizing data transformation programs inspired by the classic A* algorithm [18]. With appropriate pruning rules and careful choice of exploration order, we are able to achieve good performance.

Operator	Description
Add	Add a cell to table
Delete	Remove a cell from table
Move	Move a cell from location (x_1, y_1) to (x_2, y_2)
Transform	Syntactically transform a cell into a new cell

Table 3: Table Edit Operators

A* is a common approach to address search problems and has been successfully applied in previous work on program synthesis [25, 35]. To find a path in the graph from the initial state to the goal state, the A* algorithm continually expands the state with the minimum cost $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach state n from the initial state and heuristic function $h(n)$ is the approximate cost of the cheapest path from state n to the goal state. The definition of cost depends on the performance measure of the search task. In robotic pathfinding, the cost is typically distance traveled. In our problem, we prefer shorter programs over longer ones, because we believe shorter programs will be easier to understand. For these programs, correctness and readability are far more important than the program's computational efficiency (our operators all have complexity linear in the size of the input file), so we do not search for computationally "cheap" programs. We define cost as follows:

Definition 4.2 (Data transformation cost)**.** *Given any two states (v_i, v_j) in graph G, cost is the minimum number of data transformation operations needed to transform v_i to v_j.*

Note that we treat all operators equally. Although, some operators like Fold might be conceptually more complex for users to understand, we observe that such operators rarely occur more than once in our benchmarks.

Additionally, an *admissible* heuristic helps synthesize a program with the minimum number of data transformation operations. This is ideal but not necessary. By relaxing the need for admissibility, we may accept a program that is slightly longer than the program with the minimal length.

Naïve Heuristic — Possibly the most straightforward heuristic is a rule-based one. The intuition is that we create some rules, based on our domain knowledge, to estimate whether a certain Potter's Wheel operator is needed given $\mathcal{E}$, and use the total count as the final heuristic score in the end. An example heuristic rule for the Split operator is "number of cells from $T_i[k]$ (i.e., the *row k* in T_i) with strings that do not appear fully in $T_o[k]$, but do have substrings that appear in $T_o[k]$." (This is a reasonable rule because the Split operator splits a cell value in the input table into two or more pieces in the output table, as in Figures 7 and 8.) The details about this naïve heuristic are presented in Appendix C.

Although this naïve heuristic might appear to be effective for our problem, it is weak for two reasons. First, the estimation is likely to be inaccurate when the best program entails layout transformations. Second, the heuristic is defined in terms of the existing operators and will not easily adapt to new operators in the future. We expect different operators to be helpful in different application scenarios and our framework is designed to be operator independent.

To overcome these shortcomings, we have designd a novel heuristic function explicitly for tabular data transformation.

4.2.1 Table Edit Distance

The purpose of the heuristic function in A* is guiding the search process towards a more promising direction. Inspired by a previous research [35], which used edit distance as the heuristic function, we define *Table Edit Distance* (TED), which measures the table dissimilarity:

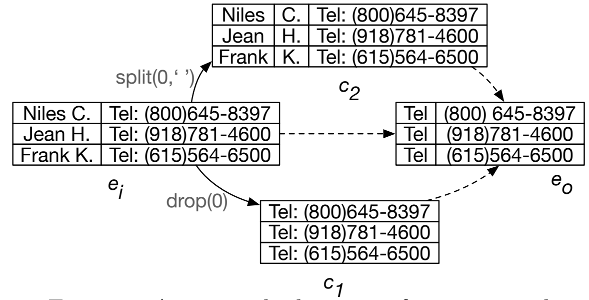


Figure 9: An example data transformation task

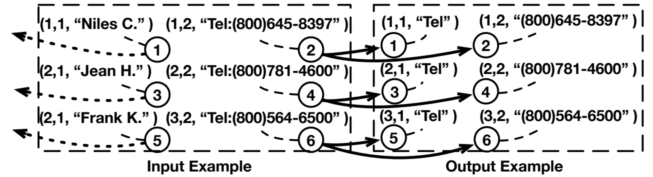


Figure 10: Cell-level edit operations composing the transformation from Table e_i to Table e_o in Figure 9 (circles are cells, bold arrows are Transforms, dashed arrows are Deletes)

$$TED(T_1, T_2) = \min_{(p_1,\ldots,p_k) \in P(T_1,T_2)} \sum_{i=i}^{k} cost(p_i) \qquad (1)$$

TED is the minimum total cost of *table edit operations* needed to transform T_1 to T_2, where $P(T_1, T_2)$ denotes the set of edit paths transforming T_1 to T_2 and $cost(p_i)$ is the cost of each table edit operation p_i. The table edit operations include Add, Delete, Move, Transform (see Table 3 for definition).

Inspired by the *graph edit distance* algorithm [31], we designed an algorithm to calculate the exact TED. Unfortunately, computing TED in real time is not practical: it is equivalent to computing graph edit distance, which is NP-complete [11]. (See Appendix D for this algorithm.)

We therefore designed an efficient greedy algorithm to approximate TED, shown in Algorithm 1. The idea behind Algorithm 1 is to greedily add the cheapest operations among the candidate operations to formulate each cell in the output table e_o, building up a sequence of edits until we obtain a *complete edit path*. The edit path formulates the entire output table. The final heuristic score is the total cost of this path.

Algorithm 1 consists of three core steps. We use Figure 10, which describes the edit path found to transform input table e_i to e_o in Figure 9, as an example to explain each step.

Step 1. *(lines 3–19) For each unprocessed cell in the output table (picked in row-major order), we choose the cheapest cell-specific operation sequence (a tie is broken by row-major order of the cell from the input table), from one of:*

1. *Transformation from an unprocessed cell in e_x into a cell in e_o. Transformation sequences for a pair of cells is generated by the function "AddCandTransform" and can include a Move operator (if the cell coordinates differ), a Transform operator (if the cell contents differ), or both operators (if both conditions apply).*
2. *Add a new cell to e_o.*

Algorithm 1: Approximate TED Algorithm

Data: Intermediate Table $e_x = \{u_1, u_2, \ldots, u_{|e_x|}\}$, where u_i represents a cell from e_x; Example Output Table $e_o = \{v_1, v_2, \ldots, v_{|e_o|}\}$, where v_i represents a cell from e_o

Result: cost, edit path

```
1  p_final ← ∅;
2  p_temp ← ∅;
3  for w in e_x do
4      add AddCandTransform(w, v_1) to p_temp;
5  add Add(v_1) to p_temp;
6  p_final ← argmin_{∀p∈p_temp} cost(p) ;
7  Let {u_1, ..., u_j} & {v_1, ..., v_k} be processed cells;
8  while j <| e_x | and k <| e_o | do
9      p_temp ← ∅;
10     for w ∈ {u_{j+1}, ..., u_{|e_x|}} do
11         add AddCandTransform(w, v_{k+1}) to p_temp;
12     add Add(v_{k+1}) to p_temp;
13     if cost(argmin_{∀p∈p_temp} cost(p)) ≥ ∞ then
14         p_temp ← ∅;
15         for w ∈ {u_1, ..., u_{|e_x|}} do
16             add AddCandTransform(w, v_{k+1}) to p_temp;
17         add Add (v_{k+1}) to p_temp;
18     p_final ← p_final ∪ argmin_{∀p∈p_temp} cost(p) ;
19     Let {u_1, ..., u_j} & {v_1, ..., v_k} be processed cells;
20 if j <| e_x | then
21     for w ∈ {u_{j+1}, ..., u_{|e_x|}} do
22         add Delete (w) to p_final
23 if k <| e_o | then
24     for q ∈ {v_{k+1}, ..., v_{|e_o|}} do
25         p_temp ← ∅;
26         for w ∈ {u_1, ..., u_{|e_x|}} do
27             add AddCandTransform(w, q) to p_temp;
28         add Add (q) to p_temp;
29         p_final ← p_final ∪ argmin_{∀p∈p_temp} cost(p) ;
30 Return cost(p_final), p_final
```

After picking an operation sequence, we hypothesize an edit path (p_{temp}) for each cell that consists of all edits made so far, plus the chosen operation. We measure the cost of each edit path using the cost function. By default, all table edit operations have equal cost; however, we assign a cost of infinity to: (1) Transform operations between cells with no string containment relationship and (2) Add operations for non-empty cells. (These fall into the category of tasks beyond the scope of our technique described in Section 2.)

For example, for $\mathcal{O}_1$ in Figure 10[1], Algorithm 1 finds that transforming from $\mathcal{I}_2$ to $\mathcal{O}_1$ to is the best, because the costs of transforming from $\mathcal{I}_1$, $\mathcal{I}_3$, and $\mathcal{I}_5$ are all infinite (no string containment), and although transforming from $\mathcal{I}_4$ or $\mathcal{I}_6$ costs the same as transforming from $\mathcal{I}_2$, $\mathcal{I}_2$ has a higher row-major order in the input table. For $\mathcal{O}_2$, we find that transforming from any unprocessed cell in the input example (i.e., $\mathcal{I}_{1,3,4,5,6}$) to $\mathcal{O}_2$ yields an infinite cost, so using only the unprocessed cells would not result in a reasonable edit path. We fix this problem by adding transformations from the processed cells in lines 13–18; this helps find the cheapest edit operation to formulate $\mathcal{O}_2$: transforming from $\mathcal{I}_2$ to $\mathcal{O}_2$.

[1] $\mathcal{O}_n$ means cell n from e_o. $\mathcal{I}_m$ means cell m from e_i.

Step 2. *(line 20–22) Delete all unprocessed cells from e_x.*

In our running example, after we have discovered edit operations for all cells in the output example, we find that cells 1, 3, and 5 from the input example remain unprocessed. We simply add Delete operations to remove them.

Step 3. *(line 23–29) When we have unprocessed cells in e_o, but no remaining unprocessed cells in e_x, our only options are to: (1) Transform from a processed cell in e_x, (we process every input cell at least one time before processing any cell for a second time) OR (2) Add a new cell.*

The edit path discovered in Figure 10 is as follows:[2]

$$P_0 = \left\{ \begin{array}{cc} \text{Transform((1,2),(1,1))}, & \text{Move((1,2),(1,1))} \\ \text{Transform((1,2),(1,2))}, & \text{Transform((2,2),(2,1))} \\ \text{Move((2,2),(2,1))}, & \text{Transform((2,2),(2,2))} \\ \text{Transform((3,2),(3,1))}, & \text{Move((3,2),(3,1))} \\ \text{Transform((3,2),(3,2))}, & \text{Delete((1,1))} \\ \text{Delete((2,1))}, & \text{Delete((3,1))} \end{array} \right\}$$

Figure 9 shows a data transformation task, where e_i is the input example and e_o is the output example. c_1 and c_2 are two child states of e_i representing outcomes of two possible candidate operations applied to e_i. Below we define P_1 and P_2, the edit paths discovered by Algorithm 1 for c_1 and c_2:

$$P_1 = \left\{ \begin{array}{cc} \text{Transform((1,1),(1,1))}, & \text{Transform((1,1),(1,2))} \\ \text{Move((1,1),(1,2))}, & \text{Transform((2,1),(2,1))} \\ \text{Transform((2,1),(2,2))}, & \text{Move((2,1),(2,2))} \\ \text{Transform((3,1),(3,1))}, & \text{Transform((3,1),(3,2))} \\ \text{Move((3,1),(3,2))} & \end{array} \right\}$$

$$P_2 = \left\{ \begin{array}{cc} \text{Transform((1,3),(1,1))}, & \text{Move((1,3),(1,1))} \\ \text{Transform((1,3),(1,2))}, & \text{Move((1,3),(1,2))} \\ \text{Transform((2,3),(2,1))}, & \text{Move((2,3),(2,1))} \\ \text{Transform((2,3),(2,2))}, & \text{Move((2,3),(2,2))} \\ \text{Transform((3,3),(3,1))}, & \text{Move((3,3),(3,1))} \\ \text{Transform((3,3),(3,2))}, & \text{Move((3,3),(3,2))} \\ \text{Delete((1,1))}, & \text{Delete((2,1))} \\ \text{Delete((3,1))}, & \text{Delete((1,2))}, \\ \text{Delete((2,2))}, & \text{Delete((3,2))} \end{array} \right\}$$

. The actual cost of edit paths P_0, P_1, and P_2 are 12, 9, and 18, respectively. These costs suggest that the child state c_1, as an intermediate state, is closer to the goal than both its "parent" e_i and its "sibling" c_2. Those costs are consistent with the fact that Drop(0) is a more promising operation than Split(0,' ') from the initial state. (Only one operation—Split(1,':')—is needed to get from c_1 to e_o, whereas three operations are needed to transform c_2 to e_o). This example shows that our proposed heuristic is effective in prioritizing the good operations over the bad ones.

4.2.2 Table Edit Distance Batch

Although TED seems to be a good metric for table dissimilarity, it is not yet a good heuristic function in our problem because (1) it is an estimate of the cost of *table edit path* at a cell level which is on a scale different from our data transformation operations *cost* defined in Definition 4.2 and

[2] Transform((a_1,a_2),(b_1,b_2)) means Transform the cell at (a_1,a_2) in e_i to the cell at (b_1,b_2) in e_o.
Move((a_1,a_2),(b_1,b_2)) means Move the cell from (a_1,a_2) in e_i to (b_1,b_2) in e_o
Delete((c_1,c_2)) means Delete the cell at (c_1,c_2) in e_i.

Algorithm 2: Table Edit Distance Batch

```
   Data: p_final = {u_i1 → v_1, ..., u_i|T2| → v_|T2|}, patterns
         from Table 4
   Result: cost
 1 batch_temp ← ∅;
 2 batch_final ← ∅;
 3 G_type ← Group p_final by table edit operators type ;
 4 for g ∈ G_type do
 5     for p ∈ patterns do
 6         batch_temp ← batch_temp ∪ Group g by p;
 7 while ⋃ batch_final is not a complete edit path do
 8     batch_max ← argmax_{b∈batch_temp} size(b) ;
 9     if batch_max ∩ ⋃ batch_final = ∅ then
10         add batch_max to batch_final;
11     batch_temp ← batch_temp \ batch_max;
12 cost ← 0;
13 for group ∈ batch_final do
14     sum ← 0;
15     for editOp ∈ group do
16         sum ← sum + cost(editOp);
17     cost ← cost + sum/size(group);
18 Return cost;
```

(2) the TED score depends on the number of cells in the example tables The scaling problem in our setting cannot be fixed by simply multiplying the cost by a constant like has been done in other domains [20], because different Potter's Wheel operators affect different number of cells.

We have developed a novel method called **Table Edit Distance Batch (TED Batch)** (Algorithm 2) that approximates the number of Potter's Wheel operators by grouping table edit operations belonging to certain *geometric patterns* into batches and compacting the cost of each batch. The intuition behind this methodology is based on the observation that data transformation operators usually transform, remove or add cells within the same column or same row or that are geometrically adjacent to each other. Consider Split, for example: it transforms one column in the input table into two or more columns, so instead of counting the individual table edit operations for each affected cell, the operations are batched into groups representing the affected columns.

The definitions of the geometric patterns and related data transformation operators are presented in Table 4. For example, "Vertical to Vertical" captures the edit operations that are from vertically adjacent cells (in the same column) in the input table to vertically adjacent cells in the output table. In Figure 10, Deletes of $\mathcal{I}_{1,3,5}$ are a batch of edit operations that follow "Vertical to Vertical" pattern.

To recalculate the heuristic score using this idea, we propose Algorithm 2, which consists of the following three steps. We use Figure 10 and P_0 to demonstrate each step.

Step 1. *(lines 3 –6) Find all sets of edit operations (from the edit path obtained by Algorithm 1) following each geometric pattern. Each set is a candidate batch. Each edit operator could only be batched with operators of same type (e.g., Move should not be in the same batch as Drop); line 3 first groups operations by types.*

In P_0, Transform((1,2),(1,1)) ($\mathcal{I}_2$ to $\mathcal{O}_1$ in Figure 10) should be grouped by pattern "Vertical to Vertical" with Transform((2,2),(2,1)) ($\mathcal{I}_4$ to $\mathcal{O}_3$) and Transform((3,2),(3,1)) ($\mathcal{I}_6$ to $\mathcal{O}_5$). Meanwhile, it could also be grouped by pattern "One to Horizontal" with Transform((2,2),(2,2)) ($\mathcal{I}_2$ to $\mathcal{O}_2$).

Step 2. *(lines 7 – 11) One edit operation might be included in multiple batches in Step 1. To finalize the grouping, Algorithm 2 repeatedly chooses the batch with the maximum number of edit operations, and none of the operations in this batch should be already included batch$_{final}$. The finalization terminates when batch$_{final}$ covers a complete edit path.*

In the example in Step 1, Transform((1,2),(1,1)) will be assigned to the "Vertical to Vertical" group because it has more members than the "One to Horizontal" group.

Step 3. *(lines 13 – 17) The final heuristic score is the sum of the mean cost of edit operations within each chosen batch.*

In this case, the cost of the batch with Transform((1,2),(1,1)), Transform((2,2),(2,1)) and Transform((3,2),(3,1)) will be 1, not 3. Finally, the batched form of P_0 is $\{p_1, p_2, p_3, p_4\}$, where

$$p_1 = \{\text{Transform((1,2),(1,1))}, \text{Transform((2,2),(2,1))}, \text{Transform((3,2),(3,1))}\},$$
$$p_2 = \{\text{Transform((1,2),(1,2))}, \text{Transform((2,2),(2,2))}, \text{Transform((3,2),(3,2))}\},$$
$$p_3 = \{\text{Move((1,2),(1,1))}, \text{Move((2,2),(2,1))}, \text{Move((3,2),(3,1))}\},$$
$$p_4 = \{\text{Delete((1,1))}, \text{Delete((2,1))}, \text{Delete((3,1))}\}.$$

The estimated cost of P_0 is reduced to 4 which is closer to the actual Potter's Wheel cost and less related to the number of cells than using TED alone. Likewise, cost of P_1 is now 3 and cost of P_2 is now 6. In general, this shows that the TED Batch algorithm effectively "scales down" the TED heuristic and reduces the heuristic's correlation to the table size.

4.3 Pruning Techniques for Better Efficiency

If we indiscriminately tried all possible operations during graph expansion, the search would quickly become intractable. However, not all the potential operations are valid or reasonable. To reduce the size of the graph and improve the runtime of the search, we created three *global pruning rules* (which apply to all operators) and two *property-specific pruning rules* (which apply to any operators with certain specified properties). The following pruning rules are designed to boost efficiency; our proposed data transformation program synthesis technique is still complete without them.

Global Pruning Rules — These pruning rules apply to all operations in the library.

- *Missing Alphanumerics* — Prune the operation if any letter (a–z, A–Z) or digit (0–9) in e_o does not appear in the resulting child state. We assume transformations will not introduce new information, thus if an operation completely eliminates a character present in e_o from current state, no valid path to the goal state exists.
- *No Effect* — Prune the operation that generates a child state identical to the parent state. In this case, this operation is meaningless and should be removed.
- *Introducing Novel Symbols* — Prune the operation if it introduces a printable non-alphanumeric symbol that is not present in e_o. If an operator were to add such a symbol, it would inevitably require an additional operation later to remove the unneeded symbol.

Pattern	Formulation (X is a table edit operator)	Related Operators
Horizontal to Horizontal	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i + 1), (x_j, y_j + 1)), \dots\}$	Delete(Possibly)
Horizontal to Vertical	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i + 1), (x_j + 1, y_j)), \dots\}$	Fold, Transpose
Vertical to Horizontal	$\{X((x_i, y_i), (x_j, y_j)), X((x_i + 1, y_i), (x_j, y_j + 1)), \dots\}$	Unfold,Transpose
Vertical to Vertical	$\{X((x_i, y_i), (x_j, y_j)), X((x_i + 1, y_i), (x_j + 1, y_j)), \dots\}$	Move, Copy, Merge, Split, Extract, Drop
One to Horizontal	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i), (x_j, y_j + 1)), \dots\}$	Fold(Possibly), Fill(Possibly)
One to Vertical	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i), (x_j + 1, y_j)), \dots\}$	Fold, Fill
Remove Horizontal	$\{X((x_i, y_i)), X((x_i, y_i + 1)), \dots\}$	Delete
Remove Vertical	$\{X((x_i, y_i)), X((x_i + 1, y_i)), \dots\}$	Drop, Unfold

Table 4: Geometric patterns

Property-specific Pruning Rules — The properties of certain operators allow us to define further pruning rules.

- *Generating Empty Columns* — Prune the operation if it adds an empty column in the resulting state when it should not. This applies to Split, Divide, Extract, and Fold. For example, Split adds an empty column to a table when parameterized by a delimiter not present in the input column; this Split useless and can be pruned.
- *Null In Column* — Prune the operation if a column in the parent state or resulting child state has null value that would cause an error. This applies to Unfold, Fold and Divide. For example, Unfold takes in one column as header and one column as data values: if the header column has null values, it means the operation is invalid, since column headers should not be null values.

4.4 Complexity Analysis

The worst-case time complexity for our proposed program synthesis technique is $O((kmn)^d)$, where m is the number of cells in input example e_i, n is the number of cells in the output example e_o, k is the number of candidate data transformation operations for each intermediate table, and d is the number of components in the final synthesized program. In comparison, two of the previous works related to our project, ProgFromEx and FlashRelate, have worst-case time complexities that are exponential in the size of the example the user provides. ProgFromEx's worst-case time complexity is $O(m^n)$, where m is the number of cells in the input example and n is the number of cells in the output example. FlashRelate's worst-case complexity is $O(t^{t-2})$, where t is the number of columns in the output table.

In practice, we believe the complexity exponential in input size will not cause a severe performance issue because none of the three PBE techniques require large amount of user input. However, if a new usage model arises in the future that allows the user to provide a large example easily, ProgFromEx might become impractical.

4.5 Synthesizing Perfect Programs

Since the input-output example $\mathcal{E}$ is the only clue about the desired transformation provided by the user, the effectiveness of our technique could be greatly impacted by the quality of $\mathcal{E}$. We can consider its *fidelity* and *representativeness*.

Fidelity of $\mathcal{E}$ — The success of synthesizing a program is premised on the *fidelity* of the user-specified example $\mathcal{E}$: the end user must not make any mistake while specifying $\mathcal{E}$. Some common mistakes a user might make are: *typos*, *copy-paste-mistakes*, and *loss of information*. This last mistake occurs when the user forgets to include important information, such as column headers, when specifying $\mathcal{E}$. When such mistakes occur, our proposed technique is almost certain to fail. However, the required user input is small, and, as we show in Section 5.6, our system usually fails quickly. As a result, it is easy for the user to fix any errors. In Section 7, we describe future work that allows tolerance for user error.

Representativeness of $\mathcal{E}$ — Once a program $\mathcal{P}$ is generated given the user input, the synthesized program is guaranteed to be *correct*: $\mathcal{P}$ must transform the input example e_i to the output example e_o. However, we do not promise that $\mathcal{P}$ is *perfect*, or guarantees to transform the entire raw data $\mathcal{R}$ as the user may expect. How well a synthesized program generalizes to $\mathcal{R}$ relies heavily on the *representativeness* of $\mathcal{E}$, or how accurately $\mathcal{E}$ reflects the desired transformation. Our proposed synthesis technique requires the user to carefully choose a representative sample from $\mathcal{R}$ as the input example to formulate $\mathcal{E}$. With a small sample from $\mathcal{R}$, there is a risk of synthesizing a $\mathcal{P}$ that will not generalize to $\mathcal{R}$ (similar to overfitting when building a machine learning model with too few training examples). Experimentally, however, we see that a small number (e.g., 2 or 3) of raw data records usually suffices to formulate $\mathcal{E}$ (Section 5).

Validation — [C2] In Section 1, we mentioned that one way the user can validate the synthesized program is by understanding the semantics of the program. Alternatively, the user could follow the sampling-based *lazy* approach of Gulwani et al. [17] To the best of our knowledge, no existing work in the PBE area provides guarantees about the reliability of this approach or how many samples it may require. Of course, not only PBE systems, but work in machine learning and the program test literature must wrestle with the same sampling challenges. Our system neither exacerbates nor ameliorates the situation, so we do not address these issues here.

5. EXPERIMENTS

In this section, we evaluate the effectiveness and efficiency of our PBE data transformation synthesis technique and how much user effort it requires. We implemented our technique in a system called Foofah. Foofah is written in Python and C++ and runs on a 16-core (2.53GHz) Intel Xeon E5630 server with 120 GB RAM.

We first present our benchmarks and then evaluate Foofah using the benchmarks to answer several questions:

- How generalizeable are the synthesized programs output by Foofah? (Section 5.2)
- How efficient is Foofah at synthesizing data transformation programs? (Section 5.2)
- How is the chosen search method using the TED Batch heuristic better than other search strategies, including BFS and a naïve rule-based heuristic? (Section 5.3)
- How effectively do our pruning rules boost the search speed? (Section 5.4)

- What happens to FOOFAH if we add new operators to the operator library? (Section 5.5)
- How much effort does FOOFAH save the end users compared to the baseline system WRANGLER? (Section 5.6)
- How does FOOFAH compare to other PBE data transformation systems? (Section 5.7)

Overall, when supplied with an input-output example comprising two records, FOOFAH can synthesize perfect data transformation programs for over 85% of test scenarios within five seconds. We also show FOOFAH requires 60% less user effort than a state-of-art data transformation tool, WRANGLER.

5.1 Benchmarks

To empirically evaluate FOOFAH, we constructed a test set of data transformation tasks. Initially, we found 61 test scenarios used in related work including PROGFROMEX [17], WRANGLER [22], Potter's Wheel (PW) [33,34] and Proactive Wrangler (Proactive) [16] that were candidate benchmark tests. However, not all test scenarios discovered were appropriate for evaluating FOOFAH. One of our design assumptions is that the output table must be relational; we eliminated 11 scenarios which violated this assumption. In the end, we created a set of benchmarks with 50 test scenarios[3], among which 37 are real-world data transformation tasks collected in Microsoft Excel forums (from PROGFROMEX [17]) and the rest are synthetic tasks used by other related work.

For test scenarios with very little data, we asked a Computer Science student not involved with this project to synthesize more data for each of them following a format similar to the existing raw data of the scenario. This provided sufficient data records to evaluate each test scenario in Section 5.2.

5.2 Performance Evaluation

In this section, we experimentally evaluate the response time of FOOFAH and the *perfectness* of the synthesized programs on all test scenarios. Our experiments were designed in a way similar to that used by an influential work in spreadsheet data transformation, PROGFROMEX [17], as well as other related work in data transformation [4,24].

Overview — For each test scenario, we initially created an input-output example pair using a single data record chosen from the raw data and sent this pair to FOOFAH to synthesize a data transformation program. We executed this program on the entire raw data of the test scenario to check if the raw data was completely transformed as expected. If the inferred program did transform the raw data correctly, FOOFAH synthesized what we term a *perfect* program. If the inferred program did not transform the raw data correctly, we created a new input-output example that included one more data record chosen from the raw data, making the example more descriptive. We gave the new example to FOOFAH and again checked if the synthesized program correctly transformed the raw data. We repeated this process until FOOFAH found a perfect program, giving each round a time limit of 60 seconds.

Results — Figure 11a shows numbers of data records required to synthesize a perfect program. FOOFAH was able to synthesize perfect programs for 90% of the test scenarios (45 of 50) using input-output examples comprising only 1 or 2 records from the raw data. FOOFAH did not find perfect programs for 5 of the 50 test scenarios. The five failed test scenarios were real-world tasks from PROGFROMEX, but overall FOOFAH still found perfect programs for more than 85% of the real-world test scenarios (32 of 37).

Among the five failed test scenarios, four required unique data transformations that cannot be expressed using our current library of operators; FOOFAH could not possibly synthesize a program that would successfully perform the desired transformation. The remaining failed test scenario required a program that *can* be expressed with FOOFAH's current operations. This program has five steps, which contain two Divide operations. FOOFAH likely failed in this case because Divide separates a column of cells in two columns conditionally, which requires moves of cells following no geometric patterns we defined for TED Batch. The TED Batch heuristic overestimates the cost of paths that include Divide. FOOFAH required more computing time to find the correct path, causing it to reach the 60 second timeout.

Figure 11b shows the average and worst synthesis time of each interaction in all test scenarios. The y-axis indicates the synthesis time in seconds taken by FOOFAH; the x-axis indicates the percentage of test scenarios that completed within this time. The worst synthesis time in each interaction is less than 1 second for over 74% of the test scenarios (37 of 50) and is less than 5 seconds for nearly 86% of the test scenarios (43 of 50), and the average synthesis time is 1.4 seconds for successfully synthesized perfect programs.

Overall, these experiments suggest that FOOFAH, aided by our novel TED Batch heuristic search strategy, can efficiently and effectively synthesize data transformation programs. In general, FOOFAH can usually find a perfect program within interactive response times when supplied with an input-output example made up of *two data records from the raw data*.

5.3 Comparing Search Strategies

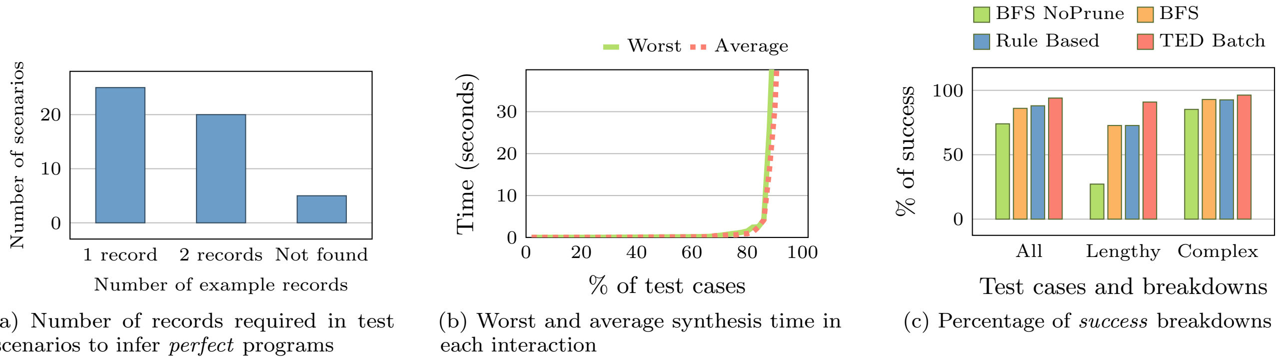
In this section, we evaluate several search strategies to justify our choice of TED Batch.

Overview — We considered Breadth First Search (BFS) and A* search with a rule-based heuristic (Rule), both mentioned in Section 4, and a baseline, Breadth First Search without pruning rules (BFS NoPrune). Based on the conclusion from Section 5.1, we created a set of test cases of input-output pairs comprising two records for all test scenarios. In this experiment, each search strategy was evaluated on the entire test set and the synthesis times were measured. The *perfectness* of the synthesized programs was not considered. A time limit of 300 seconds was set for all tests. When a program was synthesized within 300 seconds, we say FOOFAH was successful for the given test case.

Results — Figure 11c shows that TED Batch achieves the most successes among all four search strategies and significantly more than the baseline "BFS NoPrune" over the full test suite. To understand the performance of the search strategies in different type of data transformation tasks, we examined the data for two specific categories of test cases.

We first checked the test cases requiring lengthy data transformation programs, since program length is a key factor affecting the efficiency of the search in the state space graph. We considered the a program to be *lengthy* if it required four or more operations. Figure 11c shows the success rate for all four search strategies in lengthy test cases. TED Batch

[3]https://github.com/markjin1990/foofah_benchmarks

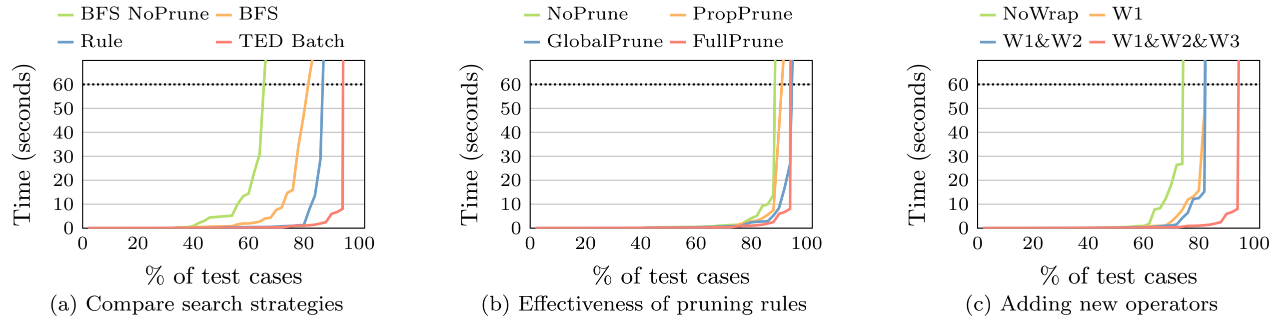


(a) Number of records required in test scenarios to infer *perfect* programs

(b) Worst and average synthesis time in each interaction

(c) Percentage of *success* breakdowns

Figure 11: (a) and (b) show number of records and synthesis time required by Foofah in the experiments of Section 5.1; (c) Percentage of successes for different search strategies in the experiments of Section 5.3.



(a) Compare search strategies

(b) Effectiveness of pruning rules

(c) Adding new operators

Figure 12: (a) Percentage of tests synthesized in $\leq$ Y seconds using different search strategies; (b) Percentage of tests synthesized in $\leq$ Y seconds with different pruning rules settings; (c) Percentage of tests synthesized in $\leq$ Y seconds adding Wrap variants.

achieves the highest success rate of any of the strategies, with a margin larger than that for over *all* test cases. This indicates that our proposed strategy, TED Batch, is effective at speeding up the synthesis of lengthy programs.

Since end users often feel frustrated when handling complex data transformations, we wished to know how TED Batch fared compared to other search strategies on complex tasks. We considered test cases that required the operators Fold, Unfold, Divide, Extract to be *complex*. Figure 11c shows the success rate for those complex test cases. TED Batch outperforms the other three strategies.

Figure 12a shows the time required to synthesize the programs for our set of tests for each search strategy. The TED Batch search strategy is significantly the fastest, with over 90% of the tests completing in under 10 seconds.

5.4 Effectiveness of Pruning Rules

One contribution of our work is the creation of a set of pruning rules for data transformation. We examine the efficiency of Foofah with and without these pruning rules to show how effectively these pruning rules boost the search speed, using the benchmarks from Section 5.1.

Figure 12b presents the response times of Foofah with pruning rules removed. The pruning rules do improving the efficiency of the program synthesis. However, the difference between the response time of Foofah with and without pruning rules is only moderate in size. This is because the search strategy we use—TED Batch—is itself also very effective in "pruning" bad states, by giving them low priority in search. In fact, if we look at "BFS NoPrune" and "BFS" in Figure 12a, the difference between their response time is quite significant, showing that the pruning rules are indeed quite helpful at reducing the size of the search space.

5.5 Adaptiveness to New Operators

A property of our program synthesis technique is its operator-independence, as we discussed in Section 4. To demonstrate this, we compared the efficiency of our prototype, Foofah, with and without a newly added operator: Wrap (defined in Appendix A). Wrap has three variants: Wrap on column x (W1), Wrap every n rows (W2) and Wrap into one row (W3). We examined the responsiveness of Foofah on all test cases as we sequentially added the three variants of Wrap.

Figure 12c shows the response time of Foofah as we add new variants of Wrap, using the same set of test cases as in Section 5.3. The addition of the Wrap operations allowed more test scenarios to be successfully completed, while the synthesis time of overall test cases did not increase. This is evidence that the system can be improved through the addition of new operators, which can be easily incorporated without rewriting the core algorithm.

5.6 User Effort Study

Foofah provides a Programming By Example interaction model in hopes of saving user effort. In this experiment, we asked participants to work on both Wrangler and Foofah and compared the user effort required by both systems.

Overview — We invited 10 graduate students in Computer Science with no experience in data transformation to participate in our user study. From our benchmark test suite, we chose eight user study tasks of varied length and complexity, shown in Table 5. Column "Complex" indicates if a task requires a complex operator: Fold, Unfold, Divide, and Extract. Column "$\geq$ 4 Ops" indicates if a task requires a data transformation program with 4 or more operations.

Before the experiment, we educated participants on how to

Test	Complex	≥ 4 Ops	Wrangler Time	Wrangler Mouse	Wrangler Key	Foofah Time vs Wrangler	Foofah Mouse	Foofah Key
PW1	No	No	**104.2**	17.8	11.6	**49.4** ↘**52.6%**	20.8	22.6
PW3 (modified)	No	No	**96.4**	28.8	26.6	**38.6** ↘**60.0%**	14.2	23.6
ProgFromEx13	Yes	No	**263.6**	59.0	16.2	**145.8** ↘**44.7%**	43.6	78.4
PW5	Yes	No	**242.0**	52.0	15.2	**58.8** ↘**75.7%**	31.4	32.4
ProgFromEx17	No	Yes	**72.4**	18.8	11.6	**48.6** ↘**32.9%**	18.2	15.2
PW7	No	Yes	**141.0**	41.8	12.2	**44.4** ↘**68.5%**	19.6	35.8
Proactive1	Yes	Yes	**324.2**	60.0	13.8	**104.2** ↘**67.9%**	41.4	57.0
Wrangler3	Yes	Yes	**590.6**	133.2	29.6	**137.0** ↘**76.8%**	58.6	99.8

Table 5: User study experiment results

use both Wrangler and Foofah with documentation and a complex running example. During the experiment, each participant was given four randomly selected tasks, covering complex, easy, lengthy, and short tasks, to complete on both systems. Each task had a 10 minute time limit.

Evaluation Metrics — To quantify the amount of user effort on both systems, we measured the time a user spends to finish each user study task. In addition to time, we also measured the number of user mouse clicks and key strokes.

Results — Table 5 presents the measurement of the average user efforts on both Wrangler and Foofah over our 8 user study tasks. The percentages of time saving in each test is presented to the right of the time statistics of Foofah. The timing results show that Foofah required 60% less interaction time in every test on average. Foofah also saved more time on complex tasks. On these tasks, Foofah took one third as much interaction time as Wrangler. On the lengthy and complex "Wrangler3" case, 4 of 5 test takers could not find a solution within 10 minutes using Wrangler, but all found a solution within 3 minutes using Foofah.

Additionally, in Table 5 we see that Foofah required an equal or smaller number of mouse clicks than Wrangler. This partially explains why Foofah required less interaction time and user effort. Table 5 also shows that Foofah required more typing than Wrangler, mainly due to Foofah's interaction model. Typing can be unavoidable when specifying examples, while Wrangler often only requires mouse clicks.

Another observation from the user study was that participants often felt frustrated after 5 minutes and became less willing to continue if they could not find a solution, which justifies our view that a Programming By Demonstration data transformation tool can be hard to use for naïve users.

5.7 Comparison with Other Systems

Foofah is not the first PBE data transformation system. There are two other closely related pieces of previous work: ProgFromEx [17] and FlashRelate [4]. In general, both ProgFromEx and FlashRelate are less expressive than Foofah; they are limited to *layout transformations* and cannot handle *syntactic transformations*. Further, in practice, both systems are likely to require more user effort and to be less efficient than Foofah on complex tasks.

Source code and full implementation details for these systems are not available. However, their published experimental benchmarks overlap with our own, allowing us to use their published results in some cases and hand-simulate their results in other cases. As a result, we can compare our system's success rate to that of ProgFromEx and FlashRelate on at least some tasks, as seen in Table 6. Note that syntactic transformation tasks may also entail layout transformation steps, but the reverse is not true.

5.7.1 ProgFromEx

The ProgFromEx project employs the same usage model as Foofah: the user gives an "input" grid of values, plus a desired "output" grid, and the system formulates a program to transform the input into the output. A ProgFromEx program consists of a set of *component programs*. Each component program takes in the input table and yields a map, a set of *input-output cell coordinate pairs* that copies cells from the input table to some location in the output table.

A component program can be either a *filter program* or an *associative program*. A filter program consists of a mapping condition (in the form of a conjunction of cell predicates) plus a *sequencer* (a geometric summary of where to place data in the output table). To execute a filter program, ProgFromEx tests each cell in the input table, finds all cells that match the mapping condition, and lets the sequencer decide the coordinates in the output table to which the matching cells are mapped. An associative program takes a component program and applies an additional transformation function to the output cell coordinates, allowing the user to produce output tables using copy patterns that are not strictly one-to-one (e.g., a single cell from the input might be copied to multiple distinct locations in the output).

Expressiveness — The biggest limitation of ProgFromEx is that it cannot describe *syntactic transformations*. It is designed to move values from an input grid cell to an output grid cell; there is no way to perform operations like Split or Merge to modify existing values. Moreover, it is not clear how to integrate such operators into their cell mapping framework. In contrast, our system successfully synthesizes programs for 100% of our benchmark syntactic transformation tasks, as well as 90% of the layout transformation tasks (see Table 6). (Other systems can handle these critical syntactic transformation tasks [16, 22, 34], but Foofah is the first PBE system to do so that we know of). ProgFromEx handles slightly more layout transformations in the benchmark suite than our current Foofah prototype, but ProgFromEx's performance comes at a price: the system administrator or the user must pre-define a good set of cell mapping conditions. If the user were willing to do a similar amount of work on Foofah by adding operators, we could obtain a comparable result.

User Effort and Efficiency — For the subset of our benchmark suite that both systems handle successfully (i.e., cases without any syntactic transformations), ProgFromEx and Foofah require roughly equal amounts of user effort. As we describe in Section 5.1, 37 of our 50 benchmark test scenarios are borrowed from the benchmarks of ProgFromEx. For each of these 37 benchmarks, both ProgFromEx and Foofah can construct a successful program with three or fewer user-provided examples. Both systems yielded wait times under 10 seconds for most cases.

	Layout Trans.	Syntactic Trans.
FOOFAH	88.4%	100%
PROGFROMEX	97.7%	0%
FLASHRELATE	74.4%	0%

Table 6: Success rates for different techniques on both layout transformation and syntactic transformation benchmarks

5.7.2 FlashRelate

FLASHRELATE is a more recent PBE data transformation project that, unlike PROGFROMEX and FOOFAH, only requires the user to provide output examples, not input examples. However, the core FLASHRELATE algorithm is similar to that of PROGFROMEX: it conditionally maps cells from a spreadsheet to a relational output table. FLASHRELATE's cell condition tests are more sophisticated than those in PROGFROMEX (e.g., they can match on regular expressions and geometric constraints).

Expressiveness — Like PROGFROMEX, FLASHRELATE cannot express syntactic transformations, because FLASHRELATE requires exact matches between regular expressions and cell contents for cell mapping. Moreover, certain cell-level constraints require accurate schematic information, such as column headers, in the input table. FLASHRELATE achieves a lower success rate than FOOFAH in Table 6. In principle, FLASHRELATE should be able to handle some tasks that PROGFROMEX cannot, but we do not observe any of these in our benchmark suite.

User Effort and Efficiency — FLASHRELATE only requires the user to provide output examples, suggesting that it might require less overall user effort than FOOFAH or PROGFROMEX. However, on more than half of the benchmark cases processed by both FLASHRELATE and FOOFAH, FLASHRELATE required five or more user examples to synthesize a correct transformation program, indicating that the effort using FLASHRELATE is not less than either PROGFROMEX or FOOFAH. Published results show that more than 80% of tasks complete within 10 seconds, suggesting that FLASHRELATE's runtime efficiency is comparable to that of FOOFAH and PROGFROMEX.

6. RELATED WORK

Both program synthesis and data transformation have been the focus of much research, which we discuss in depth below.

Program Synthesis — Several common techniques to synthesize programs have been discussed in Section 4.1: constraint-base program synthesis [15, 21] does not fit our problem because existing logic solvers could not scale to solve a large number of constraints quadratic in the input size; sketching [40] is computationally unfeasible for interactive data transformation; version space algebra [12, 23] is usually applied in PBD systems. Therefore, we formulate our problem as a search problem in the state space graph and solve it using a search-based technology with a novel heuristic—TED Batch—as well as some pruning rules.

Researchers have applied program synthesis techniques to a variety of problem domains: parsers [25], regular expressions [5], bit-manipulation programs [14, 21], data structures [39]; code snippets and suggestions in IDEs [29, 35], and SQL query based on natural language queries [26] and data handling logic [10], schema mappings [2]. There are also several projects that synthesize data transformation and extraction programs, discussed in more detail next.

Data Transformation — Data extraction seeks to extract data from unstructured or semi-structured data. Various data extraction tools and synthesizers have been created to automate this process: TextRunner [3] and WebTables [6] extract relational data from web pages; Senbazuru [7, 8] and FlashRelate [4] extract relations from spreadsheets; FlashExtract [24] extracts data from a broader range of documents including text files, web pages, and spreadsheets, based on examples provided by the user.

Data transformation (or data wrangling) is usually a follow-up step after data extraction, in which the extracted content is manipulated into a form suitable for input into analytics systems or databases. Work by Wu et al. [43–45], as well as FlashFill [12, 38] and BlinkFill [36] are built for syntactic transformation. DataXFormer [1] and work by Singh and Gulwani [37] are built for semantic transformation. ProgFromEx [17] is built for layout transformation, and Wrangler [22] provides an interactive user interface for data cleaning, manipulation and transformation.

Some existing data transformation program synthesizers follow Programming By Example paradigm similar to FOOFAH [4, 12, 13, 17, 24, 36, 43–45]. ProgFromEx [17] and FlashRelate [4] are two important projects in PBE data transformation which have been compared with our proposed technique in Section 5.7. In general, their lack of expressiveness for syntactic transformations prevent them from addressing many real-world data transformation tasks.

7. CONCLUSION AND FUTURE WORK

In this paper, we have presented a Programming By Example data transformation program synthesis technique that reduces the user effort for naïve end users. It takes descriptive hints in form of input-output examples from the user and generates a data transformation program that transforms the input example to the output example. The synthesis problem is formulated as a search problem, and solved by a heuristic search strategy guided by a novel operator-independent heuristic function, TED Batch, with a set of pruning rules. The experiments show that our proposed PBE data transformation program synthesis technique is effective and efficient in generating perfect programs. The user study shows that the user effort is 60% less using our PBE paradigm compared to Wrangler [22].

In the future, we would like to extend our system with an interface allowing the user to easily add new data transformation operators and to explore advanced methods of generating the geometric patterns for batching. Additionally, we would like to generate useful programs even when the user's examples may contain errors. We could do so by alerting the user when the system observes unusual example pairs that may be mistakes, or by synthesizing programs that yield outputs *very similar* to the user's specified example.

8. ACKNOWLEDGMENTS

This project is supported by National Science Foundation grants IIS-1250880, IIS-1054913, NSF IGERT grant 0903629, as well as a Sloan Research Fellowship and a CSE Department Fellowship. We would like to thank our anonymous reviewers for their careful reading of our manuscript and many insightful comments and suggestions. We also thank our user study participants and Andreas Dressel for his ideas.

9. REFERENCES

[1] Z. Abedjan, J. Morcos, I. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. DataXFormer: A robust transformation discovery system. In *ICDE*, 2016.

[2] B. Alexe, B. T. Cate, P. G. Kolaitis, and W.-C. Tan. Characterizing schema mappings via data examples. *ACM Transactions on Database Systems*, 36(4):23, 2011.

[3] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction for the web. In *IJCAI*, 2007.

[4] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *PLDI*, 2015.

[5] A. Blackwell. Swyn: A visual representation for regular expressions. *Your Wish is My Command: Programming by Example*, pages 245–270, 2001.

[6] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment*, 1(1):538–549, 2008.

[7] Z. Chen and M. Cafarella. Automatic web spreadsheet data extraction. In *Proceedings of the 3rd International Workshop on Semantic Search over the Web*, 2013.

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

[9] Z. Chen, M. Cafarella, and H. Jagadish. Long-tail vocabulary dictionary extraction from the web. In *WSDM*, 2016.

[10] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, 2013.

[11] M. R. Gary and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness, 1979.

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

[13] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[14] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Component based synthesis applied to bitvector circuits. Technical report, Technical Report MSR-TR-2010-12, Microsoft Research, 2010.

[15] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[16] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *UIST*, 2011.

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

[18] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[19] D. Huynh and S. Mazzocchi. OpenRefine. http://openrefine.org, 2012.

[20] F. Islam, V. Narayanan, and M. Likhachev. Dynamic multi-heuristic a*. In *IEEE International Conference on Robotics and Automation*, 2015.

[21] S. Jha, S. Gulwani, S. Seshia, A. Tiwari, et al. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[22] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *CHI*, 2011.

[23] T. A. Lau, P. M. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, 2000.

[24] V. Le and S. Gulwani. FlashExtract: A framework for data extraction by examples. In *PLDI*, 2014.

[25] A. Leung, J. Sarracino, and S. Lerner. Interactive parser synthesis by example. In *PLDI*, 2015.

[26] F. Li and H. Jagadish. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment*, 8(1):73–84, 2014.

[27] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[28] S. Lohr. For big-data scientists, janitor work is key hurdle to insights. *The New York Times*, 17, 2014.

[29] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, 2005.

[30] H. Massalin. Superoptimizer: a look at the smallest program. In *ASPLOS II*, 1987.

[31] M. Neuhaus and H. Bunke. *Bridging the gap between graph edit distance and kernel machines*. World Scientific Publishing Co., Inc., 2007.

[32] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. In *ASPLOS*, 2016.

[33] V. Raman and J. Hellerstein. An interactive framework for data cleaning. Technical report, 2000.

[34] V. Raman and J. M. Hellerstein. Potter's Wheel: An interactive data cleaning system. In *VLDB*, volume 1, pages 381–390, 2001.

[35] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev. Refactoring with synthesis. In *OOPSLA*, 2013.

[36] R. Singh. BlinkFill: Semi-supervised programming by example for syntactic string transformations. *Proceedings of the VLDB Endowment*, 9(10):816–827, 2016.

[37] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8):740–751, 2012.

[38] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *Computer Aided Verification*, pages 398–414. Springer, 2015.

[39] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *ESEC/FSE*, 2011.

[40] A. Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[41] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. Data curation at scale: The Data Tamer system. In *CIDR*, 2013.

[42] I. H. Witten and D. Mo. Tels: Learning text editing tasks from examples. In *Watch what I do*, pages 183–203. MIT Press, 1993.
[43] B. Wu and C. A. Knoblock. An iterative approach to synthesize data transformation programs. In *IJCAI*, 2015.
[44] B. Wu, P. Szekely, and C. A. Knoblock. Learning data transformation rules through examples: Preliminary results. In *IIWeb*, 2012.
[45] B. Wu, P. Szekely, and C. A. Knoblock. Minimizing user effort in transforming data by example. In *IUI*, 2014.

APPENDIX

A. DEFINITIONS OF ALL TABLE TRANSFORMATION OPERATIONS

Note that all operators have R as a default input parameter, which represents the input example table. $a_1, a_2, \ldots$ represent the columns. The domains for all column indexes is $\{1, \ldots, k\}$, where k is the number of columns in R.

- **Drop** deletes a column in the table. Its input parameter is the column index i that will be dropped.
  Definition: $Drop(R, i) = \{(a_1, ..., a_{i-1}, a_{i+1}, ..., a_n)|(a_1, ..., a_n) \in R\}$
- **Move** relocates a column from one position to another in the table. Its input parameters are two different column indexes, i, j, meaning column i will be moved before column j in the table.
  Definition: $Move(R, i, j) = \{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_j, a_i, a_{j+1}, \ldots, a_n)|(a_1, \ldots, a_n) \in R,\ \text{if } i < j\}$
  $Move(R, i, j) = \{(a_1, \ldots, a_{j-1}, a_i, a_{j+1}, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)|(a_1, \ldots, a_n) \in R,\ \text{if } i > j\}$
- **Copy** duplicates a column and append the copied column to the end of the table. Its input parameter is the index of the column i to be copied.
  Definition: $Copy(R, i) = \{(a_1, \ldots, a_n, a_i)|(a_1, \ldots, a_n) \in R\}$
- **Merge** concatenates two columns and append the merged column to the end of the table. Its input parameters include column i and column j, representing two columns to be merged, and an optional parameter string d interposed in between. The domain for d is printable non-alphanumerics.
  Definition: $Merge(R, i, j, d) = \{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n, a_i \oplus d \oplus a_j)|(a_1, \ldots, a_n) \in R\}$
- **Split** separates a column into two or more halves from at the occurrences of the splitter. Its input parameters are the index of the column to be splitted and a string d representing the splitter. The domain for d is printable non-alphanumerics.
  Definition: $Split(R, i, d) = \{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, leftSplit(a_i, d), rightSplit(a_i, d))|(a_1, \ldots, a_n) \in R\}$
- **Fold** collapses columns starting from i until the last column in the table into one column in the output table. If b is True, then it will add an additional column with the corresponding header value for each row. Its input parameters are column index i and boolean b. *Definition:* $Fold(R, i, b) = \{(a_1, ..., a_{i-1}, header(k), a_k)|(a_1, ..., a_n) \in R \land k \in \{i, \ldots, n\},\ \text{if } b = False\}$
  $Fold(R, i, b) = \{(a_1, ..., a_{i-1}, a_k)|(a_1, ..., a_n) \in R \land k \in \{i, \ldots, n\},\ \text{if } b = True\}$
- **Unfold** "unflattens" tables and move information from data values to column names.
  Definition: $Unfold(R, i, j)$ takes in two columns, i and j. It creates new columns for each unique value found in column i and takes data in column j as the column headers of new columns.
- **Divide** divides one column into two columns based on some predicate. Its input parameters include a column index i and a predicate. The domain of predicates in our prototype includes "if all digits", "if all alphabets", "if all alphanumerics".
  Definition: $Divide(R, i, predicate) = \{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, a_i, null)|(a_1, ..., a_n) \in R,\ \text{if } predicate = True\} \cup \{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, null, a_i)|(a_1, \ldots, a_n) \in R,\ \text{if } predicate = False\}$
- **Extract** captures the first match of a given regular expression in the cell data of a designated column in each row. Its input parameters include a column index i and a regular expression. The regular expressions are PCRE-like regular expressions, with optional prefixes and suffixes.
  Definition: $Extract(R, i, regex) = \{(a_1, \ldots, a_{i-1}, a_i, match(a_i, regex), a_{i+1}, \ldots, a_n)|(a_1, \ldots, a_n) \in R\}$
- **Fill** assign empty cells with the value from above. Its input parameter is a column index.
- **Delete** removes rows that matches a given predicates. In our case, we restrict **Delete** to remove rows that has an empty cell in a given column. Its input parameter is a column index.
- **Transpose** Transpose the rows and columns of the table. It has no other input parameter.
- **Wrap** Wrap has three options:
  1. Concatenates rows on one column. It has an input parameter of column index i. *Definition:* $Wrap(R, i) = \{(r_{x_1} \cup r_{x_2} \cup \cdots \cup r_{x_n})|r_{x_1}, r_{x_2}, \ldots, r_m \in R, r_m(i)\ equal,\ m = x_1, x_2, \ldots, x_n\}$
  2. Concatenates multiple rows sequentially into one row. It has a input parameter k representing number of rows to be concatenated. The domain of k in our prototype is $\{1, \ldots, 5\}$. *Definition:* $Wrap(R, k) = \{(r_{m*k+1} \cup r_{m*k+2} \cup \cdots \cup r_{m*k+k-1})|r_{m*k+1}, r_{m*k+2}, \ldots, r_{m*k+k-1} \in R, m = 0, 1, \ldots\}$
  3. Concatenates all rows sequentially into one row. It has no other input parameter. *Definition:* $Wrap(R) = \{(r_1 \cup r_2 \cup \cdots \cup r_n)|r_m \in R, m = 0, 1, \ldots, n\}$

B. EXTRA USE CASE EXAMPLES

Example 1 — The following tables show another example use case borrowed from [33] where the end user needs some data extraction and table reshaping. Table 7 is the raw data that contains some human name information. This table is not *atomic* because the second column contains multiple data items, which violates the first normal form (1NF), and hence is not a good relational database design. The user wants to have each person as a single data entry in the table where

the last name is in the first column and first name in the second column as Table 8.

Latimer	George, Anna
Smith	Joan
Bush	John, Bob

Table 7: Input example

Latimer	George
Latimer	Anna
Smith	Joan
Bush	John
Bush	Bob

Table 8: Output example

The program synthesized by our technique is as follows:

```
1 t = split(t, 1, ',')
2 t = split(t, 2, ' ')
3 t = fold(t, 1)
4 t = delete(t, 1)
```

Simple as the task seems, it is indeed complex (as in the motivating example) in that it relies on syntactic transformations to extract each individual first name into a separate column, using Split (line 1-2), and then a layout transformation to reshape the table by collapsing the second column and third column into one column using Fold. Finally, Delete removes the rows with no first name in the second column.

Example 2 — This example shows how our data transformation technique can extract data from less-structured text files that are initially seen as tables with a single cell. Figure 13 is detailed directory listing information output by the "ls -l" linux command[4]. The user wants to extract the file names and the owner names. The synthesized program (Figure 14) first splits the raw data into rows (line 1-2) and then performs operations to extract the desired fields.

-rw-r–r– 1 mjc staff 180 Mar 12 07:18 accesses.txt
-rw-r–r– 1 mjc staff 183 Mar 12 07:15 accesses.txt∼
drwxr-xr-x 5 mjc staff 170 Mar 14 14:14 bin

Figure 13: Input example

mjc	access.txt
mjc	access.txt∼
mjc	bin

Table 9: Output example

```
1 t = split(t, 0, '\n')
2 t = transpose(t)
3 t = extract(t, 0, '\w+',
       suffix = '\ staff')
4 t = split(t, 0, ':')
5 t = drop(t, 0)
6 t = split(t, 0, ' ')
7 t = drop(t, 0)
```

Figure 14: Synthesized program

C. NAÏVE HEURISTIC FUNCTION

The Potter's Wheel operators in our library can be divided into two groups: *one-to-one* and *many-to-many* [34]. One-to-one operations (e.g., Split) transform the same column(s) in each row, and to estimate the required number of one-to-one operations, it is often effective to apply some operator-specific rules on cells in the same row of both input and output tables (lines 4–6). In contrast, many-to-many operations, such as Fold, structurally transform the entire table. We propose Algorithm 3 to iteratively estimate if each Potter's Wheel operator should be used to transform the current state of

[4]https://github.com/cloudera/RecordBreaker/blob/master/src/samples/textdata/filelisting.txt

Algorithm 3: Rule-based Naïve Heuristic

Data: Intermediate Table T_i, Target Table T_o
Result: cost

1. cost $\leftarrow 0$;
2. **if** *# of rows in* T_i = *# of rows in* T_o **then**
3. hscoreSet $\leftarrow \emptyset$;
4. **for** $T_i[k] \in T_i$ *and* $T_o[k] \in T_o$ **do**
5. **for** $p \in$ *one-to-one operator rules* **do**
6. add $p(T_i[k], T_o[k])$ to hscoreSet
7. cost $\leftarrow$ median(hscoreSet);
8. **else**
9. **for** $p \in$ *many-to-many operator rules* **do**
10. cost $\leftarrow$ cost + $p(T_i, T_o)$
11. **if** *existSyntacticalHeterogeneities*$(T_i, T_o) = True$ **then**
12. cost $\leftarrow$ cost + 1
13. **return** cost;

the table to the goal state, and assign the total count as the heuristic score for this intermediate table.

In Algorithm 3, we create estimates for both kinds of operators separately: lines 4–7 estimate how many one-to-one operators are used, while lines 9–12 estimate how many many-to-many operators and additional one-to-one operators are used. Line 9 checks if the number of rows in T_i equals the number of rows in T_o to determine if a many-to-many operator is needed, since we observe that many-to-many operators usually change the total number of rows in the table, while one-to-one operators never do. We evaluate the two cases separately since it is easier to estimate the number *one-to-one* operators that are used if many-to-many operations can be ignored. Take Table 7 and Table 8 as an example, neither of the two cells in the third row in Table 8 has an exact match in cell contents in the third row in Table 7, but both of them are substrings of the first cell in third column of Table 7. Hence, we know that a Split operator is likely used. Similar rules used for other operators are presented in Table 10. We evaluate the cost individually for each row in T_i and T_o and finally take the median of all the costs as the final cost (line 7) as the final estimate.

For many-to-many operators, the estimation becomes harder, because many-to-many operators always perform layout transformations, and depending on the size of the input table and the chosen parameterization for the operator, a many-to-many operator may move a cell to many different locations in the table. Fortunately, we found that each of the many-to-many operator change the shape, width (number of columns) and height (number of rows), of the input table in a unique way, so there is a possibility that we can tell which operator is used by simple comparing the shapes of T_i and T_o. For example, Transpose is a many-to-many operator that flips the table converting columns into rows and vice versa, hence the width of T_i becomes the height of T_o and the height of T_i becomes the width of T_o. Other rules detecting many-to-many operators are shown in Table 11. Note that when more than one many-to-many operators are used in a transformation, none of rules in Table 11 might work, making it hard to make an estimate about which are the operators needed. In this case, we simply assume that two many-to-many operators

Operator	Rules
Drop/Copy	The absolute difference of common cells from $T_i[k]$ and $T_o[k]$
Move	Number of cells from both $T_i[k]$ and $T_o[k]$ but in different positions
Extract	Number of cells from $T_o[k]$ not in $T_i[k]$ but are substrings of cells from $T_i[k]$
Merge	Number of cells from $T_o[k]$ not in $T_i[k]$ but there are substrings of the cells in $T_i[k]$
Split	Number of cells from $T_i[k]$ not in $T_o[k]$ but there are substrings of the cells in $T_o[k]$

Table 10: Rules estimating the number of one-to-one operators used to transform row k in input table T_i to row k in output table T_o

Operator	Rules
Fold	Height of T_o is multiple of height of T_i
Unfold	Height of T_o is smaller than height of T_i but width of T_o is greater than width of T_i
Delete	Height of T_o does not equal to height of T_i but width of T_o equals to width of T_i
Transpose	Height of T_i is the width of T_o and height of T_o is the width of T_i
Wrap	Height of T_i is multiple of height of T_o

Table 11: Rules estimating which many-to-many operators are used to transform input table T_i to output table T_o

are used, because it is rare that two or more many-to-many operators are used in a real-world data transformation task.

When many-to-many operators are used in a transformation task, one-to-one operators can also be used (e.g., the example in Section 2). However, given the fact that many-to-many operators may move the cells to any position in the output table, estimating one-to-one operators row by row like is done lines 2–7 using the rules form Table 10 is not feasible. We therefore give a rough estimate: assume at least one *one-to-one* operator is used if there is a cell in T_o without an exact match on cell content in T_i (checked by $existSyntacticalHeterogeneities$ in line 11).

Algorithm 4: Table Edit Distance Algorithm

```
Data: Intermediate Table e_x = {u_1, u_2, ..., u_|e_x|},
      where {u_1, u_2, ..., u_|e_x|} is a sequence of cells
      from e_x; Example Output Table e_o =
      {v_1, v_2, ..., v_|e_o|}, where {v_1, v_2, ..., v_|e_o|} is a
      sequence of cells from e_o
Result: cost, edit path
1  Open ← ∅;
2  for w in e_o do
3      add Transform(u_1, w) to Open;
4  add Delete(u_1) to Open;
5  while True do
6      p_min ← argmin_{∀p∈Open} cost(p) ;
7      Remove p_min from Open;
8      if p_min is complete edit path then
9          Return cost(p_min), p_min
10     else
11         Let p_min = {u_1 → v_i, ..., u_k → v_ik};
12         if k < | e_x | then
13             for w ∈ e_o\{v_1, ..., v_k} do
14                 add p_min ∪ {Transform(u_{k+1}, w)} to Open;
15             add p_min ∪ {Delete(u_{k+1})} to Open;
16         else
17             add p_min ∪ ⋃_{w∈e_o\{v_i1,...,v_ik}} Add(w) to
               Open;
```

D. OPTIMAL TED ALGORITHM

In Algorithm 4, we show an optimal Table Edit Distance Algorithm. The set *Open* contains all partial edit paths (line 1). In each iteration, it selects the most promising partial edit path with minimum total cost (line 7). It creates a set of successors of this path by taking an unprocessed cell u_i from input table e_x and substituting it with all unprocessed cells w from output table e_o and then deleting the cell u_i (lines 13–17). When there are no more unprocessed cells in e_x, all remaining cells in e_o will be added (line 19). The algorithm terminates when there is an complete edit path (lines 9–10). Finally, Algorithm 4 returns the cost of the cheapest path.