Graphical Numerical Inference

a.k.a Brain Surgery for Excel

Author: Jerene Zhe Yang
Carnegie Mellon University
Pittsburgh, PA
U.S.A.
yangzhe@gmail.com

Advisor: Professor Manuel Blum
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, U.S.A.
mblum@cs.cmu.edu

***Abstract*—Excel's drag and auto-fill feature works for most simple numerical cases like addition. However, it fails when someone gives it a checkerboard pattern with 1s and 0s and tries to extend the pattern. Excel is unable to expand this obvious pattern because its entire inference is based on a static snapshot of the final data. Graphical Inference Program (GIP), on the other hand, takes a dynamic approach by monitoring how the sequence is being filled. It will then try to figure out how the user is filling up the entries. After that, it picks up where the user has left off and fills in the rest of the entries.**

I. Introduction

The drag and expand feature in Excel is trying to look for a trend in what the user is doing and fill the empty grids according to this trend. In essence, if the user has a predictable pattern, the program is trying to guess it. However, this feature in Excel has several major limitations making it useless except for doing the simplest of tasks. This paper tackles most of these limitations by looking at the data given from a dynamic point of view rather than a static one. The program that we have created monitors the way the user inputs the data in order to more accurately guess what the user's intention is. Excel's drag and expand feature only allows expansion in one direction (horizontal or vertical) but never both. This is because Excel assumes that the data is entered in a column-wise manner or a row-wise manner. However, this assumption may not hold for some data. Our program, the Graphical Inference Program (GIP) does not make such assumptions and is hence able to have more accurate predictions. We hope that this program can offer a new perspective of how a spreadsheet program can be. Specifically, it is one that takes on a more proactive role when it comes to data extrapolation.

Numerical inference is not a solved problem. To properly describe the problem that we are trying to solve, we need to first define what it means to for a pattern to be inferable. We classify a sequence as inferable if and only if there exists a state machine with finite states such that if I run the state machine (possibly for infinitely many steps), then the sequence that we produce is the sequence that the user has in mind. There are many pre-existing methods for inferring patterns. However, there is no set of methods that can infer all possible inferable sequences. Therefore, there are websites like the Online Encyclopedia of Integer Sequences (OEIS) which is a huge database of integer sequences.

II. Examples

A. Checkerboard Pattern

Let's start off with a simple example: The user wants to enter a checkerboard pattern into excel that consists of 1s and 0s, in the following manner:

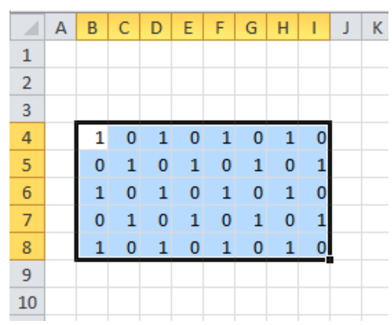
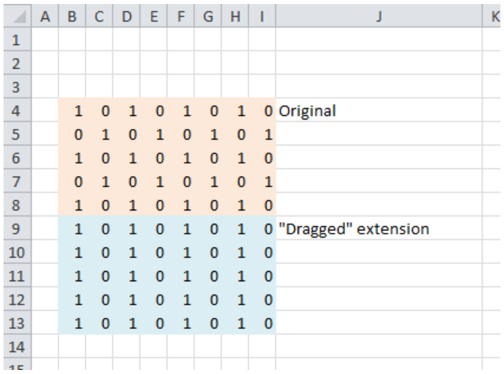


Figure 1

If you fill up a checkerboard pattern in Excel as above (Figure 1), and you try to drag the borders of this selection in attempts to expand the checkerboard, this is what you will get:



	A	B	C	D	E	F	G	H	I	J
1										
2										
3										
4		1	0	1	0	1	0	1	0	Original
5		0	1	0	1	0	1	0	1	
6		1	0	1	0	1	0	1	0	
7		0	1	0	1	0	1	0	1	
8		1	0	1	0	1	0	1	0	
9		1	0	1	0	1	0	1	0	"Dragged" extension
10		1	0	1	0	1	0	1	0	
11		1	0	1	0	1	0	1	0	
12		1	0	1	0	1	0	1	0	
13		1	0	1	0	1	0	1	0	
14										

Figure 2

Instead of continuing the checkerboard pattern that the user wants, Excel merely duplicates the last line 5 times. Furthermore, the data was entered in a row-wise fashion.

B. *Diagonal Inputs*

Besides predicting the pattern a user has in mind, GIP can also help users solve mathematical problems like the following:

What is the closed form for Grid(m,n) in the table on the next page?

We can input the integer sequence into a program like Wolfram|Alpha to get the closed form for each row or column. However, for row 3 and below, we do not have enough data to feed into Wolfram|Alpha to get anything useful. This is where GIP comes in.

	n=1	**n=2**	**n=3**	**n=4**	**n=5**
m=1	**1**	**2**	**4**	**7**	**11**
m=2	**3**	**5**	**8**	**12**	
m=3	**6**	**9**	**13**		
m=4	**10**	**14**			
m=5	**15**				

Figure 3

This table will gradually be filled in a certain order and the program will observe and try to continue the trend by filling up the rest of the table. Once the entire table is filled up / we have enough data, we can now search for the closed forms for each of the rows individually try to find a formulae for Grid(m, n).

III. Methodology

There are two main stages in GIP: Deciphering the input and generating the output.

A. *Deciphering the Input*

Input can be grouped into two categories depending on whether the input makes sense if we split up the rows and columns to consider them individually. For example, in the example of the n by 8 checkerboard, let us represent each 1 with a click and each 0 without a click. And we will fill in the grid in a row by row order. So the first row will be filled left to right with each alternating grid clicked, and then the second, and then the third, and so on. If we consider the row-coordinates of the grids that are clicked, we will see: 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4,… Now if we consider the column-coordinates, we will see: 1, 3, 5, 7, 2, 4, 6, 8, 1, 3, 5, 7, 2, 4, 6, 8, … Notice how the two sequences have predictable patterns independent of the other. This case falls into the first category – Analyze row and column independent of each other.

However, consider a spiral that looks like the following:

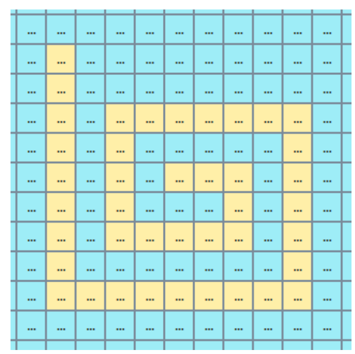


Figure 4

If we look at the row coordinates, we get this: 9, 9, 9, 10, 11, 11, 11, 11, 11, 10, 9, 8, 7, 7, 7, 7, 7, 7, 7, 8, 9, 10, 11, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 12, 11, 10, 9, 8, 7, 6, 5. This pattern, compared to the previous, is not as clear. However, if we look at how the coordinates are moving as a 2-tuple, the motion is a lot more obvious. R, R, D, D, L, L, L, L, U, U, U, U, R, R, R, R, R, R, D, D, D, D, D, D, … This falls into the second category – Analyze displacements as 2-tuples.

B. *Generating the Output*

a) Analyze row and column independent of each other

For the first category of sequences, we can only look at the row coordinates and column coordinates separately. We will then pass the row (or column) coordinates through a database of patterns which will then determine if the input is an instance of any of the classes of patterns in the database. Since each class of pattern in the database is essentially a state machine waiting for the input to fill in the constants. If the pattern does not match the state machine, then that particular state machine will return false for the given input. Each of the state machines in the database will individually assess whether the pattern matches its template. For example, the following is an example of a state machine that accepts input which accepts sequences $i - p = d$, where i is the input that is currently being red, p is the previous input, and c is some natural number. The starting state, a_0, must be a natural number too.

The example on the next page is a template for the state machine that is described above. There is one variable stored in every state except the fail state, which is denoted by "F". The program is initiallized with the variables set to null. When the first input is read, a0 will be updated and d will be updated once. From then on, only p will be updated. If we have to ever update d, we will end up in the "F" state and the machine will terminate. If we read until the end of the input and never get to the "F" state, then the input given matches the given machine. This machine is then run forever to produce the continuation for the given input.

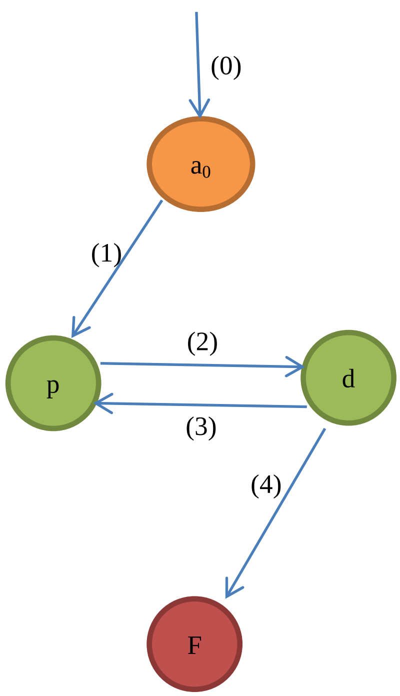


Figure 5

(0) Upon reading the first input, we update a_0 with the first number. For the example given, a_0 will be set to 3. Set i, the variable that keeps track of the last input, to a_0.
(1) We will move to the "*p*" state which stands for print. We will set the value at p to a_0.
(p) At state p, we will set $p := i$. Then we will print out the value of p.
(2) We will then read the next input, i, and move to d.
(d) If the variable at d is unset, it will be set to $i - a0$. Else, we will check whether $d == i - p$. If i is null (we have reached the end of the input), set $i := p+d$, keeping the value of d intact.
(3) If $d == i-p$, then we will move back to state p.
(4) If $d != i-p$, then we will move to the "*F*" state.
(F) Do nothing.
To generate the continuation, all we have to do is continue running the state machine with null input.

There are 11 state machines in total in GIP. The following are the sequences that they can identify.
State Machine 1:

- Recognizes sequences where consecutive terms differ by a fixed amount.
- Examples
  - 1, 2, 3, 4, 5, 6, ...
  - 3, 6, 9, 12, 15, 18, …
  - 19, 17, 15, 13, 11, …

State Machine 2:

- Recognizes sequences of repeated intervals.
- Examples
  - 2, 6, 3, 1, 4, 2, 6, 3, 1, 4, …

State Machine 3:

- Recognizes blocks of numbers of fixed lengths.
- Examples
  - 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4, …
  - 4, 4, 4, 4, 7, 7, 7, 7, 10, 10, 10, 10, 13, 13, 13, 13, …

State Machine 4:

- Recognizes patterns of increasing / decreasing intervals
- Examples
  - 3, 3, 5, 3, 5, 7, 3, 5, 7, 9, …
  - 3, 5, 3, 7, 5, 3, 9, 7, 5, 3, …
  - 12, 10, 8, 6, 4, 2, 10, 8, 6, 4, 2, …
  - 2, 4, 6, 8, 10, 12, 2, 4, 6, 8, 10, 2, 4, 6, 8, …

State Machine 5:

- Recognizes blocks of same numbers repeated for increasing / decreasing intervals
- Examples
  - 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, …
  - 8, 8, 8, 8, 8, 8, 7, 7, 7, 7, 7, 6, 6, 6, 6, 5, …

State Machine 6:

- Recognizes increasing / decreasing snake-like patterns
- Examples
  - 1, 2, 2, 1, 1, 2, 3, 3, 2, 1, 1, 2, 3, 4, 4, 3, 2, 1, …
  - 1, 2, 3, 4, 5, 6, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 5, 4, 3, 2, 1, …

State Machine 7:
- Checks for interleaved sequences where if we extract the individual sequences, these individual sequences will fit into one of the state machines 1 to 6.
- Examples
  - 1, 1, 3, 1, 5, 1, 7, 1, 9, 1, … [odds interleaved with ones]

State Machine 8
- Recognizes pattern that appear in "squares"
- Examples
  - 1, 2, 3, 2, 3, 4, 5, 6, 4, 5, 6, 4, 5, 6, 7, 8, 9, 10, 7, 8, 9, 10, 7, 8, 9, 10, 7, 8, 9, 10, …

State Machine 9
- Recognizes the pattern where the first number is being repeated once, the next two numbers repeated twice, the next three numbers repeated thrice, etc.
- Examples
  - 1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9, 10, 10, 10, 10, …

State Machine 10
- This is technically not a state machine. It searches for this sequence in the online encyclopedia of integer sequences for a hit and a continuation.
- Examples
  - 1, 1, 2, 3, 5, 8, 13, 21, …
  - 2, 3, 5, 7, 11, 13, 17, 19, …

State Machine 11
- This state machine caters to noise at the start of a sequence and then runs state machines 1-10 on the de-noised version.
- Examples
  - 120, 284, 1, 2, 3, 4, 5, 6, …

b) Analyze displacement from previous entry

1. We will first transform the sequence of inputs from coordinates to displacements from the previous point. In the example of the spiral that is shown above, we will get . R, R, D, D, L, L, L, L, U, U, U, U, R, R, R, R, R, R, D, D, D, D, D, D, … where R = (1, 0), L = (-1, 0), U = (0, 1) and D = (0, -1).
2. We will then map each of these displacements to a natural number. Hence, the sequence will now become 1, 1, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, …
3. This sequence will then be passed through state machines 1-11 to find a continuation.
4. The continuation will be in the form of natural numbers of we will then maps these back to the displacement form.
5. Lastly, we will find out the last input point that we have and calculate the next points based on the current point and the displacement from the current point.

IV. References