Sketched Flash Fill **Hadi Ravanbakhsh**

1. Introduction

Flash fill component of Excel 2013 is one of the most interesting parts of this version. The algorithm uses input-output examples to synthesize a specific program for a specific purpose (given those kinds of inputs, it can generate expected outputs for other inputs). More precisely, the program is a substring extractor. The algorithm extracts different substrings from the input and concatenates them to build the output. Its best feature that make this product commercial, is its simplicity. The user just gives some sample inputs and outputs and the rest is done by the algorithm. However, because very few information is available as input, it has high complexity and as a result, the author guesses that Excel developers simplified some parts to get a good performance. This causes some shortcomings specially when there are loop patterns in the examples.

In this project, the author tries to take a look at loop patterns from another perspective. Also in order to make the synthesizer more powerful, the author assumes more information is available about the inputs. In fact, it is assumed that the user provides more information to reduce the search space (This approach is usual in synthesizing problems). This increases the performance and decreases the number of iterations needed for synthesizing and all in all, the author introduces this approach as an alternative to “Flash Fill”.

2. Synthesizer Interface

In this section, it is explained how the procedure works. First the user gives the template for the output format, along with a “complete” input-output example. After that, he/she provides “normal” input-output examples. In addition, it is assumed that the output is provided not just as a string, but as a set of substrings in a specific format. This procedure is illustrated in the following example:

The user has some references in a paper and want to extract the last name of authors and initials of their name. He/she also want the year for each publication. He/she first provides the output format like below:

{**[**last name**]** **{[**Initial**]**. **} }** - **[**year**]**

Every thing between { and } means that it can happens several times (It is a loop). The string between [and] is a name for a wanted substring in input. And the rests are constant parts of the output.

More formally, syntax of the format is as follows:

```
Language   ->   Chars Language | {Language} Language | [Name] Language | Epsilon
Name       ->   [a-zA-Z\ ]+
Chars      ->   ( [^ \ { } [ ] ] | \\ | \{ | \} | \[ | \] )*
```

\\, \{, \}, \[and \] are \, {, }, [and] respectively.

After providing the output template, the user provides a “complete” input-output example, i.e. for each loop in the template there is at least one instance of the loop, that has at least two elements.

For example, having the above sketch and input

“[12] Y. Zhang, P. L. Jones, and R. Jetley, A Hazard analysis for a generic insulin infusion pump, J. Diabetes Sci. and Tech., March 2010”

the user provides the output in a similar format to the output format:

{[Zhang] {[Y]} [Hones] {[P][L]} [Jetley] {[R]}} [2010]

More formally the syntax of the output is as follows:

```
Language   ->   {Language} Language | [SubString] Language | Epsilon
SubString  ->   ( [^ \ { } [ ] ] | \\ | \{ | \} | \[ | \] )+
```

This example is called a complete example, because for each loop, there is an instance that has at least two element:

Loop 1 : **{****[**last name**]** **{[**Initial**]**. **} }** --> there are more than two authors
Loop 2 : **{[**Initial**]**. **}** --> there is an author (instance of the loop) with more than one initial

After providing the “complete” input-output example, the user provides other examples with no restrictions.

As you see, instead of asking for a string, the procedure asks for a set of strings. In fact, this can be done in an interactive environment to make the procedure more user-friendly. And finally, the algorithm will provide an expected outputs for a different inputs according to the format.

3. Algorithm

In order to solve this problem, in the first step, the whole problem is divide into smaller problems by solving each substring extraction problem, independently. Considering the above example, the procedure finds an algorithm to extract “last name”s and the whole procedure is independent of the work that is done for extracting “initial”s. These substrings classes are called “segments”. Thus, in the example, first the problem is solved for “lastname” segment, then “initial” segment and finally “year” segment and once this is done, the output is generated using the output format.

3. 1. Positions

The algorithm takes similar approach to “Flash Fill” algorithm for lower levels pattern extraction. Concisely, some tokens like digits, alphabets and special characters like “,” and their NOT (for example [^0-9]) are provided. Then, each position can be represented as a set of possible “C-Position” and “RE-Position”. The “C-Position” means constant position. The “RE-Position” used here is a combination of a regular expression and a point in that regular expression. The regular expression has closure only in token level and doesn’t have “OR” operator! Therefore, such regular expressions can be built as a sequence of tokens, because only concatenation is arbitrary. Dynamic programming is used in the implementation to generate these sequences as fast as possible. And the point in the regular expression determines our desired position, when the regular expression matches the input.

For example, given “abc def06” as input, RE-Position “[a-z].[0-9]” matches “def06” and the place of dot in this matching is between “f” and “0”. This means that this RE-Positions represents the 7th position “abc def06”. To make this more powerful, the times of occurrence is added to the RE-Position. For example, having “book08 store3 job10” as input, the 13th position can be represented as “position of ‘.’ in the second match of [a-z][0-9]**.**[^0-9] over input.

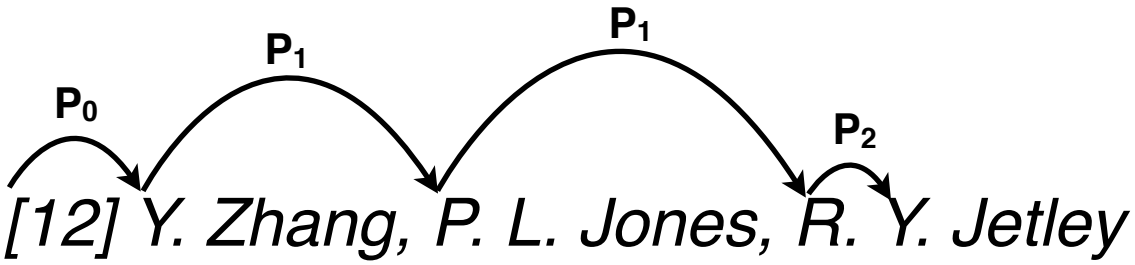
3. 2. Patterns

Now that one can represent the positions, the start position and end position of each instance of a segment (substring of interest) can be generated. However, in order to generalize these positions for all instance of a specific segment, “Patterns” are defined. For each segment, two patterns are used to represent the start position and end position of each segment instance.

Considering the loops, each segment has a depth. For our example, the “last name” segment has depth 1, “initial” segment has depth 2 and “year” segment has depth 0. Each of the two patterns for a segment with depth d is defined as follows:

$P_0 + w_1 P_1 + ... + w_d P_d$

, where each P_i is a set of Positions and w_i is a non-negative number. For example, to represent the start position of second initial of the third author, one assigns w_1 to 2 (3 - 1) and w_2 to 1 (2 - 1). And to find such position, the following procedure is done:



Starting from beginning of the string, first we find a position P_0. For the rest, starting from current position, next position (P_i) is searched for w_i times ($1 \leq i \leq d$), till desired position is reached.

Notice that when one wants to find a position P_i ($i > 1$), he/she needs to make sure that $P_0 + w_1 P_1 + ... + w_i P_i$ is before $P_0 + w_1 P_1 + ... + (w_{i-1} + 1) P_{i-1}$. It means that loops should not have any overlap. Having this condition, helps to find compatible solutions and to throw away unwanted candidates.

The goal of the algorithm is to find two patterns for start and end position of each segment. Finding a pattern means finding P_is, because w_is will be assigned when the patterns are being used. In order to find all compatible patterns, first, the input is searched for all instances of a segment in the output. Then the algorithm looks for feasible combination of positions. In general, there may be more than one solution. For example, if the input is “a b c a b” and the output is “{[a][b]}”, there are three possible way to find a and b in the input where the positions are feasible and there is one possible way to find a and b while it is not feasible. For each feasible combination of these positions, a pattern (P_0, P_1, ... , P_d) is built, where each P_i is a set of Positions.

3. 3. Adding Examples

Because of having a complete example in the beginning, one can find all P_is from the first example and all necessary information is available. After that, for each example, there is no need to build new patterns and one just removes all incompatible patterns in the current feasible pattern set. This way, there is no need to find different set of patterns for different input and find the desired program with intersection. As a result, this approach does not

support “OR” operation and all classifications techniques used in Flash Fill. This means that this method fails to detect totally different structures.

4. Conclusion and Future Work

While flash fill is a user friendly tool, it doesn’t work as it is specified theoretically. For example Excel 2013 couldn’t extract even last name of authors in the above example. This shows that such method is not practical for a little complicated situations. In this project, the author tries to introduce a less user friendly approach which is more powerful. Here it is explained why it is more powerful.

First of all, using this new approach one can handle the inner loops and this can be really helpful when there are more complicated structures in the data. Second, the result are better. It could solve all the examples with only two input-output examples (one of them is the complete example). This suggest an alternative approach for users, where instead of providing many example, they try to express their desired output more precisely and they should do this for only two or three times.

And finally, our approach is better in terms of memory and performance. This is because:

1. This method does not use DAGs and instead set of positions are used where the size of such set can be really small in practice. And also once pattern data structure created, its size will decrease, while in flash fill, they use DAGs and with adding each example, the size of DAG will increase with a factor of “size of DAG for new example”.
2. There is no need to search for all possible kinds of loops, where lots of them are not desired.

For the future, the author should optimize the algorithm to decrease the time complexity even more, though the non-optimized version provides solution in less that a second. Another research direction is to build a more powerful model for patterns to capture the positions from right to left and may be combine left-to-right and right-to-left positions in each depth of the pattern structure.