BlinkFilLO Final Technical Report

Landen Doty*
landoty@ku.edu
University of Kansas
Lawrence, Kansas

Jaxon Avena*
jaxon.avena@ku.edu
University of Kansas
Lawrence, Kansas

ABSTRACT

This project is an alternative implementation of BlinkFill, a Programming by Example (PBE) tool for spreadsheet string transformations in the Microsoft Office application Excel. Introduced in 2016 as a successor to the renowned *FlashFill*, BlinkFill makes a number of enhancements to provide improved quality of results as well as performance. Despite the widespread use of *FlashFill*, and its likely integration of BlinkFill, there exists no comparable utility for LibreOffice Calc, an open-source alternative to Excel. As such, our work aims to fill this gap by implementing BlinkFilLO, an open-source Calc extension for automatic programming of spreadsheet string transformation formulas.

This report serves as technical documentation of our work in re-implementing BlinkFill's core algorithms, designing our plug-in, and evaluating its performance relative to the original publication.

CCS CONCEPTS

• **Software and its engineering** → **Automatic programming**; **Programming by example**; *Domain specific languages.*

KEYWORDS

Program Synthesis, Spreadsheet Formula, Automatic Programming

1 INTRODUCTION

Our project's goal is to provide an open source extension to LibreOffice Calc that allows users to quickly process string-based spreadsheet data using the algorithms and data structures presented by BlinkFill. Specifically, we have implemented an integrated synthesis tool for Calc that will take, as input, string transformation examples via spreadsheet data then generates, as output, programs in BlinkFill's Domain Specific Language (DSL). This program is then translated into LibreOffice Calc's formula DSL. The synthesized programs are then available in the spreadsheet context to be applied to subsequent rows.

BlinkFill's key contribution is the use of *adjacent* spreadsheet data to learn shared substructures among the provided input-output examples used for formula synthesis [1]. When selecting a set of rows in the spreadsheet, BlinkFill will generate a specialized graph data structure for each cell, representing all tokens in the DSL matched within a cell's data. Each cell graph, colloquially referred to as an *Input Data Graph*, is intersected with other cell graphs in the spreadsheet column to produce a concise encoding of the shared structure among a single column. Multi-column inputs require each column Input Data Graph to be joined with neighboring column graphs to produce a single encoding of shared structure among all input data [1].

Table 1: Input-Output Spreadsheet Data

Input	Output
Landen, Doty	LD
Jaxon, Avena	JA
John, Smith	JS
Vince, Carter	VC
Samantha, Burke	CC

The Input Data Graph generated for the entire input is used to produce a second specialized graph structure, referred to only as the *DAG* [1]. Using the Input Data Graph, a DAG is learned for each output item in the selected spreadsheet data and encodes all possible DSL expressions that transform the input data into the output data. That is, any path from the input node in the DAG to the terminal node produces a special sub-string concatenation formula that produces the output data given any respective input row.

Table 1 provides a few simple substring input-output examples. BlinkFill would learn two primary patterns among the input:

- Two proper case substrings separated by a ", " string literal
- A capital letter; a lowercase substring; a ", " string literal; another capital letter; and a final lower case substring

This information would be encoded in the Input Data Graph, and then the DAG would learn that the desired outputs can be produced by concatenating the two capital letters. In BlinkFill's DSL, the synthesized formula would be as such:

```
Concat(SubStr((CAPS, (1,-2), Start), (CAPS,
(1,-2), End)), SubStrExpr((CAPS, (2,-1),
Start), (CAPS, (2,-1), End)))
```

Our synthesis plugin implements each of these components (Input Data Graph, DAG, and DSL synthesis) in addition to a heuristic-based translation from BlinkFill's DSL to Calc-specific formula functions. The remainder of this report provides the theoretical underpinnings and implementation-specific details of our work. In addition, we perform an evaluation of our implementation's performance on a standard PBE benchmark and provide our results. Finally, we include a discussion of existing limitations of our plug-in, lessons learned, and future work.

In summary, this report documents the following:

*Both authors contributed equally to this work.

EECS700, December 13, 2024,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

Table 2: DSL Base Tokens

Token Name	Regular Expression
ProperCase	[A-Z][a-z]+
Caps	[A-Z]+
LowerCase	[a-z]+
Digits	[0-9]+
Alphabets	[A-Za-z]+
Alphanumeric	[A-Za-z0-9]+
WhiteSpace	\s+
Start	^
End	$
ProperCaseWSpaces	ProperCase(ProperCase WhiteSpace)*
CapsWSpaces	Caps(Caps WhiteSpace)*
LowerCaseWSpaces	LowerCase(LowerCase WhiteSpace)*
AlphaWSpaces	Alphanumeric(Alphanumeric WhiteSpace)*

- Theoretical overview of BlinkFill's internal data structures and algorithms
- Our implementation of the synthesis component and Calc plug-in
- Accuracy and runtime performance evaluation
- Discussion of limitations and takeaways from the project and course

2 THEORY OVERVIEW

As stated, the original BlinkFill work introduced two key graph data structures used in synthesizing string transformation formulas for spreadsheet data. Effectively, the **Input Data Graph** encodes common structures among input data and the **DAG** succinctly represents all string formulas that may produce the provided outputs given the inputs.

2.1 Input Data Graph Generation

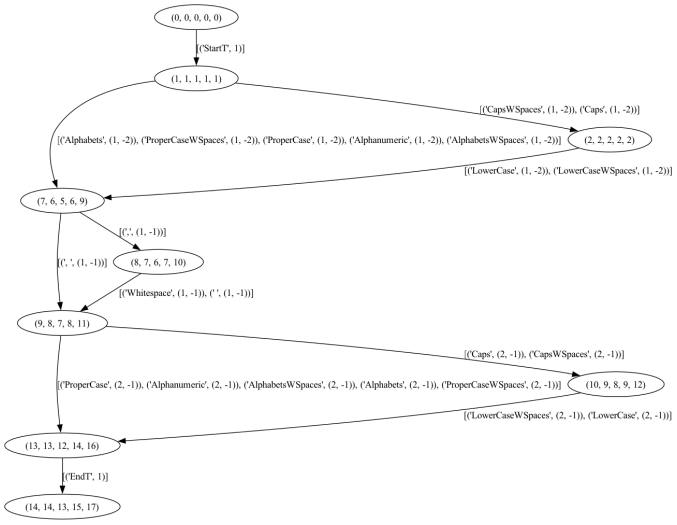

Figure 1: IDG Generated from Example

The synthesis procedure first generates an Input Data Graph (IDG) for each row in the provided input, iteratively intersecting the resulting IDG with the neighboring rows. The specific algorithms are detailed in the original publication, but the general procedure for generating an IDG is as follows. For each substring in an input string, record the position and occurrence (from beginning and end) of each token matched in the base tokens in Figure 2. In addition, the procedure also records each literal substring and its index and occurrence. Nodes in the IDG represent individual indices in the input string, and the edges are tokens that transition one index to another.

As an example, Figure 1 is the IDG generated from the example in Table 1. The tuples on each node represent the index in each input string - that is, the node "(14,14,13,15,17)" confirms the length of each of the input strings in Table 1. Further, each edge is labeled with a *list* of token matches in Table 2 that matches the indices of the connected nodes. For instance, the edge from "(9,8,7,8,11)" to "(10,9,8,9,12)" matches the second capital letter. Note, the tuple "(2,-1") says that this match is the second occurrence from the beginning of the string and the first occurrence from the end of the string.

2.2 DAG Generation

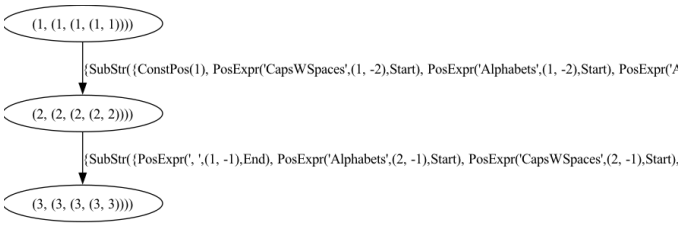

Figure 2: DAG Generated from Example

Following IDG generating, the DAG data structure is "learned" by iterating over each substring in the *output* strings and building a concatenation expression using sub-expressions that generate each substring. Thus, each node in the DAG is labeled with the index of the *output* string it represents and the edges are labeled with sub-expressions in the BlinkFill DSL that generate the substring on the indices.

Following our previous examples, Figre 2 includes a snippet of the DAG generated from the IDG in Figure 1 and example data in Table 1. We cropped a few of the sub-expressions on each edge for formatting, but included the "CAPS" position expressions we anticipated in Section 1. Thus, we see that there a number of simple expressions that generate the expected output string. For instance, we can use the position expression `PosExpr('CAPSWSpaces', (1,-2), Start)` to extract the capital letter of each first name. Here, "Start" means that the position should be taken *preceding* the matched pattern.

2.3 Formula Synthesis

After generating a DAG for the input-output examples, synthesizing a formula is as simple as extracting a path from the start to end node. The original BlinkFill work uses a modification of Dijkstra's Algorithm to determine a *maximum* weighted path. Edges

Table 3: Method Overrides for Set Intersection

Sub-Expression	Params.	__hash__	__eq__
ConstStr	str	hash(str)	str
ConstPos	pos	pos	pos
SubStr	str,l,r	hash(l) $\wedge hash(r)$	(l,r)
Pos	tok,idx,dir	hash(tok+idx+dir)	(tok,idx,dir)

representing constant strings have lower weights than edges representing position expressions and thus dis-incentivize the occasional random constant string [1].

3 IMPLEMENTATION

In general, we follow the algorithms for generating the IDG, learning the DAG, and extracting a formula, as defined and described in the BlinkFill paper [1]. We implement both graphs as nested Python dictionaries, where a start node is a key in the top-level dictionary with a second dictionary containing each connected node and the edge labels as the internal values. We represent the base tokens as an `enum`, where each value is a compiled regular expression. Enums in Python are special singleton classes, so this only incurs a single expense to compile the regular expression when the synthesizer is invoked rather than compiling them each time they are used. Finally, each sub-expression in the DSL is represented as a `class` with special overridden methods for convenient set intersections. We discuss this further in the following section.

Our project can be found on GitHub at the following link https://github.com/landoty/blinkfilLO.

3.1 Synthesis Driver

The entirety of our synthesis driver is implemented in pure Python with no third-party dependencies. This choice is due to the fact that LibreOffice executes our plugin in an embedded Python interpreter that does not respect package installations from the user's environment.

3.1.1 Intersection and Overrides. The most involved component of our synthesis driver is in the intersection of IDGs and DAGs. For instance, intersecting two DAGs requires our implementation to create a Cartesian product of the nodes in the two graphs then intersects each edge by recursively intersecting each sub-expression on the edges *and* their arguments. This process is tedious; however, we were able to alleviate it by representing most collections as `set` objects rather than `list` objects. In doing so, we are able to utilize the built-in set intersection operator `&`.

Python implements `set` objects as dictionaries and intersection utilizes the `__hash__` and `__eq__` methods. By default, these methods return the unique identifier of the object and equivalence of the identifier with another, respectively. It is not reasonable to leave these implementations default for our sub-expression objects as two objects representing the same expression should return the same hash and be found equal to each other. As such, we provide special implementations for `__hash__` and `__eq__` for each sub-expression object in the DSL.

Table 4: BlinkFill to Calc DSL Conversions

BlinkFill	Calc
ConstStr	"string"
CostPos	int<pos>
SubStr	MID(<in>, l, r-l)
PosExpr	SEARCH(REGEX(<in>,tok„idx), <in>)
StartT	0
EndT	len(<in>)

Table 3 summarizes each of these overrides; for brevity, the `__eq__` column notes the fields that are compared for the equivalence of two sub-expressions. Note that the hash override for `SubStr` expressions performs a bit-wise XOR on the resulting hashes of the left and right position expressions. Both of these positions are a set of `Pos` expression objects, so we first derive their hashes before performing the XOR.

3.1.2 Formula Translation. The result of our synthesis process is a formula in the BlinkFill DSL; however, our target execution environment is the Calc formula DSL interpreter. As such, we also needed to convert formulas in the BlinkFill DSL to formulas in the Calc DSL. We perform this heuristically (or ad-hoc-ly) as there is not an ideal one-to-one semantic conversion.

We summarize this conversion process in Table 4. For `SubStr` expressions, we use the `MID` function which takes an absolute index as the second argument and an *offset* as the third argument. This requires us to retrieve the formula for the left `PosExpr` twice in order to calculate the offset. For each `PosExpr`, we must first perform the regular expression match then search for that substring in the input. We make a slight optimization when matching for the `StartT` and `EndT` tokens, returning the absolute index 0 and the length of the input, respectively. The generated Calc formula from our running example is as follows:

> CONCAT(MID(A1,SEARCH(REGEX(A1,"[A-Z][a-z]+(+[A-Z][a-z]+)*"„1),A1,1),SEARCH(REGEX(A1,"[a-z]+(+[a-z]+)*"„1),A1,1)-(SEARCH(REGEX(A1,"[A-Z][a-z]+(+[A-Z][a-z]+)*"„1),A1,1))),MID(A1,SEARCH(REGEX(A1,"[A-Z][a-z]+(+[A-Z][a-z]+)*"„2),A1,1),SEARCH(REGEX(A1,"[a-z]+(+[a-z]+)*"„2),A1,1)-(SEARCH(REGEX(A1,"[A-Z][a-z]+(+[A-Z][a-z]+)*"„2),A1,1))))

Clearly, this is a much more complicated formula than we would expect, but upon applying it in Calc, we receive the intended output[1].

Converting formulas from one DSL to the other is relatively simple with these rules in place. Since top-level expressions are simply concatenations of multiple `SubStr` or `ConstStr` sub-expressions, we just iterate over the list (path extracted from the DAG) and convert each sub-expression. As such, a recursive conversion is only required when retrieving the Calc formulas for the left and right positions in a `SubStr` expression.

[1] If copying this formula from the paper into Calc directly, some tokens may be interpreted incorrectly. For instance the double comma may be copied as a single token. If you receive an error, it may be easier to run the tool and copy the formula from its output. We provide this running example under the "report" subdirectory in the repo.

3.2 Plugin

LibreOffice Calc utilizes an extension framework with support for Java, JavaScript, Python, and C++. Already working in Python, we opted to write the plugin in Python as well. We planned to implement our project as an extension to Calc, bypassing the need to recompile the entire application with BlinkFill as a core utility. Unfortunately, there was confusion when attempting to package our script. The sources of documentation on the subject seemed to contradict one another in various areas and leave out necessary details in others. It was unclear as to what path we should take in embedding BlinkfilLO directly into Calc. While some writings directed you to add custom plugins via macros, others suggested libraries, others extensions, or even some sort of combination of them. In order to have some sort of functioning connection from Calc to script and back by the time of our presentation, we decided to bridge them directly from the CLI. We utillized the UNO (Universal Network Objects) API, a component technology. The UNO component model is the base of the Office API, and allows us to do necessary actions such as reading from selected cells in Calc, and writing back to others with our synthesized program. To do so, we simply run

soffice –accept="socket,host=localhost,port=2002;urp;"

from the command line, which connects to, or starts, a running instance of Calc via the designated port. A connection script then resolves the context and allows for read/write actions to take place. After reading the example data from the spreadsheet, the connection script prepares a JSON object that is then passed to the synthesizer.

4 EVALUATION AND RESULTS

To evaluate the performance and quality of our implementation, we utilized a subset of the PROSE benchmarks, which can be found at https://github.com/microsoft/prose-benchmarks. Specifically, we selected benchmarks from the Transformation.text set which include only the "concatenation", “multicol”, and “substring” features. This subset is representative of all the program types that BlinkFill supports [1]. In total, this selection includes 109 synthetic string transformation benchmarks with the following distribution:

- 97 Substring
- 34 Concatentation
- 8 Multicolumn

Given our issue in directly integrating the synthesis component via a plug-in to LibreOffice Calc, we include a script in the GitHub repository that runs our entire benchmark set and records those that 1) generate a formula, 2) timeout, or 3) crash. We set our timeout threshold to one second, as most benchmarks from the BlinkFill paper completed in 150ms or less [1]. For the benchmarks that generate a formula, we run our process described in Section 3.2 to manually verify that the formula produces the intended output.

4.1 Synthesis Results

During our presentation we reported that a total of 63 benchmarks passed with 24 timing out and 22 failing/crashing. Since then, we were able to identify a few bugs in the source which led to an additional 3 successful benchmarks. We summarize our current results in Figure 3.

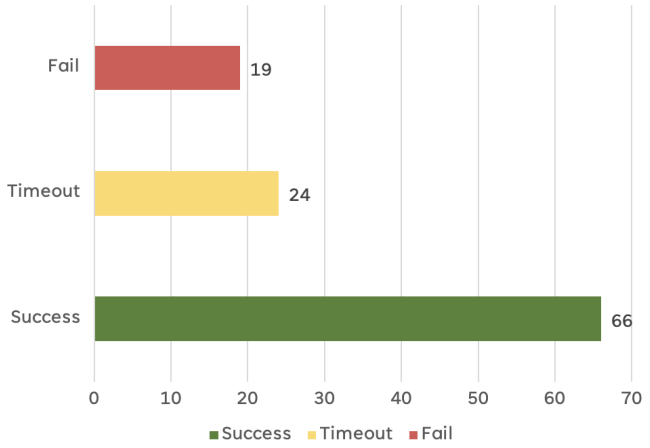

Figure 3: Overall Benchmark Results

4.2 Performance Results

Figure 4: Successful Synthesis Execution Time

We also recorded the overall runtime of our synthesis algorithm for each of our successful benchmarks. Originally, we had reported an average runtime of 0.088s (or 88ms); however, after some additional optimizations, we were able to achieve an average of 0.077s (or 77ms). Comparatively, the BlinkFill work has an approximate runtime of 0.06s (or 60ms), so our implementation is only marginally slower [1]. Given that we implement our project in Python while BlinkFill is implemented in a likely optimized version of C#, we are satisfied with these results.

4.3 Runtime Profiling

In the previous section we mentioned that we performed additional optimizations that allowed us to achieve an improved runtime performance. Despite these improvements, we still have a number of benchmarks which timeout. Further, only one additional benchmark passes when increasing our timeout threshold to two seconds from one.

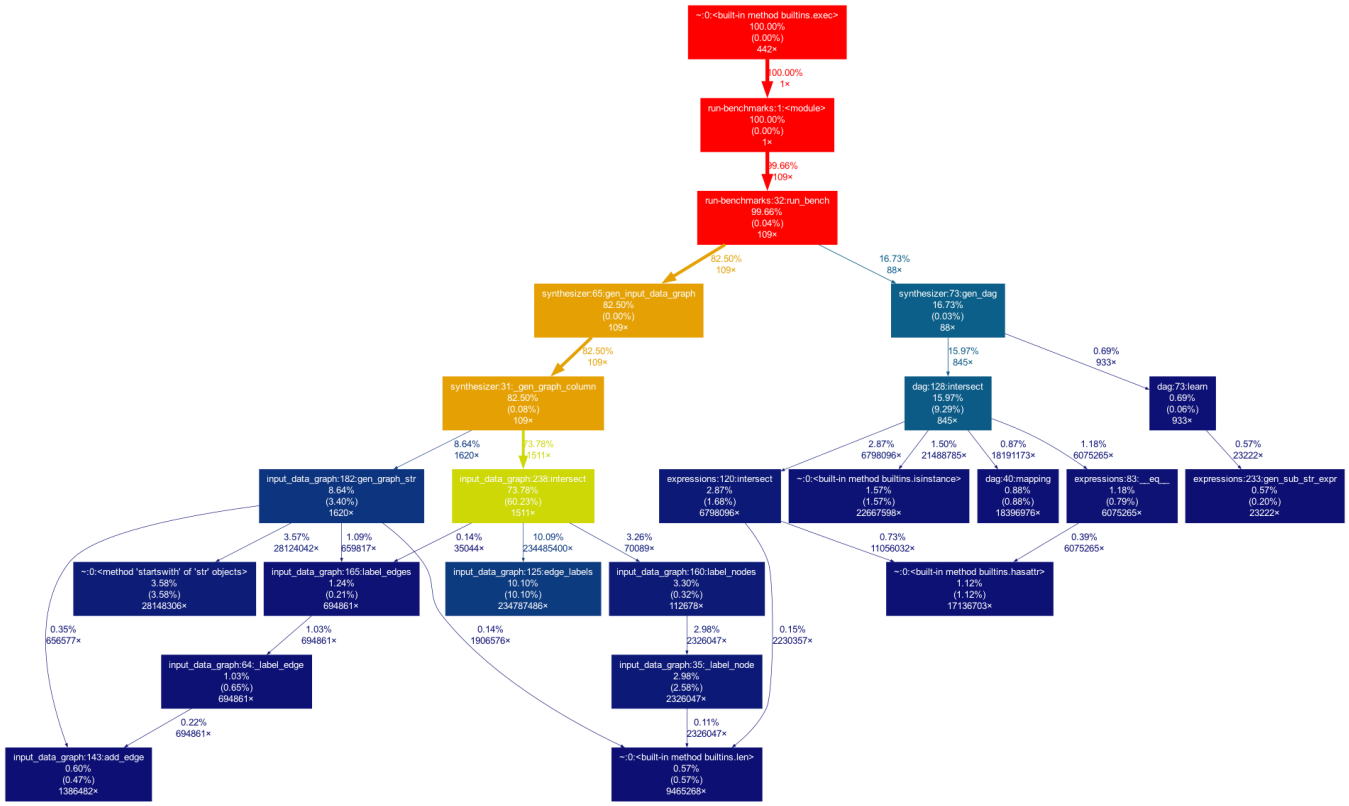

Figure 5: Runtime Execution Profiling

As part of our optimization process, we performed runtime execution profiling. Initially, our profiling highlighted a number of hot functions and blocks that we were able to quickly optimize away. Now, with these optimizations implemented, we are left with the performance profile graphically depicted in Figure 5. As this figure shows, the two most consuming procedures are IDG intersection and DAG intersection. This is expected, given that these were the most demanding components to implement.

5 FUTURE WORK

While we are satisfied with our ability to implement a synthesis tool, like BlinkFill, on the "first" attempt, our results provide important insights for future work moving forward.

5.1 Full Plug-In Integration

Most immediately, we would like to see our plug-in fully implemented and integrated within Calc. Our primary goal was to provide a formula synthesis capability to the open-source community, so this would remain a top priority.

5.2 Performance Improvements

Clearly, there are potential optimizations to be made in the intersection procedures for the Input Data Graph and DAG structures. Figure 5 highlights that over 70% of execution time, across all benchmarks, is spent in IDG intersection. If we were to pursue this project further, the performance of intersection would be of top priority.

5.3 Failure Analysis

At this point, we have not fully analyzed the remainder of the failing benchmarks to understand *why* our synthesis implementation crashes. The bug fixes mentioned in Section 4.1 were due to incorrectly calculated offsets and issues parsing strings with special Regex characters such as brackets and slashes. We have since addressed these issues and would need to perform additional investigation using the debugging utilities we have implemented (Graphviz output, primarily).

6 CONCLUSION

We have documented and demonstrated a successful re-implementation of BlinkFill towards an open-source plug-in for LibreOffice Calc. Through our learning from the course, we were able to accurately, and with relatively acceptable performance, produce a synthesis tool that learns substring transformation formulas for processing spreadsheet data. Our solution, BlinkFilLO, shows that prototypes from the synthesis research community can be successfully developed as user-facing utilities and we provide a baseline implementation towards a fully-functional tool. This cummulative result

communicates our thorough understanding of the course material as well as our ability to digest active research in the space.

Given its early success and our integration of debugging and profiling utilities, we believe our work can be easily followed to produce a final, user-friendly plugin for LibreOffice. As scuh, we intend to continue development and provide BlinkFilLO as a first of its kind plugin for the open-source community.

REFERENCES

[1] Rishabh Singh. 2016. BlinkFill: semi-supervised programming by example for syntactic string transformations. *Proc. VLDB Endow.* 9, 10 (June 2016), 816–827. https://doi.org/10.14778/2977797.2977807