Automatic Restoration and Management of Computational Notebooks

Satish Venkatesan

Thesis proposal for

Master of Science

in

Computer Science and Applications

Muhammad Ali Gulzar, Chair

Eli Tilevich

Na Meng

02/11/22

Blacksburg, Virginia

Keywords: Computational Notebooks, Dependency Analysis, Version Control

Copyright 2022, Satish Venkatesan

Automatic Restoration and Management of Computational Notebooks

Satish Venkatesan

(ABSTRACT)

Computational Notebook platforms are very commonly used by programmers and data scientists. However, due to the interactive development environment of notebooks, developers struggle to maintain effective code organization which has an adverse effect on their productivity. In this thesis, we research and develop techniques to help solve issues with code organization that developers face in an effort to improve productivity. Notebooks are often executed out of order which adversely effects their portability. To determine cell execution orders in computational notebooks, we develop a technique that determines the execution order for a given cell and if need be, attempt to rearrange the cells to match the intended execution order. With such a tool, users would not need to manually determine the execution orders themselves. In a user study with 9 participants, our approach on average saves users about 95% of the time required to determine execution orders manually. We also developed a technique to support insertion of cells in rows in addition to the standard column insertion to help better represent multiple contexts. In a user study with 9 participants, this technique on a scale of one to ten on average was judged as a 8.44 in terms of representing multiple contexts as opposed to standard view which was judged as 4.77.

Automatic Restoration and Management of Computational Notebooks

Satish Venkatesan

(GENERAL AUDIENCE ABSTRACT)

In the field of data science computational notebooks are a very commonly used tool. They allow users to create programs to perform computations and to display graphs, tables and other visualizations to supplement their analysis. Computational Notebooks have some limitations in the development environment which can make it difficult for users to organize their code. This can make it very difficult to read through and analyze the code to find or fix any errors which in turn can have a very negative effect on developer productivity. In this thesis, we research methods to improve the development environment and increase developer productivity. We achieve this by offering tools to the user that can help organize and cleanup their code making it easier to comprehend the code and make any necessary changes.

Dedication

To my parents, sister, and friends who have supported me in my academic endeavors throughout the pandemic.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Muhammad Ali Gulzar. In the last year and a half that I have been working with him, he has offered me very valuable advice and was instrumental in my educational and professional development. I would also like to thank the members of my committee Dr. Eli Tilevich and Dr. Na Meng.

Next, I would like to thank my family and friends. Developing a thesis and attending graduate school in the middle of a pandemic has offered its fair share of challenges. Their constant and unwavering support has been crucial and is greatly appreciated.

Lastly, I would like to express my gratitude towards the Graduate Computer Science Department for the opportunities and resources that have allowed me to extract maximum value from my higher education.

Contents

List of Figures viii

1 Introduction 1

1.1 Limitations . . . 2

1.2 Research Objectives . . . 4

1.3 Research Contributions . . . 5

2 Background on Computational Notebooks 7

2.1 Walkthrough . . . 8

3 Review of Literature 13

4 Automatic Order Restoration in Computational Notebooks 16

4.1 Introduction . . . 17

4.2 Background . . . 18

4.3 Approach . . . 23

4.4 Implementation . . . 23

4.5 Evaluation . . . 31

5 Code Management in Multi-Context Notebooks 38

5.1 Introduction 39

5.2 Background 40

5.3 Approach 44

5.4 Implementation 46

5.5 Evaluation 49

6 Conclusions **53**

Bibliography **55**

Appendices **59**

Appendix A User Study **60**

A.1 User Study Questions 60

List of Figures

2.1 Markdown Cell Before Execution 7

2.2 Markdown Cell After Execution 8

2.3 Jupyter Notebook File Header 9

2.4 Header Cell 9

2.5 Data Loading Cell 10

2.6 Visualization Cells 10

2.7 Teacher's Report 11

2.8 Edited Visualization Cell 12

2.9 Modified Output 12

4.1 Sample Notebook 26

4.2 Notebook after Running Plugin 27

4.3 Notebook after Clearing Borders 28

4.4 Notebook Before Running Plugin 28

4.5 Notebook After Running Plugin 29

4.6 Notebook with Overwritten Executions Paths 30

5.1 Teacher's Report 41

5.2 Teacher's Edited Report 42

5.3 Example Notebook with Three Parallel Cells. 45

5.4 Example Notebook in Matrix View 45

5.5 Teacher's Report with Parallel Cell Support 46

5.6 Notebook Before Parallelization Plugin is Run 47

5.7 Notebook After Parallelization Plugin is Run 48

Chapter 1

Introduction

Computational Notebooks are digital notebooks that can perform simple computing tasks. This allows the authors of computational notebooks to supplement their analysis and written content with graphs and other visual components leading to a more comprehensible computing process [19, 21, 25]. Common Computational Notebook platforms include Azure Notebooks, Databricks, and Wolfram Computational Notebooks. However, the most popular computational notebook platform is Jupyter. The popularity of Jupyter Notebook and computational notebooks is evidenced by the fact that 0.4% of all users on Github have shared a Jupyter Notebook[1]. Computational notebooks are widely used as they make it easier for users to write more detailed and organized reports. Notebook environments can also be more accessible to users as they contain a built-in modularity with the usage of cells. So, rather than planning out an entire algorithm, users can write pieces of code in different cells and as cells can be run individually, users can test each part as they develop which aids in exploratory programming. The user base of Computational Notebooks consists of many data scientists, computer scientists and analysts from a variety of concentrations. As it is the most prominent computational notebook platform, this paper and the research is focused on the functionality of Jupyter Notebook.

1.1 Limitations

While computational notebooks contain many useful features, there are certain limitations which negatively impact the productivity of users. These limitations are enumerated as follows.

Limitation 1: Limited Display Options Computational Notebooks have limited display capabilities, in that cells can only be shown vertically. While this is intuitive in cases where code execution occurs linearly and in a definite order, if there are multiple cell execution orders, or interchangeable cells, this display can be confusing for the user working on the notebook or new users who are viewing the notebook. The current method to deal with this problem is having multiple notebooks or having header cells or comments to annotate the flow of the code as it pertains to the cells. However, with already complex notebooks, commentating unused code, or containing multiple versions of cells can lead to exceedingly large notebooks with only a portion of it containing computational value.

Limitation 2: Cell Organization. Due to how cell-based execution is implemented in Computational Notebooks, users have to keep track of cell dependencies. To elaborate, users must know what order code cells have to be run in order to obtain the output they desire [20]. While some notebooks are simply ran in vertical order as they appear, in nearly half of computational notebooks shared on the platform Github, the cell execution order does not match the order as presented [13]. Right now, Computational Notebooks do not keep track of execution order, which means to get around this problem users have few options. One option is to keep a mental note of cell dependencies for each cell in the notebook. This may be practical for simple notebooks with few cells, however for more complex notebooks with many cells, this is very difficult and can cause the user to have to spend time analyzing

the cells to reconstruct the cell dependency which greatly decreases productivity. Another option, which is available in the Jupyter environment, is for users to use an extension from the common extension library, which uses manually entered cell-dependencies to run cells in the dependency tree before running the selected cell. While this is better than keeping mental notes or written annotations, it still requires users to manually write and update cell dependencies in the cell metadata. All of these current solutions place the onus completely on the developer, and therefore are highly error prone and also lead to a drop in productivity. Furthermore, solutions which depend heavily on the author can be difficult as data scientists often struggle to retrace the steps they performed in their data analysis [11].

Limitation 3: Programming Background of User Base The term data scientist refers to a group of people with a wide variety of educational backgrounds and skill-sets [13]. Many Computational Notebook users don't have extensive programming backgrounds. This means that they are less likely to be aware of let alone follow best practices and typical coding conventions. For example, they may give multiple variables the same name across different cells which can result in them getting different outputs for different executions. Moreover, without an extensive coding background, in the event of an error users may find it more difficult to debug their code especially when it is split amongst multiple cells [10].

Limitation 4: Notebook Code vs Traditional Code While Computational Notebooks support a multitude of languages, traditional and notebook code still operate rather differently, due to the usage of cells. Cell-based execution allows users to create small pieces of code and more easily test as they develop their computational processes. This makes it more simple for users to develop full solutions in notebook environments than in coding environments [12]. However, the more approachable environment can lead to more incomplete and poorly designed notebooks.

Limitation 5: Lack of version control Handling multiple versions of cells within notebooks and visualizing the change in both the code and the resulting output is very difficult to perform in computational notebook environments [4]. Version control is imperative to software engineering and even more critical to statistical analysis especially in regard to collaboration [15]. In traditional software engineering, users use distributed version control systems, such as Git to manage multiple versions of code between multiple users [27]. However, using Git for computational notebooks is not ideal for keeping track of incremental changes [17]. Git compares changes in the JSON files which encapsulates notebook metadata and cell metadata on top of the notebook contents, rather than just changes the user makes in code which makes it very difficult for users to understand the differences between versions [18]. The lack of an intuitive version control can lead to users maintaining numerous cells in the notebook to represent different versions, requiring the user to recognize and execute the latest one.

1.2 Research Objectives

To address limitation 1 and limitation 5, we take inspiration from branching and merging in traditional version control (VCS) and redesign the existing notebook layout to incorporate a matrix view which can better represent the distinct execution flow of cells within a notebook. Instead of just the column view of notebook, matrix view offers rows that allow users to switch between multiple alternate implementations, as sometimes reflected in VCS branches. The addition of side-by-side cell organization can help users organize their code in a more intuitive fashion if they need to visualize parallel cells, represent multiple versions, or handle multiple input cells. This improved code organization will help users to better extract the semantic meaning of their notebook, enabling them to debug the notebook faster and thereby improving productivity. This will also help new users, when familiarizing themselves with a

new notebook, understand the flow and purpose of the notebook.

To address limitation 2 and limitation 4, we design a static dependency analysis technique that finds any given cell's execution order, turning unusable shared notebooks usable. This feature will scan a notebook and using syntactic analysis, determine the execution order(s) of the designated cell, and automatically rearrange the cells. With this approach, we plan on reducing the amount of human input required and thereby reducing the error probability when running a notebook.

To address limitation 3 and limitation 4 we ensure the plugins that we develop only require a button press from the user to run, and returns output in a visually intuitive manner.

1.3 Research Contributions

The prime contributions of our research are enumerated as follows.

- We have added support for scrollable horizontal cells within the computational notebook environment. With the current notebook, a user can only add cells vertically. Adding support for horizontal cells allows users to more intuitively compare different versions of cells or execution threads.

- We have also ensured that this horizontal cell formation persists. By default, Computational Notebooks forces the cells into the standard vertical order, however, by using cell metadata, the horizontal arrangement can be persisted.

- We have designed a technique to derive the def-use sets of a notebook and retroactively recognize and arrange parallel cells in a horizontal order. This allows existing notebooks to also reap the benefits of being presented in a matrix view.

- We have also designed a technique that illustrates execution orders for a given cell within a notebook by highlighting the cells in the execution order. If there is only one possible execution order and the cell order differs from the execution order, the notebook will rearranged to match the execution order and improve the cell organization.

The rest of the thesis is outlined as follows. In Chapter 2 we give some background into Computational Notebooks by discussing the unique features they offer and walking through a basic use case scenario. In Chapter 3, we analyze similar papers in the field. In Chapter 4 we discuss how we implement automatic order restoration and identifying execution orders in computational notebooks. In Chapter 5, we discuss how we approach the problem of code management in multi context notebooks.

Chapter 2

Background on Computational Notebooks

Feature Discussion Computational Notebooks contain a sequence of cells, which are displayed in vertical order. Each cell operates as a mini component as it can be run or edited independently of other cells in the notebook. Jupyter, which we will use in our example, supports three types of cells: markdown, code, and raw cells. Markdown cells allow users to enter formatted text within the notebook. Markdown cells support text styling, such as italics, bold, underline and simple math formulas written in LaTeX notation. While markdown cells are only text, they are still run in the notebook as upon execution the styling and mathematical notations are evaluated and executed. In Figure 2.1 we can see a Markdown cell before execution, and in Figure 2.2 we can see the same cell after execution.

```
# The Effect of Driving Features on Driver Safety
   ## In this report we will analyze different features and their potential impact on driver safety
   **Driver Safety Features**
    1. Lane Assist
    2. Blind Spot Detection
    3. Automatic Cruise Control
   **Entertainment Features**
    1. Android Auto/Apple Play
    2. Navigation
```

Figure 2.1: Markdown Cell Before Execution

The Effect of Driving Features on Driver Safety

In this report we will analyze different features and their potential impact on driver safety

Driver Safety Features

1. Lane Assist
2. Blind Spot Detection
3. Automatic Cruise Control

Entertainment Features

1. Android Auto/Apple Play
2. Navigation

Figure 2.2: Markdown Cell After Execution

Code cells are executable cells, and the language code cells are written is dependent on the kernel, which by default is python. The content of code cells is considered to be the input. Upon execution, the output of the computation is displayed in an output cell directly beneath the original code cell. Computational Notebooks support a variety of outputs, such as text output, HTML tables and matplotlib figures. Lastly, raw cells are cells which allow users to directly enter output to be displayed. Raw cells are the only type of cells to not be executed by the notebook.

Cell-based execution and organization allow users to create independent code cells and edit these cells without changing the content or functionality of other cells. While the use of cells splits code into smaller pieces and makes development easier for the author, it also means that cells can be run in any order, and as more cells are added, it can be difficult to keep track of the correct execution order.

2.1 Walkthrough

To show the basics of computational notebooks, and their various features, let us walk through a basic use case, where a teacher is conducting an experiment with their class to

determine the effect of study hours on exam scores and wants to display the results. In the image below is the header of a Jupyter notebook file. The header contains a toolbar, which allows the user to perform certain functionalities such as: adding cells, deleting cells, moving cells, running cells etc.

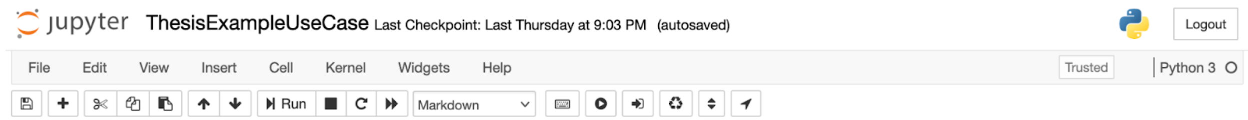


Figure 2.3: Jupyter Notebook File Header

The teacher starts by creating a header cell, as shown below, to title her report. This makes use of the 'markdown cell' feature in Jupyter Notebook.

The Effect of Time Spent Studying on Exam Scores

Data was gathered in a poll from students asking how long they studied before taking an exam ¶

Figure 2.4: Header Cell

After writing a header cell for their report, the teacher wants to load data. The notebook, through the use of the pandas library, allows users to import data from local files, like csv files, from urls or from zip files. In this case, the teacher's data resides in an excel document and she writes code to extract the information. The teacher adds a cell to include all their imports, and a cell to import data, as shown below.

```
In [1]: import pandas as pd
        import collections
        import statistics
        import matplotlib.pyplot as plt

In [2]: #Imports Data from an Excel Sheet
        timeGradeData = pd.read_excel (r'TimeVsGrade.xlsx', sheet_name='Sheet1')
```

Figure 2.5: Data Loading Cell

The teacher decides that a frequency graph of the number of hours studied and a line graph comparing hours studied with average exam score is the best method to show students the effect study time might have on their grade.

The teacher adds and creates cells to perform the computation for each of these visualizations in python code cells. Afterwards, the teacher takes advantage of matplotlib library and the Notebook's ability to visualize output from code cells to create a frequency graph and a line graph to display her results. The visualization cells:

```
In [ ]: #Frequency Plot Code
        freq = collections.Counter(list(timeGradeData['Hours Spent Studying']))
        fig = plt.figure()
        axes = fig.add_axes([0,0,1,1])
        axes.bar(list(freq.keys()), list(freq.values()))
        plt.xlabel("Number of Hours Studied")
        plt.ylabel("Number of Students")
        plt.title("Freqeuncy Plot of Study Hours and Number of Students")
        plt.show()

In [ ]: #Line Graph Code
        #The teacher decides to take the mean of each group and plot it
        xvalues = list(set(timeGradeData['Hours Spent Studying']))
        yvalues = []
        for val in xvalues:
            hour_df = timeGradeData.loc[timeGradeData['Hours Spent Studying'] == val]
            grade_lst = list(hour_df['Grade'])
            yvalues.append(statistics.mean(grade_lst))
        hoursVsScore = {'Hours Studied': xvalues, 'Average Exam Score': yvalues}
        hoursScoreDf = pd.DataFrame(data=hoursVsScore)
        print(hoursScoreDf)
        plt.plot(xvalues,yvalues,'--bo')
        plt.xlabel("Hours Studied")
        plt.ylabel("Average Exam Score")
        plt.title("Hours Studied vs Average Exam Score")
        plt.show()
```

Figure 2.6: Visualization Cells

After executing these cells, the teacher's report looks like this:

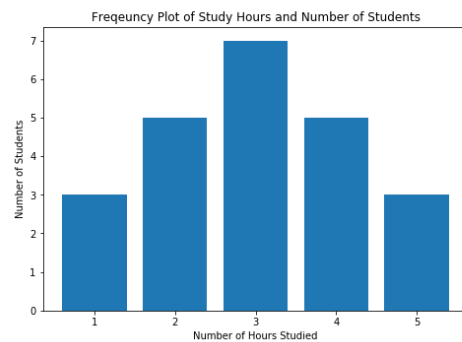
The Effect of Time Spent Studying on Exam Scores

Data was gathered in a poll from students asking how long they studied before taking an exam

In [1]:
```
import pandas as pd
import collections
import statistics
import matplotlib.pyplot as plt
```

In [2]:
```
#Imports Data from an Excel Sheet
timeGradeData = pd.read_excel (r'TimeVsGrade.xlsx', sheet_name='Sheet1')
```

In [3]:
```
#Frequency Plot Code
freq = collections.Counter(list(timeGradeData['Hours Spent Studying']))
fig = plt.figure()
axes = fig.add_axes([0,0,1,1])
axes.bar(list(freq.keys()), list(freq.values()))
plt.xlabel("Number of Hours Studied")
plt.ylabel("Number of Students")
plt.title("Freqeuncy Plot of Study Hours and Number of Students")
plt.show()
```



In [4]:
```
#Line Graph Code
#The teacher decides to take the mean of each group and plot it
xvalues = list(set(timeGradeData['Hours Spent Studying']))
yvalues = []
for val in xvalues:
    hour_df = timeGradeData.loc[timeGradeData['Hours Spent Studying'] == val]
    grade_lst = list(hour_df['Grade'])
    yvalues.append(statistics.mean(grade_lst))
hoursVsScore = {'Hours Studied': xvalues, 'Mean Exam Score': yvalues}
hoursScoreDf = pd.DataFrame(data=hoursVsScore)
print(hoursScoreDf)
```

```
   Hours Studied  Mean Exam Score
0              1        51.953333
1              2        58.150000
2              3        70.642857
3              4        83.334000
4              5        94.760000
```

In [5]:
```
plt.plot(xvalues,yvalues,'--bo')
plt.xlabel("Hours Studied")
plt.ylabel("Exam Score")
plt.title("Hours Studied vs Mean Exam Score")
plt.show()
```

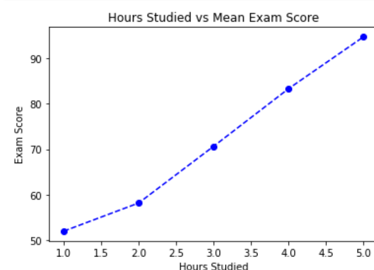


Figure 2.7: Teacher's Report

After presenting this report to their class, some students remained skeptical over the teacher's use of mean scores as a metric, stating that outliers could affect the results heavily and therefore the study isn't reliable. The students then state it would be better to use median instead of mean. Thanks to cell-based execution, the teacher only has to edit the fourth cell, which performs the computations and displays the line plot. The teacher edits the cell to this:

```
In [5]: #Line Graph Code
        #The teacher decides to take the mean of each group and plot it
        xvalues = list(set(timeGradeData['Hours Spent Studying']))
        yvalues = []
        for val in xvalues:
            hour_df = timeGradeData.loc[timeGradeData['Hours Spent Studying'] == val]
            grade_lst = list(hour_df['Grade'])
            yvalues.append(statistics.median(grade_lst))
        hoursVsScore = {'Hours Studied': xvalues, 'Median Exam Score': yvalues}
        hoursScoreDf = pd.DataFrame(data=hoursVsScore)
        print(hoursScoreDf)
        plt.plot(xvalues,yvalues,'--bo')
        plt.xlabel("Hours Studied")
        plt.ylabel("Median Exam Score")
        plt.title("Hours Studied vs Median Exam Score")
        plt.show()
```

Figure 2.8: Edited Visualization Cell

And the following output is shown below:

```
   Hours Studied  Median Exam Score
0              1              50.87
1              2              53.02
2              3              71.21
3              4              86.70
4              5              94.51
```

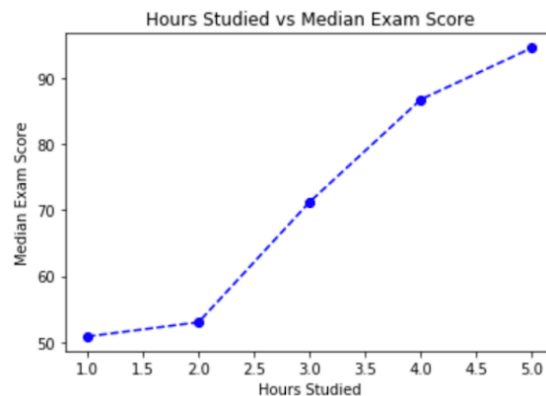


Figure 2.9: Modified Output

In this basic scenario, we can see how the teacher uses the different cell types, visualizations, and cell-based execution to develop a comprehensive report.

Chapter 3

Review of Literature

Def-Use Chain Analysis Kennedy asserts that use-definition chains, which we refer to as def-use chains, can be used to optimize programs by eliminating useless computation [8]. Kennedy proposes two different algorithms to delete useless computation through use-definition chain analysis. While we do not plan to delete any code in our research, identifying 'useless' or redundant cells could enable us to improve the cell organization of a notebook.

Subotić et. al developed a static analysis framework for data science notebooks, NBLyzer, which can aid users by offering code impact analysis as well as Data Leakage Analysis [22]. Subotić et. al utilize def-use chain analysis to determine inter-cell and intra-cell data relationships which is then used to perform the analysis their framework offers. The paper outlines how they develop the def-use sets and determine variable relationships between cells which could be valuable in our approach.

Version Control Visualization German et. Al propose a token-level blame system which when tested against is capable of identifying the commit that is responsible for the error between 94.5% and 99.2%, compared to the standard line-based blame system which performs between 75% and 91% [6]. German et. Al assert that with a token-based blame system, more than just the direct previous commit is analyzed, and therefore, can more accurately analyze what code change led to an error. From this paper we can grasp that

when displaying version control, simply supporting a dual view of the latest commit and current commit does not suffice. Specifically, with computational notebooks, it is beneficial to be able to support displaying more than two versions of a cell.

Kery et. al develop a lightweight local version control system, Variolite [9]. Variolite allows users to select blocks of code that they would like to commit locally. On top of the code, Variolite keeps track of the outputs and allows users to switch between commits based on the output. In a user study of Computational Notebook users, nine out of ten participants stated that they found the tool easy to use with all ten stating they would use it outside the study. Variolite exhibits that unlike version control in traditional programming contexts, maintaining code differences does not suffice for computational notebooks. It is critical to keep track of outputs and cell metadata as well.

Computational Notebook Analysis and Survey Timov et. al propose an algorithm ReSplit, which analyzes notebooks and either merges or splits cells in order to improve their cell organization [23]. The evaluation showed that in 29.5% of cases people preferred the restructured notebooks as a result of ReSplit. The results of ReSplit are very valuable to us, as they convey that when attempting to improve code organization within notebooks, rearranging cell contents may not be the preferred approach.

Kery et. al in a survey of data scientists, found that 76% prioritized finding a solution over high-quality code [9]. In the survey, they also discovered that many users keep obsolete code and sometimes comment it out as they do not want to lose that work and believe that they may need to use this code in future computations. This is valuable to our research as it gives us an insight into the mentality and practices of computational notebook users towards code organization. By developing tools to aid them with code organization, users can have clean notebooks without compromising their efforts toward finding a solution.

Head et. al propose a variety of extensions and tools to help users find, clean up and compare and recover different versions of code in messy notebooks [7]. In their user study of 12 computational notebook users, they found that users valued the post hoc management of messes in their approach. Users liked that they were not required to put in effort to manage the notebook up front. This insight is critical and can guide our approach towards improving code organization.

Chapter 4

Automatic Order Restoration in Computational Notebooks

One significant challenge when dealing with computational notebooks is keeping track of cell execution orders which are usually lost over the long code development and execution session of a notebook. The history of cell execution is critical to reproduce the results from a notebook, especially when such notebooks are shared among developers and expected to be reused. This chapter addresses the irreproducibility of existing notebooks by exploring the following research questions. *How can we determine the correct execution order of otherwise unordered cells in a computational notebook?*. Our hypothesis is that *we can leverage the static data dependencies of each cell of a given notebook to determine a set of statically correct cell execution orders and then leverage the outputs stored in markdown cells to eliminate the semantically incorrect orders.* In this chapter, we realize our hypothesis in a real-world technique that automatically extracts the data dependencies of each cell of a notebook and uses the stored output to restore a notebook's cell execution order producing the expect output.

4.1 Introduction

Notebooks can very easily have execution orders that do not match the linear order in which they are shown and force users to follow painful debugging procedures to recover the execution orders in their notebooks [20]. In a survey conducted with 156 data scientist practitioners, it was found that 90% of participants found exploring code and cell history to be important and that 60% of participants believed this to be a difficult procedure [4]. Computational notebooks without cell execution orders are considered broken and are unlikely to be reused. Almost every user of computational notebooks is going to face the problem of deriving the execution order of cells in their notebook, and a significant portion will find solving this problem to be difficult.

This problem space poses a couple primary challenges in the way of deriving cell execution orders due to the unique coding environment of computational notebooks. One challenge occurs when a cell has multiple cells it's dependent upon. If a selected cell is dependent on n cells, there are n! possible execution orders that end with the selected cell. With complicated notebooks that represent the average coding environment of our user base, this can result in numerous execution orders. Consequently, showing users all execution orders would be minimally helpful. Even if this search space is reduced, the user will have to capture the semantics of the program which is traditionally performed by a suite of test cases. However, notebooks rarely contain test cases.

To address the common pain point of cell organization in computational notebooks, we design a technique that restores the correct execution order of notebook cells for a given output, making unusable notebooks operational. We leverage existing static analysis techniques to generate the abstract syntax tree of code in each cell and construct a set of variables and functions defined and used, referred to as def-use sets. Our insight is that, to run the application

in accordance with its semantic purpose, the code must follow the syntactic data dependencies, represented by the topological sort of cells based on their def-use sets. However, execution orders derived from syntactic data dependencies may also contain semantically incorrect orders. We observe that notebooks often contain the output of an execution in the output markdown cells. To eliminate such execution orders, our insight is that such output cells can be translated into test cases and help us choose the execution order that complies with the expected semantics of the program. This approach can be instrumental in making a large set of shared public notebooks on version control systems (e.g., GitHub) usable. We realize our technique in Jupyter Notebook plugin in which a user can automatically restore an unordered notebook with just a click of a button.

We evaluate our approach on three primary criteria: ease of use, and time savings. In a user study with 9 human subjects, we find that our approach saves, on average, between 4 and 5 minutes compared to manually reordering the notebooks. Users also rated the difficulty of determining the cell dependency of a given notebook manually as a 5.1. However, with the plugin the difficult of the task was rated on average as a 1.44.

The rest of chapter is structured as follows. Section 4.2 provides background information into the problem space and also demonstrates the problem with the help of a motivating example. Section 4.3 presents the approach used to address the problem and challenges while Section 4.4 goes into the details of the implementation of our approach with some discussion as to future steps. Lastly, Section 4.5 evaluates the approach through a user study.

4.2 Background

Cell Dependencies In Computational Notebooks, each cell can be written, edited, and run independently. However, that does not mean that a cell will compile on its own success-

fully. We will use the image below as an example.

```
In [ ]: i = 15
        j = 12
        k = 9

In [ ]: l = i + j
        print(l)
```

In this image, we have two cells. In the first cells, variables i, j, and k are declared, while in the second cell, variable l is declared and printed. It is important to recognize though, that in the second cell, both variables i and j are referenced without being declared. If a user tried to run the second cell without running the first cell, they would receive an error as shown below.

```
In [ ]: i = 15
        j = 12
        k = 9

In [1]: l = i + j
        print(l)

        ---------------------------------------------------------------------------
        NameError                                 Traceback (most recent call last)
        <ipython-input-1-7a8ed18c68a5> in <module>
        ----> 1 l = i + j
              2 print(l)

        NameError: name 'i' is not defined
```

In the error dialogue, it shows that a 'NameError' was thrown with the descriptive message stating that name 'i' is not defined. As stated earlier, in the second cell references variables i and j, however they are not declared. So, when the second cell is run on its own, it does not know what value variable i or for that matter j are referring to. In the first cell, however, both variables i and j are declared.

```
In [1]: i = 15
        j = 12
        k = 9

In [2]: l = i + j
        print(l)

        27
```

if the first cell is run before second, both cells run successfully, and the value of l is ac-

curately printed. In this situation, we could say that the second cell is dependent on the first cell, because for the second cell to run without an error, the first cell must be run previously. Computational Notebooks does not keep track of these cell dependencies, so the user must have a mental note of the cell dependencies. In the example above, this is rather straightforward.

Motivating Example For an assignment given by his teacher, Charles has to write a notebook where he calculates the standard deviation for a set of data. The teacher instructs the class that he will enter in the data manually in the first cell, and also provides one instance of test data for the students to use.

So, Charles creates a notebook to determine the standard deviation of this test data and runs his code. The notebook and its output are shown below.

```
In [1]: import math
        data = [24,33,72,19,24,77,82,95,100,67]

In [2]: dataSum = 0
        for val in data:
            dataSum += val
        mean = dataSum/len(data)

In [3]: dataDiffs = []
        for val in data:
            dataDiffs.append(val-mean)

In [4]: stdDev = 0
        for val in dataDiffs:
            stdDev += val
        stdDev /= len(dataDiffs)
        stdDev = math.sqrt(stdDev)

In [5]: print("The Standard Deviation of this Data is: ", stdDev)

        The Standard Deviation of this Data is:  5.331201499700045e-08
```

Charles sees the output and realizes that something is wrong with the code as the standard deviation value he received doesn't match the value he received from his calculator.

Charles then realizes that he forgot to square the difference of the distance between each value and the mean before summing, so he quickly adds a cell just before computing standard

deviation and runs that cell and the cell above to recompute and then the output cell to display the new standard deviation. The new notebook with the correct output is shown below.

```
In [1]: import math
        data = [24,33,72,19,24,77,82,95,100,67]

In [2]: dataSum = 0
        for val in data:
            dataSum += val
        mean = dataSum/len(data)

In [3]: dataDiffs = []
        for val in data:
            dataDiffs.append(val-mean)

In [7]: stdDev = 0
        for val in dataDiffs:
            stdDev += val
        stdDev /= len(dataDiffs)
        stdDev = math.sqrt(stdDev)

In [6]: for i in range(len(dataDiffs)):
            dataDiffs[i] *= dataDiffs[i]

In [8]: print("The Standard Deviation of this Data is: ", stdDev)

        The Standard Deviation of this Data is:  29.644729717101487
```

Charles sends this notebook to his teacher, and his teacher sees that for the test sample Charles has the right answer. The teacher then edits the first cell with her own test values and runs the cells in order and receives the following output.

```
In [1]: import math
        data = [97, 43, 75, 69, 30, 26, 91, 60, 88, 17]

In [2]: dataSum = 0
        for val in data:
            dataSum += val
        mean = dataSum/len(data)

In [3]: dataDiffs = []
        for val in data:
            dataDiffs.append(val-mean)

In [4]: stdDev = 0
        for val in dataDiffs:
            stdDev += val
        stdDev /= len(dataDiffs)
        stdDev = math.sqrt(stdDev)

        ---------------------------------------------------------------------------
        ValueError                                Traceback (most recent call last)
        <ipython-input-4-f0470cfe05ff> in <module>
              3     stdDev += val
              4 stdDev /= len(dataDiffs)
        ----> 5 stdDev = math.sqrt(stdDev)

        ValueError: math domain error

In [5]: for i in range(len(dataDiffs)):
            dataDiffs[i] *= dataDiffs[i]

In [6]: print("The Standard Deviation of this Data is: ", stdDev)

        The Standard Deviation of this Data is:  -1.4210854715202005e-15
```

In the output the teacher sees an error in the code output. Looking at this error, Charles analyzes the code in each cell and how it translates to the steps of calculating standard deviation. He realizes the teacher ran the notebook in the incorrect order.

Problem Statement As stated in limitation 2, due to the use of cell-based execution, Notebook requires cells to be run manually and in the same order to achieve the same results. The default is running all the cells in sequential order from top to bottom. However, this requires users to either memorize the cell execution orders, or manually derive the cell execution order themselves, both options being error prone and requiring excess effort from the user. Furthermore, as stated in limitation 3, the programming background of a significant portion of computational notebook users is not very extensive, making it even more difficult to derive the execution orders on their own. Aiding this pain point could dramatically improve developer productivity and code organization.

4.3 Approach

To address the challenge of code organization, we derived certain insights to inform the development of our technique.

Our first insight was that by deriving the syntactically valid candidates of a given notebook, we could greatly reduce user workload and eliminate the necessity for the user to evaluate each cell and derive variable and cell dependencies.

Another insight was that computational notebooks by design have a form of modularity presented in the form of cells. This greatly reduces our search space and simplifies the problem as we only need to analyze cell dependencies and execution orders.

Our last insight was that rearranging the cells to better reflect the semantic meaning of a program can help users more easily understand the notebook and follow the cell layout.

Using these insights, we will develop an approach that, for a given cell, shows the minimal set of potential execution orders greatly reducing the execution orders the user has to analyze. If there is only one possible execution order, the notebook will also be rearranged if necessary to better match the cell execution order.

4.4 Implementation

Automation When developing a technique to address the problem of code organization, one of the primary requirements we set was to automate as much of the process as possible. We recognized that the current methods used to deal with the issue required manual effort from the user, which could heavily impact their productivity. Also, human error is inescapable, meaning the likelihood of these manual solutions failing is rather significant.

Upon this failure, users would then have to analyze each code cell and manually derive the cell dependencies on their own, which is tedious and time consuming. Another issue with manual solutions is that, as stated in Limitation 3, there is a significant portion of the user base with minimal coding experience, therefore reducing their ability to accurately derive the cell dependencies on their own.

Determining Cell Dependencies The task at the center of creating an automated solution to cell organization was determining cell dependencies when only given the contents of the cells. To understand how to approach this problem, we set out to understand how users, when left with no other option, manually derived the cell execution order. The process would go as follows:

The user would start with the cell they set to determine execution order for, we will call this the target cell. They would examine the variables that were used in the target cell and identify any variables that were not declared within that cell, we will refer to these as input variables. If there are any such variables, then the user would know that the cell was dependent on another cell or set of cells to declare the input variables. Then the user would analyze the cells preceding the target cell and keep track of what variables those cells initialized, we will refer to these variables as output variables. The user would then identify the cells whose output variables contained any input variables of the target cell, and then test different combinations of these cells to determine the execution order.

After identifying this process, we noticed that the idea of determining input and output variables can benefit from def-use chain analysis using static code analysis. Leveraging the use of Abstract Syntax Trees which organize sections of code into a hierarchical tree structure based on the data types and operations performed in the source code, we could identify the variables in each cell and determine whether they were input or output variables [16, 26].

Identifying Dependent Cells With the knowledge of a cell's input variables and the preceding cell's output variables, identifying any given cell's dependent cells became as simple as selecting cells whose set of output variables contained any of the given cell's input variables. However, because of the use of cell based execution, the set of dependent cells could be run in any order. Meaning that the number of potential executions is the factorial of number of dependent cells. To maximize the value of our plugin, it was imperative to narrow down the execution orders from the set of potential execution orders. What we realized was that most potential execution orders would not actually compile. So, by analyzing the execution orders and removing ones that wouldn't compile, we would be greatly reducing the number of potential execution orders. To relay this info back to users, we developed a plugin, which highlights the cells that belong in the execution order of a selected cell. This visual representation makes a clear distinction between relevant and irrelevant cells and allows the user to analyze the execution order if they would like.

Algorithm 1 Compiling List of Dependent Cells

```
Require: n-selected cell number, defUseSets- set of def-use sets for all cells
  procedure getCellDep(n)
      cellDep ← []
      for variable in defUseSets[n]["use"] do
          for j in range(1,n) do
              if variable in defUseSets[j]["definition"] then
                  for item in getCellDep(j) do
                      if item not in cellDep then
                          cellDep.add(item)
                  cellDep.add(j)
      return cellDep
```

Here is an example of the solution in action. In this scenario, we have a simple notebook with 5 cells which is shown in Figure 4.1.

Algorithm 2 Validating Cell Execution Orders

Require: executionOrderSet - all permutations of dependent cells, defUseSets- set of def-use sets for all cells

procedure VALIDATECELLS($executionOrderSet$)
  for `executionOrder in executionOrderSet` **do**
    $declaredVariables \leftarrow []$
    for `cell in executionOrder` **do**
      if $defUseSets[cell]["use"] - declaredVariables \neq []$ **then**
        Remove execution order and iterate to next order
      declaredVariables.push(defUseSets[cell]["definition"])
  return executionOrderSet

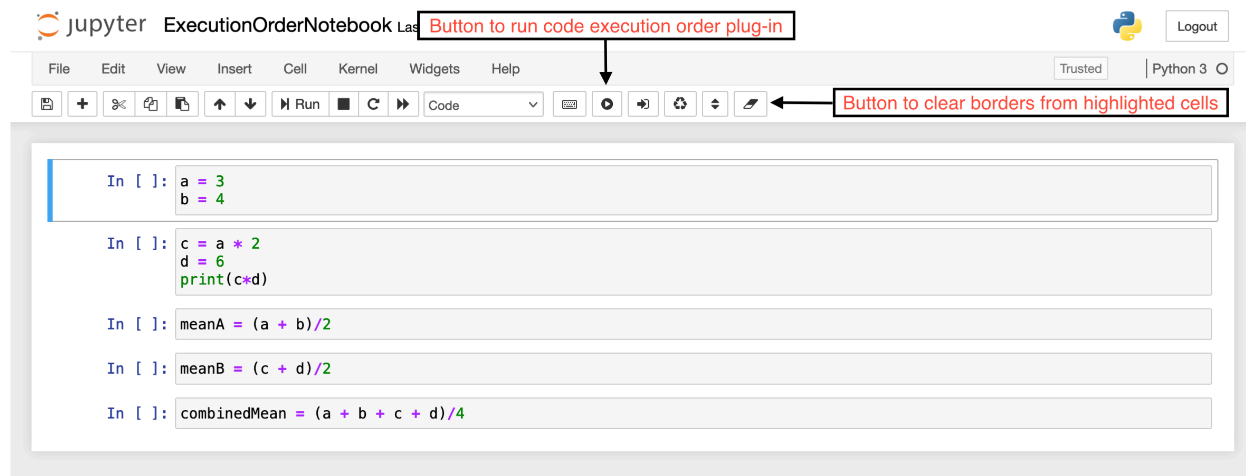


Figure 4.1: Sample Notebook

The user selects cell 5, clicks the button to run the plug-in. The first step of the algorithm is analyzing the cells in the notebooks and generating the def-use sets for each cell. Then next step is analyzing the def-use set of the selected cell, which is cell 5. In cell 5 only one variable is defined which is $combinedMean$. Cell 5's usage set contains variables a, b, c and d. Next, the algorithm analyzes the def-use sets of preceding cells and looks for any cells that contain any members of Cell 5's usage set in their definition sets. In this case, Cell 1 contains variables a and b in its definition set, while Cell 2 contains variables c and d in its definition set. Then the algorithm performs the same procedure it has just performed with Cell 5, with both Cell 1 and Cell 2 to add any of their dependent cells. In this case, Cell 1

is not dependent on any other cell, and Cell 2 is dependent on Cell 1 which is already in the list. Then the algorithm tests every permutation of the dependent cells, which is Cell 1 and Cell 2, and eliminates all syntactically incorrect orders. This leaves only on execution order of Cell 1, Cell 2 and Cell 5 which is displayed in the output shown on Figure 4.2.

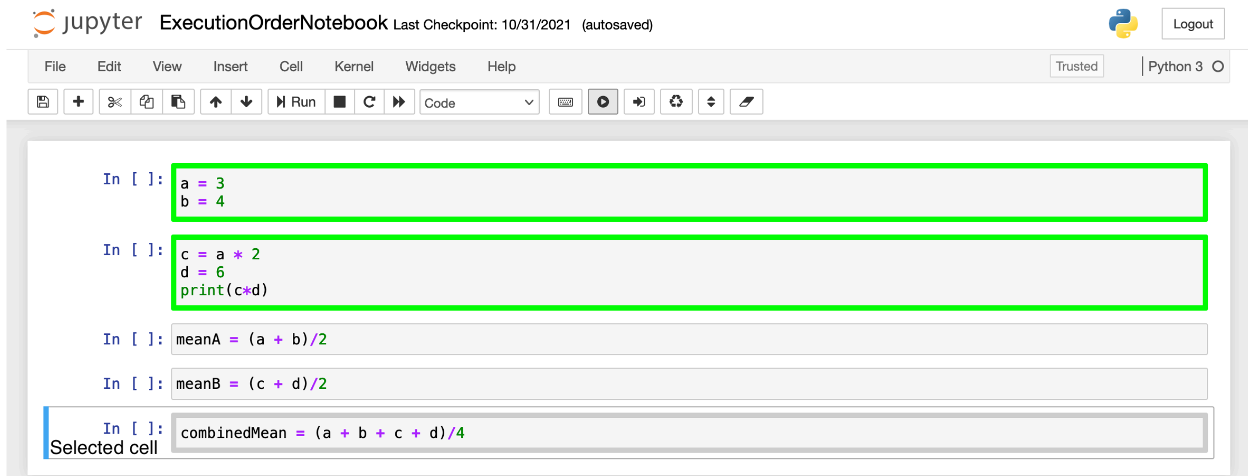


Figure 4.2: Notebook after Running Plugin

After this, the user sees that cells 1 and 2 are highlighted, and cell 5 is emboldened to indicate that is the selected cell, and can easily surmise that the execution order is Cell 1, then Cell 2, and then Cell 5. After seeing this, if the user wants to clear the highlighted cell borders, they can press the eraser button in the plugin and get the normal state of the notebook. Refreshing the notebook also clears the notebook output.

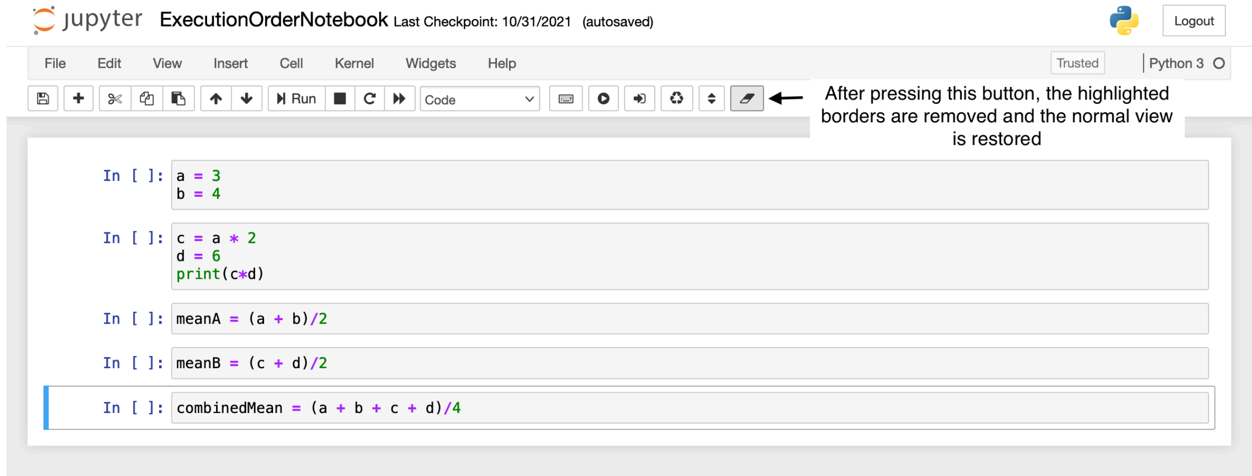


Figure 4.3: Notebook after Clearing Borders

Reorganizing Cells. In the case where there is only one possible cell execution order, the plugin will reorganize cells to reflect the execution order on top of highlighting the cells. Here is an example below.

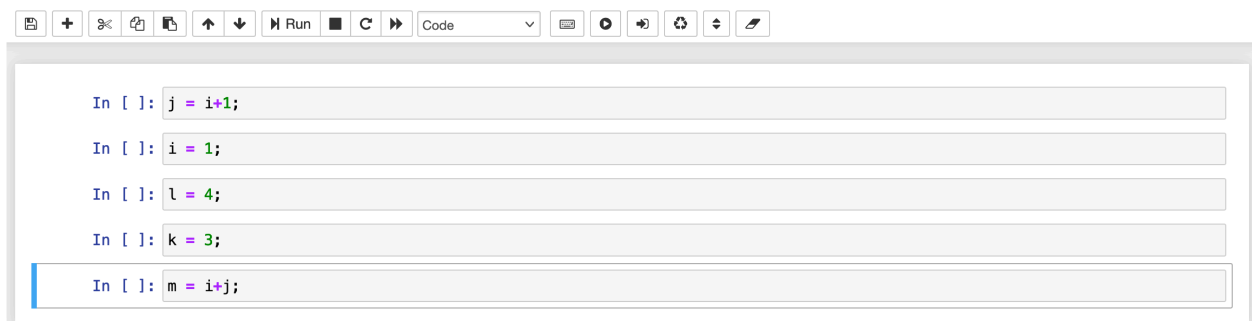


Figure 4.4: Notebook Before Running Plugin

In the notebook above we have a set of five cells, with the fifth cell being selected. We will click the play button to determine the execution order of the fifth cell, which is the second cell, then the first, then the fifth cell itself.

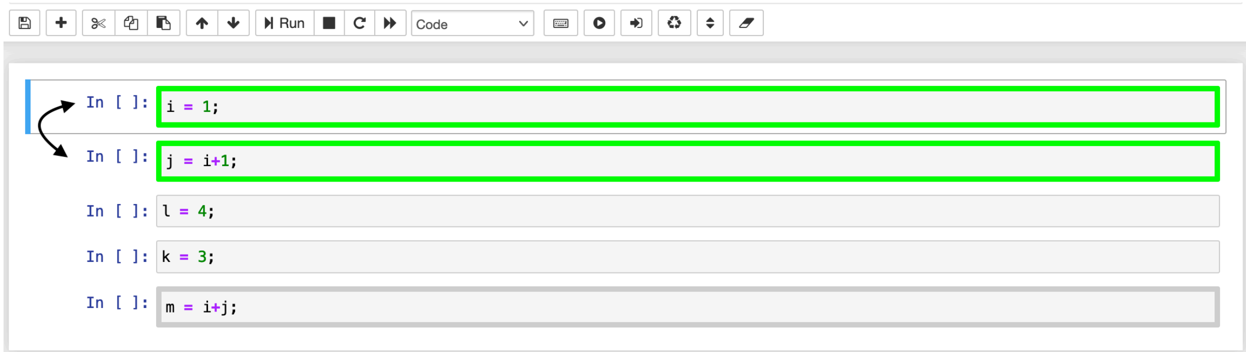


Figure 4.5: Notebook After Running Plugin

After running the plugin, the notebook recognizes that the cells do not represent the execution order and swaps the first and second cell, as well as highlighting the execution order. After this reorganization of the cells, the notebook visually reflects the execution orders it contains.

Handling Multiple Cell Execution Orders One thing we discovered was that sometimes, users reuse the same non descriptive variables throughout their notebook and as a result, we can derive multiple potential execution orders. With just the cells contents, it is not possible to determine the singular execution order. In this case, we would have to represent these cell orders to the users, for them to test and select the correct one.

In the plugin we use different colors to represent different cell execution orders. This is sufficient when the different execution orders are distinct and share no common cells. However, if a cell occurs in multiple executions, it only represents the color of the latest execution. In Figure 4.6 for example, we have three distinct semantically accurate executions. Cell1-Cell3 is one execution, Cell2-Cell3 is another and the last is Cell1-Cell2-Cell3. However, the only visible execution shown to the user is Cell1-Cell2-Cell3 as the red border overrode the previous borders. The display implies only one possible execution which is incorrect and can deter users from recognizing the other two execution orders.

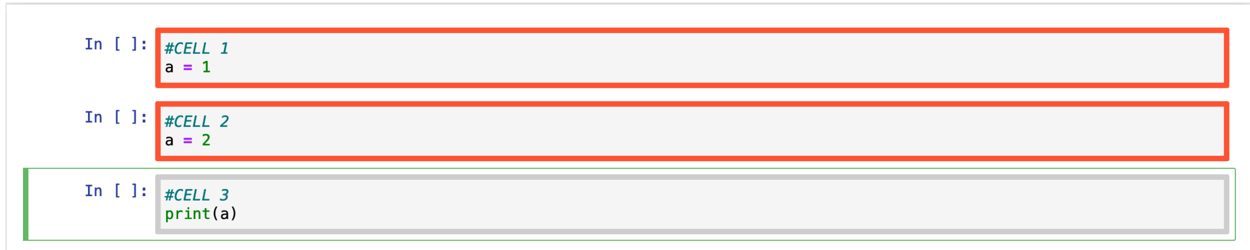


Figure 4.6: Notebook with Overwritten Executions Paths

It is imperative to develop the UI to better represent multiple cell execution orders to the user. One approach is to add multiple borders to cells that are present in multiple execution orders. Another approach is to create a small interactive window, preferably with the potential execution paths in a graph, so the user can switch between different potential execution orders as opposed to seeing them all at once,

Using Cell Output to Derive Single Execution Order One idea that has been explored in the efforts to determine the singular correct cell execution order is, if the user has already run the cells and there is output for the target cell, then, the multiple derived execution orders could be run and tested against the output to identify the correct execution order. While we have developed code that performs this procedure, there are many complications. The major complication being that any python code executed would be run on the same kernel. This could affect the user's development environment in terms of stored variables which could change the output of any cells they run. Resetting the kernel after this, could lead to the user losing valuable output.

Implementation Details To develop this tool, we wrote a Jupyter Notebook extension, which allows users to extend the functionality of the basic Jupyter Notebook environment with executable JavaScript code. The first step in the algorithm was to use the Jupyter library and obtain all the Cell metadata and contents. Then after parsing through and extracting only the Cell contents and formatting it, we pass it as an input to a python

script which is run on the same iPython kernel as the notebook. The python script parses through the input. Using the AST python library, we parse through the AST trees of the cell and based on whether the node is a Load or Store type add the variable to the def-use set. After developing the def-use sets, we start from the selected cell and build our cell dependencies which we then use to determine the execution orders. The execution orders are then formatted and returned by the python script. Lastly, to display the results to the user, the algorithm modifies the html and css attributes of the notebook to highlight the cells that belong to the execution order.

4.5 Evaluation

We use the following three criteria to evaluate the effectiveness of our technique: time savings, accuracy, and ease of use.

User Study Details: To evaluate our approach on time savings and ease of use, we carried out a user study with 9 participants. When selecting participants, we looked for people with programming background in python and experience in the computational notebook environment. Our 9 participants were a mix of graduate and undergraduate students in engineering fields. The sample population does contain students who have completed significant computer science curriculum and does not represent a portion of user class such as statisticians or researchers who may not have as much formal coding education.

Participants were sent instructions to setup the development environment for Jupyter Notebook and install the plugins we developed. In addition to this, participants were also given a google form with instructions on how to participate in the user study and questions pertaining to each task. The user study took about 25 to 35 minutes to complete.

Participants were provided with two notebooks, one used in the control case, and one used in the test case. Both notebooks were designed to be similar in complexity and comprehension, as both used the same data types and control structures while containing two distinct execution orders. With the control case participants were asked to analyze the notebook and identify the execution order for the last cell. With the test case participants were asked to perform the same task on the test notebook, however, this time they were instructed to use our cell dependency and reorder technique.

Control Case: As a control case, participants were provided a 10-cell notebook that contained two separate and unique execution paths ending in cells 9 and cells 10. The cell execution path for cell 9 calculated and displayed the average distance from mean, while the cell execution path for cell 10. The execution paths for both cell 9 and cell 10 were out of order.

The participants were asked to start a timer, and then analyze the notebook and determine which cells and in what order they must be executed to receive the correct output for cell 10. After determining the cell execution order, users were asked to stop the timer and record how long it took for them to manually determine the cell execution order. After this, users were asked to run the cells according to their order and record the output of the 10th cell. Lastly, users were asked to describe the procedure that they followed to determine the cell execution order and then score this task on how difficult it was to determine the execution order. The scoring was done on a Likert scale of one to ten, with one being 'very easy' and 10 being 'rather difficult'.

Test Case: As a test case, participants were provided with a 9-cell notebook that also contained two separate and unique execution paths ending in cells 8 and cells 9. The cell

execution path for cell 8 calculated the frequency and cumulative percent counts of a list of 10 numbers. The cell execution path for cell 9 however, only measured the percent counts of a different set of 10 numbers. The execution paths for both cell 8 and 9 were again not in order.

The participants were then asked to start a timer, however instead of manually deriving the execution order for cell 9, they were instructed to use the execution reorder tool that we developed. After receiving output from this tool they were asked to identify the execution order and write it down. After this, participants were instructed to stop the timer and record the time it took to determine the cell execution order using the plugin. Participants were then asked to give a short description of how they determined the execution order. Lastly, like in the control case participants were asked to score how difficult it was to determine the execution order of the 9th cell using the plugin. The scoring was done on the same scale of one to ten, with one being 'very easy' and ten being 'rather difficult'. Participants were also given a space to justify their scoring.

Control Results and Analysis: In the discussion that follows for the sake of anonymity we will refer to participants as P1 to P9.

In our control case, seven out of nine participants accurately determined the correct execution order. One out of the remaining two participants had the right execution order with one extra cell. The other participant had the completely incorrect order with all 10 cells listed in the execution order.

Seven out of nine participants had received the correct standard deviation value. One of the remaining two had written error, however they explained later that this error was due to a python setup issue on their machine, and after examining their results, they did derive the correct execution order. The last participant received an error due to their execution order

being incorrect.

On average, it took users around 5 minutes and 33 seconds to determine the execution order manually. The fastest participants were able to complete the task in about 3 minutes. The slowest participant took around 15 minutes, and for this reason it is important to include that the median time was 4 minutes.

When asked to describe their procedure, all participants mentioned that they used variable dependencies or searched for variable names across cells to determine the execution order. One example of this is P2's response in which they stated that they 'start[ed] with cell 10 and work[ed] backwards to find variables which needed to be defined before running the sequential cell.'

On a scale of one to ten, on average participants rated the task's difficulty to be a 5.1, with the lowest score being a three and the highest score being a ten. P8 who was one of the four participants to rate the task as a three in difficulty stated that "the task in not 'difficult' just inconvenient." P8 who also scored the task as a three in difficulty stated that the ordering of the cells was confusing and that they personally would "have moved cell 5 to the position of cell 2." Of the participants who offered justification for their rating, P2 had rated the task as the most difficult at a six. P2 specifically mentioned that having multiple execution paths made this task more difficult and confused them.

From the results in the control case, we can see that most users are able to determine the cell execution order of the 10th cell in a notebook with two distinct execution orders accurately. Participants seemed to universally follow the same procedure of searching variable names from the target cell up and determining dependencies. While five out of nine participants rated the task's difficulty as a five or below, with four stating it as three, these participants in their justifications mentioned the task to be 'confusing' or 'inconvenient.'

Test Results and Analysis: In the test case, seven out of nine participants accurately determined the correct execution order of 1-5-2-7-9. P3 and P9 were the two participants who wrote the incorrect execution orders, in fact they stated execution order of cell 8 rather than cell 9. This indicates that it is possible there is an UI issue where the participants believe the plug-in is indicating the incorrect execution path. Another possibility is user error, where participants mistakenly looked to solve the execution order of cell 8 instead of cell 9. The results of the test case show no improvement over the control case.

The seven participants who derived the correct execution order had also received the correct output. While the two who had not received either an error or incorrect output values.

When asked to provide the time it took to determine the execution orders, two participants provided relative terms such as under a minute. If we treat these entries as a minute each, the average time it took users to determine the cell execution order after using the tool is 33 seconds. This provides a significant improvement over the control case, where the average was 5 minutes and 33 seconds, and the median was four minutes. This supports our hypothesis that with the technique and our approach we can offer users of computational notebooks significant time savings when determining execution orders.

On a scale of one to ten, on average participants rated the task's difficulty to be a 1.44, with the lowest score being a one and the highest score being a three. P8 justified their rating of one because the tool made the task automated. P6 who also assigned a score of one, stated that "the plugin ordered everything in the right order and made everything simpler." This offers a significant improvement over the average score of 5.1 and supports our hypothesis that with our approach we can make the task of identifying execution orders and cell organization easier for users.

Testing For Accuracy: To test for accuracy, we developed a test notebook of five cells, where the fifth cell is dependent on each of the preceding four cells. We then tried every possible permutation of the four cells and ran the tool to see if it would identify the correct execution order and arrange the cells in the correct order.

When doing this, we found that of the 24 possible permutations the tool was successful in both identifying and reorganizing the cells in eight of these cases.

After analyzing the cases in which it failed, we saw that the tool was able to correctly identify the cells in the execution order for an additional five instances, but did not rearrange the cells properly and required users to swap positions between two cells. An improved technique to reorder the cells would fix the issue in these five instances.

For the remaining 11 instances however, the technique failed to produce any potential execution order candidates. This indicates a limitation in static analysis parsing and that the technique in building cell dependencies is prematurely and erroneously eliminating potential cell dependency candidates as invalid.

Summary of Results: In terms our time savings our approach reduced the average time required to manually determine the cell dependency in the control case from 5 minutes and 33 seconds to 33 seconds in our test case which is around a 90% improvement in terms of time savings. The time savings shown with our approach supports our hypothesis that our technique can help reduce the time developers take to identify execution orders and improve developer productivity.

For ease of use, on a scale of one to ten, with one being very easy and ten being rather difficult, determining cell dependencies through our approach was assigned an average score of 1.44, while manually determining cell dependencies was assigned an average score of 5.1.

These results support our hypothesis that our approach can simplify the task of determining cell dependencies, and comments received from participants point towards the automation of the technique as a justification.

In terms of accuracy, we designed an experiment with a notebook with five cells, where for each permutation for the first four cells we analyzed whether the technique could accurately recognize the execution and reorder the cells of the notebook to match the execution order. In 24 possible cases, the technique correctly recognized the execution order in 13 cases and reordered the cells correctly in eight cases. The small scale of this experiment makes it difficult to generalize these results. In the future we will further our evaluation with a broader range of computational notebooks.

In Chapter 5, we address further pain points in computational notebooks by discussing and evaluating our approach towards code management in multi-context notebooks.

Chapter 5

Code Management in Multi-Context Notebooks

One substantial challenge when utilizing computational notebooks is representing multiple execution paths and contexts. This is a critical problem as many notebooks contain forks and contain divergent execution paths. Furthermore, many notebooks contain multiple cells in lieu of one as a form of version control. Without representing either of these situations adequately, users must spend excess time analyzing the code to comprehend the execution flow of the notebook when revisiting or sharing the notebook. We address the challenge of code management in notebooks with divergent execution paths by answering the following research question. *How can we represent multiple contexts within a notebook?* Our hypothesis is that we can replicate the visual representation of forks in execution flows to accurately represent multiple execution paths within a notebook. We realize our hypothesis with a real-world technique that supports cells in a horizontal arrangement in addition to the standard vertical arrangement. Furthermore, we leverage the use of abstract syntax trees to retroactively apply this formation to existing multi-context notebooks.

5.1 Introduction

Many Notebooks have different execution paths or multiple contexts; however, the visual representation of notebooks does not adequately represent this and requires users to analyze the code in order to recognize it. For users who are revisiting a notebook, or new users who are viewing the notebook for the first time, it can be more difficult and time consuming to extract the semantic meaning of computational notebooks with different execution paths or multiple contexts.

This problem space consists of a couple primary challenges. One challenge is that when users maintain multiple versions of a cell, or multiple parallel cells in a notebook, these sets of cells occupy a significant portion of the user's notebook space. This can make it hard for users to keep track of the execution flows in the notebook. Another challenge is that every cell in a Computational Notebook is represented equally and in a vertical order. Therefore, there is no visual cue for users to identify a set of cells representing multiple contexts.

To address the pain point of displaying cells in multi-context notebooks, we design a technique that allows users to insert cells in the same row as another cell, in addition to the standard feature of inserting cells in the same column. We refer to this as a matrix view. Our insight is that by replicating the way forks and branches are represented in execution flow models, a notebook can better represent its execution paths and make it easier for users to identify multiple contexts and extract the semantic meaning of the notebook. Another research insight is that notebooks with multiple contexts or versions of cells can be spatially inefficient. By utilizing a matrix view and inserting parallel cells in the same row as their siblings, we can greatly reduce the notebook's vertical work-space. We recognize that there are many existing notebooks that could benefit from this technique, however it would require users to go back and analyze the code for forks, and then manually rearrange the notebook

using horizontal cells. For this, we have also designed a technique that leverages existing static analysis techniques similarly to Chapter 4 to construct def-use sets. Our insight is that if two sets of cells share the same def-use sets, then they are parallel cells and should be parallelized. We realize both techniques in a Jupyter Notebook plugin, which can add cells in the same row, or parallelize a notebook on command. We evaluate our approach on its ability to represent multiple contexts within a notebook. Our evaluation is performed through a user study with nine subjects, in which participants are asked to identify unique execution paths in a standard notebook and then identify unique execution paths in a different notebook represented in matrix view. In the user study, notebooks with a matrix view were assigned an average rating of 8.44 on a scale of one to ten on it's ability to represent a mutli-context notebook compared to the standard which was assigned an average rating of 4.77.

The rest of the chapter is outlined as follows. Section 5.2 provides some background and a motivating example to further illustrate the problem space. Section 5.3 discusses the approach used while Section 5.4 goes into specific details about the implementation and future work. Lastly, Section 5.5 evaluates the approach outlined in this chapter through a User Study.

5.2 Background

Motivating Example Take the Teacher's report example from Chapter 2.1. In this scenario, a teacher named Alex had collected data on student's exam score and the amount of time they studied for an exam to see if there is a correlation. Alex's notebook is shown again below:

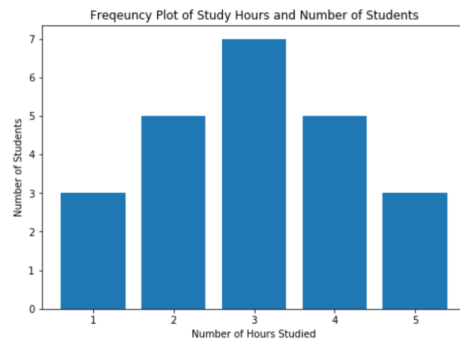
The Effect of Time Spent Studying on Exam Scores

Data was gathered in a poll from students asking how long they studied before taking an exam

In [1]:
```
import pandas as pd
import collections
import statistics
import matplotlib.pyplot as plt
```

In [2]:
```
#Imports Data from an Excel Sheet
timeGradeData = pd.read_excel (r'TimeVsGrade.xlsx', sheet_name='Sheet1')
```

In [3]:
```
#Frequency Plot Code
freq = collections.Counter(list(timeGradeData['Hours Spent Studying']))
fig = plt.figure()
axes = fig.add_axes([0,0,1,1])
axes.bar(list(freq.keys()), list(freq.values()))
plt.xlabel("Number of Hours Studied")
plt.ylabel("Number of Students")
plt.title("Freqeuncy Plot of Study Hours and Number of Students")
plt.show()
```



In [4]:
```
#Line Graph Code
#The teacher decides to take the mean of each group and plot it
xvalues = list(set(timeGradeData['Hours Spent Studying']))
yvalues = []
for val in xvalues:
    hour_df = timeGradeData.loc[timeGradeData['Hours Spent Studying'] == val]
    grade_lst = list(hour_df['Grade'])
    yvalues.append(statistics.mean(grade_lst))
hoursVsScore = {'Hours Studied': xvalues, 'Mean Exam Score': yvalues}
hoursScoreDf = pd.DataFrame(data=hoursVsScore)
print(hoursScoreDf)
```

```
   Hours Studied  Mean Exam Score
0              1        51.953333
1              2        58.150000
2              3        70.642857
3              4        83.334000
4              5        94.760000
```

In [5]:
```
plt.plot(xvalues,yvalues,'--bo')
plt.xlabel("Hours Studied")
plt.ylabel("Exam Score")
plt.title("Hours Studied vs Mean Exam Score")
plt.show()
```

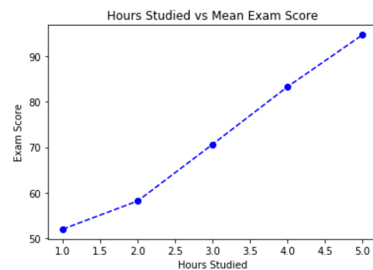


Figure 5.1: Teacher's Report

Alex presents this notebook to their class and shows the graph at the bottom as proof that there's a positive correlation between the amount of time spent studying and student's test scores. However, questions were raised on the Alex's results stating that the mean on its own is not reliable as it could be easily skewed by outliers and request to see the results when using the median score for each hour studied. Alex goes back to the notebook and adds another cell to calculate the median and computes the results. The edited notebook is shown below:

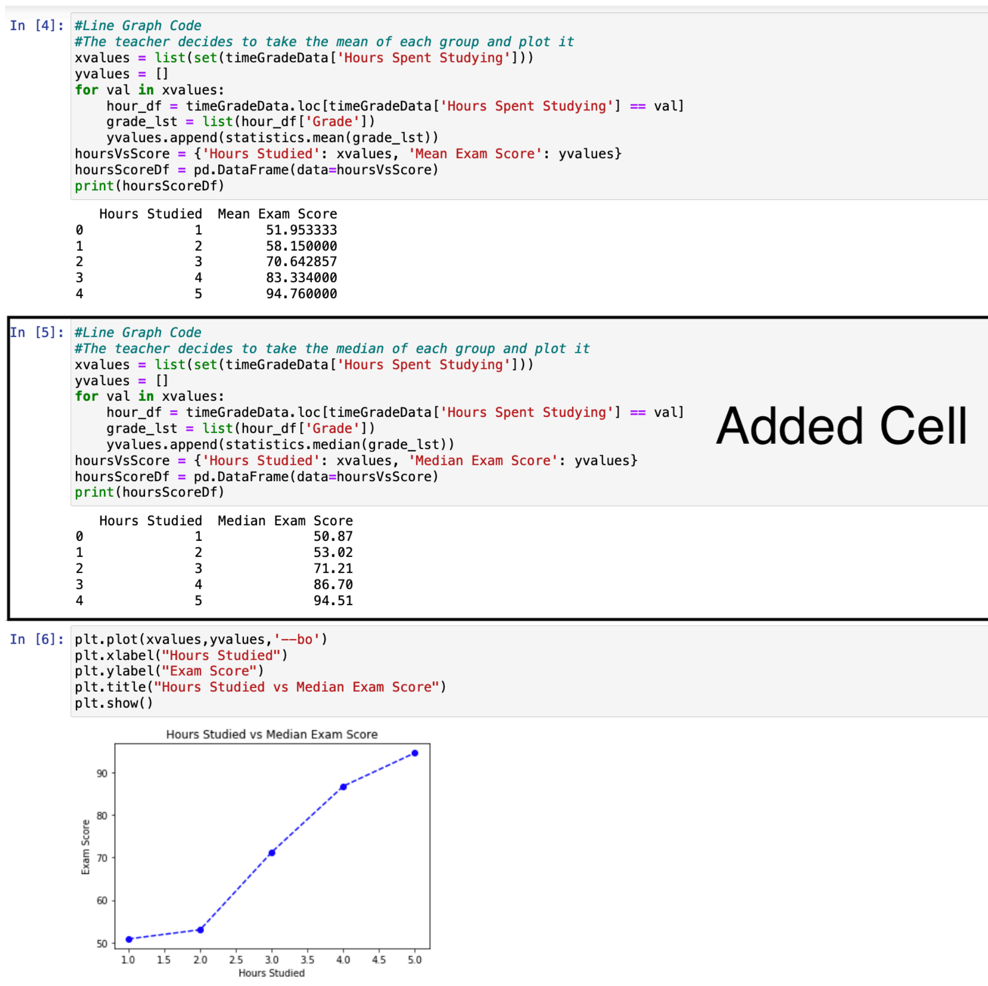


Figure 5.2: Teacher's Edited Report

In Figure 5.2, Alex has added a cell to compute the median directly beneath the cell to compute the means. Alex then runs that cell, and then re-runs the graph cell to get the

new results. This new graph also shows a positive correlation between study hours and exam scores. Alex's colleague, Blake, finds out about this experiment and asks Alex for the notebook so that they can run the same experiment on their class. For the next exam, Blake asks students how much time they studied and records their exam scores as well. He inputs this data into the notebook, uses the run all option to run every cell consecutively. As the median cell is below the mean cell, the graph uses the median data. Blake also wants to see the mean data. He runs the code again, but only receives the median results. As the cells are shown in vertical order, Blake wrongly assumes that the cells are represented in execution order. Only upon analyzing the cells and the code within him, does Blake realize that he needs to run the mean cell, skip the median cell and then run the graph cell to get the output his students desire. The display of the cells led Blake to misunderstanding the code progression in this notebook and required him to spend extra time to understand the semantic meaning of the code.

Problem Statement As stated in limitation 1, Computational Notebooks only allow for cells to be represented in a vertical fashion, which can misrepresent the code progression in a notebook and how the separate code cells interact with one another. This requires extra effort from the users to analyze the cell contents to derive the semantic meaning of the code. Another issue as we discussed in limitation 5, is that due to a lack of intuitive version control, many users keep multiple versions of cells in their notebook. These cells occupy a lot of vertical space and require the user to remember the differences in each version. To address both limitations 1 and 5, we have redesign existing notebook's code layout to support a matrix view–a view that allows code to be inserted in vertical and horizontal fashion.

5.3 Approach

Vertical cell representation is not adequate to exhibit nature code progression primarily due to parallel cells. Parallel cells are cells that lead to forks in the code execution or lead to different execution paths. In other words, they are cells that are run in place of one another and are not meant to both be run in the same execution. In the teacher's report example, the cell that computes the mean data and the cell that computes the median data are parallel cells. With the insight of parallel cells, we can develop a technique to better visually represent them.

With the understanding of parallel cells and how multiple execution threads can be formed in a notebook, we investigate the best method to visualize this within a notebook. Our insight is to replicate the standard execution flow patterns. We referenced flowcharts, which are used to model computations in neat structures[14]. Flowcharts are seen as the universal visual representation of code and could serve as a strong model to follow in neatly representing the execution flows of a notebook [5]. Flowcharts are also often preferred amongst novice programmers to represent the sequence of computation; however they can also be utilized by experienced programmers as well [3]. We utilize the structures present in flowchart design to arrange a notebook which can improve cell organization and aid users with a variety of programming backgrounds.

To represent parallel cells, we focus on the selection, also known as decision, structure. The selection structure is used to represent a fork in the execution where the result of a selection can result in two different sequences. Selection structures can converge back into one execution sequence, with this point in the being referred to as the convergence point [24]. Our problem space, however, is not completely sufficed by an unchanged adoption of the selection structure. Selection structures only support a selection result of true and

false, meaning there can only be two results or choices. However, in our application, it is possible to have more than two parallel cells, or multiple selections. To support multiple selections in a flowchart, a nested selection structure must be used [2]. Replicating this in our notebook would be unnecessarily complicated and would also be to the detriment of code organization. So, to remedy this issue we flatten the nested selection structure to support multiple selections.

In the execution flow of a notebook, the ”selection” is which of the parallel cells will be run and used in the following cells. The convergence point is the cell immediately executed following the selection. In Figure 5.3, we have an example notebook with three parallel cells, and in Figure 5.4 we have the matrix representation of the same notebook.

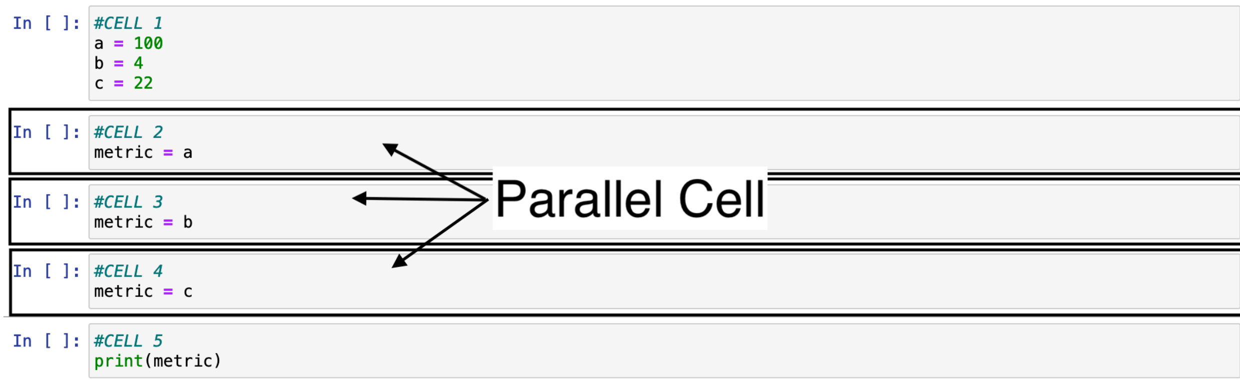


Figure 5.3: Example Notebook with Three Parallel Cells.

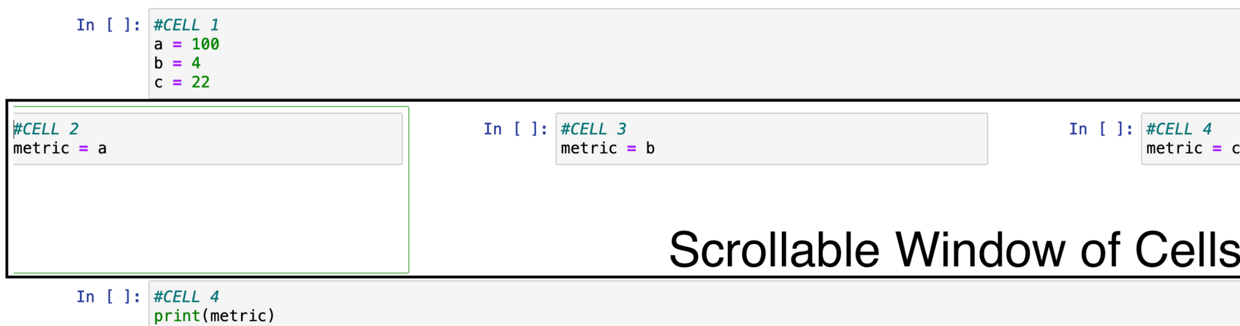


Figure 5.4: Example Notebook in Matrix View

Using these insights, we design a tool that inserts a cell into the same row as the selected cell and create a matrix view. Our approach also uses static analysis to derive the def-use

sets of a notebook, and analyze the def-use sets to determine which cells are parallel cells and parallelize a vertical notebook.

5.4 Implementation

Applying Horizontal Representation to Jupyter Notebook To mimic how flowchart represent multiple execution threads, we created a plugin to add cells in a horizontal order as well as vertical. Looking back to our motivating example, if Alex had access to the plugin, Alex could add the median cell in the same horizontal plane as the mean cell. This is shown in Figure 5.5.

Figure 5.5: Teacher's Report with Parallel Cell Support

Now the page notebook reflects that the mean and median cells do not exist in the same execution order, and rather represent a fork in the notebook.

With this visual display, if the user reopens the notebook, or shares it with a peer, it is much easier to comprehend the execution flows in the notebook and therefore decipher the semantic meaning of the code.

Parallelizing Existing Notebooks While the addition of horizontal cells will be useful for future notebooks, there are also many completed notebooks that would benefit greatly from the use of horizontal cells. Rather than require users to analyze previous notebooks to identify parallel cells and then manually reformat them to leverage the use of horizontal cells, we have designed a method to accomplish this instead. By identifying the AST trees and cell dependencies, it is possible to identify parallel cells in a notebook and arrange them in a horizontal formation.

This feature is shown in the example below:

```
In [ ]: #CELL 1
        a = 100
        b = 4
        c = 22

In [ ]: #CELL 2
        metric = a

In [ ]: #CELL 3
        metric = b

In [ ]: #CELL 4
        print(metric)
```

Figure 5.6: Notebook Before Parallelization Plugin is Run

```
In [ ]: #CELL 1
        a = 100
        b = 4
        c = 22
```

```
In [ ]: #CELL 2
        metric = a
```

```
In [ ]: #CELL 3
        metric = b
```

```
In [ ]: #CELL 4
        print(metric)
```

Figure 5.7: Notebook After Parallelization Plugin is Run

One use case for parallel cells or multiple contexts is version control. Users will often have multiple versions of cells in their notebooks, which leads to a lot of obsolete cells. An interface that allows users to compare the different outputs as a result of each parallel cells, and then decide which parallel cell to keep and which parallel cell to delete could be of great value in terms of version control.

Implementation Details To develop this tool, we wrote a Jupyter Notebook extension, which allows users to extend the functionality of the basic Jupyter Notebook environment with executable JavaScript code. After users press the button to activate our technique, we modify the html and css class attributes to create a scrollable container and insert a cell into the container next to the selected cell.

To create the parallelization feature, we use a process similar to our implementation in Chapter 4 to derive the def-use sets for each of the cells. Then the def-use sets of the cells are analyzed to determine if there are any sets of cells who have the same def-use sets. If there are, these sets of cells are flagged as parallel cells and return as output from the python script. Using the output from the python script, the parallel cells are flagged in the metadata and the html and css properties of their cells are modified to be presented in matrix view.

5.5 Evaluation

User Study Details: To evaluate our approach towards cell organization, we conducted a user study with nine participants. When selecting participants, we looked for users with experience in programming and computational notebook environments. Participants were sent setup instructions as well as a Google Form in which they were given instructions and asked questions about the experience. This took around 15 minutes to complete.

Participants were provided with two notebooks, similar in complexity and comprehension with one used in the control case and the other used in the test case. Participants were then asked to identify the unique execution threads in the notebook and evaluate how the code organization conveys the existence of multiple contexts within the notebook.

Control Case: As a control participants were provided with a multi-context notebook in the standard Jupyter Notebook view. After reading the notebook and analyzing the contents of the cells, we asked participants how many different execution flows that resulted in unique outputs for the 6th cell exist in the notebook and to list these execution paths. We then followed up by asking participants on a Likert scale of one to ten to rate the effort required to identify the unique execution paths in the notebook with one being 'No Effort at All' and ten being 'Significant Effort.' Lastly, we asked users to score how well the notebook organization represents these multiple contexts on a Likert scale of one to ten, with one being 'Not at All' and ten being 'Perfectly Represents Multiple Execution Orders.'

Test Case: In our test case, we asked participants to view a different multi-context notebook, however this notebook was presented in matrix view. Participants were then asked the same questions as in the control. How many unique outputs for the 5th cell exist in the notebook, and to score the effort required and notebook's ability to represent multiple

contexts.

Control Results and Analysis: For sake of anonymity specific participants will be referred to as P1 and so on.

In the control notebook, there were three unique execution paths. In this notebook, cells 3, 4 and 5 were the parallel cells, leading to unique execution paths of 1-2-3-6, 1-2-4-6, 1-2-5-6.

Seven out of nine participants correctly identified that there were 3 unique execution paths in the notebook. P4 stated that there were four execution paths, while P3 stated there were six execution paths. After analyzing the execution paths that P3 stated, it seemed that P3 had extraneous cells in their execution paths as well as execution paths that had the same output. Based on the execution orders that they listed, P3 did not recognize that by running one of the parallel cells after another it overwrites the values in the kernel and negates the previous parallel cell being run. P4 when asked to list the execution orders simply stated there were so many. This indicates that P4 may have thought of all the possible permutations of the cells, and not recognizing how the parallel cells interact with one another.

Of the seven participants who correctly identified that there were three execution paths, two incorrectly listed the execution paths. P1 wrote the execution paths as 1-2-3-6, 1-2-3-4-6, and 1-2-3-5-6. P7 on the other hand had listed the execution paths as 1-2-3-4-5-6, 1-2-3-5-4-6 and 1-2-4-5-3-6. While P1 and P7 correctly identified that there were 3 execution paths, they did not recognize that cells 3, 4, and 5 were meant to run in place of one another.

On average the participants rated the effort required to determine how many execution paths existed in the notebook as a 4.77 with the highest score assigned as a 10, and the lowest as 3. The median assigned score was a 4.

When asked how well the notebook's organization represented the average score was a 4.77

with a median of 6. The only participant to enter in justification was P2 who assigned a score of 6. P2 stated that ”it’s not difficult to tell there are parallel execution methods since each of the parallel cells are structured in a very similar way.”

From the results in the control case, we can see that while the seven out of nine participants were able to correctly identify the number of execution paths or contexts within the notebook, only 5 recognized the correct execution paths, with two running extraneous cells in their execution paths.

Test Results and Analysis: In the test notebook, there were three unique execution paths. In this notebook, cells 2, 3 and 4 were the parallel cells, leading to unique execution paths of 1-2-5, 1-3-5, 1-4-5.

In the test with the matrix view, six out of nine participants correctly stated that there were three execution paths. These six participants also correctly listed the three executions for the next question. Two out of nine participants stated that there were two executions, and both correctly identified both cell 2 and cell 3 as parallel cells as well as their respective execution paths. To view cell 4 however, users were required to scroll through the row, as it is not shown when the notebook loads. This indicates that the current user interface is not prominent enough to convey to the user how many cells are in the row and whether the user has to scroll or not.

P3 stated that there were six execution paths. In the execution paths they listed each of the parallel cells in every possible order, leading to six outcomes. P3 later explained that the shared row implied that the cells needed to be run at the same time. To address this, the tool could improve the GUI by adding connecting lines between the cells prior to the row and each of the parallel cells. This would indicate that each of the parallel cells are disconnected from one another and reside in separate execution flows.

On average participants rated the notebook's ability to represent multiple contexts as an 8.44, with the median value being 9.

The results from the user study support our hypothesis that by implementing a matrix view we can better represent multiple contexts in a notebook. However, the user study also indicates that the task of identifying multiple contexts or parallel cells is not very difficult, with users mentioning that as the contents and form of the cells are very similar it is not difficult to surmise that they are run in place of one another.

Summary of Results: On a scale of one to ten, the matrix view was assigned an average score of 8.44 for it's ability to represent multiple contexts, compared to the standard view which was assigned an average score of 4.77. A further analysis of the responses however, shows no improvement in number of participants correctly identifying the number of execution orders with the matrix view compared to the standard view. One reason for this is that two participants did not recognize that the rows were scrollable and did not see the third cell. Another participant indicated that the graphical design implied that the parallel cells were meant to be executed at the same time rather than separately. Due to the small scale of our user study, we can not generalize these results. However, we plan on improving the UI based on the feedback we received and expanding this user study to more computational notebook users.

Chapter 6

Conclusions

To address limitation 2 of cell-based execution and limitation 3 of the limited programming background of some computational notebook users, we designed and implemented a tool that uses AST parsing to develop def-use sets for the cells in notebook which are analyzed to derive the cell dependencies, and if necessary, reorder the notebook to better match the cell execution order for a given cell.

In a user study this tool was shown to improve the time it takes for users to determine the cell dependency for a cell by about four to five minutes. Our approach was also judged to make the task of determining cell dependencies easier by 3.56 points on a scale of one to ten. This supports our hypothesis that through our approach we can make determining cell dependencies easier and less time consuming, thereby improving developer productivity.

To improve this tool, we plan on improving the GUI to better show multiple cell dependency candidates to the user. We also plan on executing the minimal set of potential cell dependency candidates in the background and comparing the output to the desired output to derive the singular correct cell dependency order for a given cell.

To address limitation 1 of limited display options and limitation 5 of lack of version control we designed a new graphical representation for users to view cells that we call the matrix view. In the matrix view, cells can be arranged in rows in addition to the standard column view to better represent divergent execution flows or multiple versions of cells. We also added

a feature to analyze existing standard notebooks and analyze their static dependencies to apply the matrix view to these notebooks.

In a user study, the ability of a standard notebook to represent multiple contexts was judged as a 4.77 on a scale of one to ten. Our approach with the matrix view was judged on average to be an 8.44. This is an improvement over the control score of 4.77. The results of our test case support our hypothesis that by implementing a matrix view, we can better represent multi-context notebooks to users. However, with improvements to the GUI to better represent the number of parallel cells and emphasize that the parallel cells are disconnected from one another, the effectiveness of this tool can be improved further. We believe that in reality, computational notebooks are much more complex, long and pose more difficulty in program comprehension. In future, we plan to incorporate the feedback on the UI and expand this user study to large range of notebook programs and diverse range of users.

Bibliography

[1] James D. Hollan Adam Rule, Aurélien Tabard. Exploration and explanation in computational notebooks. *CHI '18: Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 1(32):1–12, 2018.

[2] Daniel Buck and Andreas Rau. On modelling guidelines: Flowchart patterns for stateflow. *Gesellschaft für Informatik, FG 2.1. 1: Softwaretechnik Trends*, 21(2):7–12, 2001.

[3] Kanis Charntaweekhun and Somkiat Wangsiripitak. Visual programming using flowchart. In *2006 International Symposium on Communications and Information Technologies*, pages 1062–1065. IEEE, 2006.

[4] Souti Chattopadhyay, Ishita Prasad, Austin Z Henley, Anita Sarma, and Titus Barik. What's wrong with computational notebooks? pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020.

[5] Thad Crews and Uta Ziegler. The flowchart interpreter for introductory programming courses. In *FIE'98. 28th Annual Frontiers in Education Conference. Moving from'Teacher-Centered'to'Learner-Centered'Education. Conference Proceedings (Cat. No. 98CH36214)*, volume 1, pages 307–312. IEEE, 1998.

[6] Daniel M German, Bram Adams, and Kate Stewart. cregit: Token-level blame information in git version control repositories. *Empirical Software Engineering*, 24(4): 2725–2763, 2019.

[7] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. Man-

aging messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2019.

[8] Ken Kennedy. Use-definition chains with applications. *Computer Languages*, 3(3):163–179, 1978. ISSN 0096-0551. doi: https://doi.org/10.1016/0096-0551(78)90009-7. URL https://www.sciencedirect.com/science/article/pii/0096055178900097.

[9] Mary Beth Kery, Amber Horvath, and Brad A Myers. Variolite: Supporting exploratory programming by data scientists. In *CHI*, volume 10, pages 3025453–3025626, 2017.

[10] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. *The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool*, page 1–11. Association for Computing Machinery, New York, NY, USA, 2018. ISBN 9781450356206. URL https://doi.org/10.1145/3173574.3173748.

[11] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. *Towards Effective Foraging by Data Scientists to Find Past Analysis Choices*, page 1–13. Association for Computing Machinery, New York, NY, USA, 2019. ISBN 9781450359702. URL https://doi.org/10.1145/3290605.3300322.

[12] Sean Kross and Philip J. Guo. Practitioners teaching data science in industry and academia: Expectations, workflows, and challenges. *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, 2019.

[13] Michael Muller, Ingrid Lange, Dakuo Wang, David Piorkowski, Jason Tsay, Vera Liao, Casey Dugan, and Thomas Erickson. How data science workers work with data: Discovery, capture, curation, design, creation. pages 1–15, 04 2019. ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300356.

[14] Isaac Nassi and Ben Shneiderman. Flowchart techniques for structured programming. *ACM Sigplan Notices*, 8(8):12–26, 1973.

[15] Deborah Nolan and Duncan Temple Lang. Computing in the statistics curricula. *The American Statistician*, 64:97–107, 05 2010. doi: 10.1198/tast.2010.09132.

[16] Maxim Rabinovich, Mitchell Stern, and Dan Klein. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535*, 2017.

[17] Jonathan Reades. Teaching on jupyter. *Region*, 7(1):21–34, 2020.

[18] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. Ten simple rules for reproducible research in jupyter notebooks. *arXiv preprint arXiv:1810.08055*, 2018.

[19] Adam Rule, Ian Drosos, Aurélien Tabard, and James D Hollan. Aiding collaborative reuse of computational notebooks with annotated cell folding. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–12, 2018.

[20] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. Ten simple rules for writing and sharing computational analyses in jupyter notebooks, 2019.

[21] Adam Carl Rule. *Design and use of computational notebooks.* University of California, San Diego, 2018.

[22] Pavle Subotić, Lazar Milikić, and Milan Stojić. A static analysis framework for data science notebooks. *arXiv preprint arXiv:2110.08339*, 2021.

[23] Sergey Titov, Yaroslav Golubev, and Timofey Bryksin. Resplit: Improving the structure of jupyter notebooks by re-splitting their cells. *arXiv preprint arXiv:2112.14825*, 2021.

[24] Xiang-Hu Wu, Ming-Cheng Qu, Zhi-Qiang Liu, Jian-Zhong Li, et al. Research and application of code automatic generation algorithm based on structured flowchart. *Journal of Software Engineering and Applications*, 4(09):534, 2011.

[25] Yifan Wu, Joseph M. Hellerstein, and Arvind Satyanarayan. B2: Bridging code and interactive visualization in computational notebooks. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, page 152–165, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450375146. doi: 10.1145/3379337.3415851. URL https://doi.org/10.1145/3379337.3415851.

[26] Mengya Zheng, Xingyu Pan, and David Lillis. Codex: Source code plagiarism detection based on abstract syntax tree. In *AICS*, pages 362–373, 2018.

[27] Nazatul Nurlisa Zolkifli, Amir Ngah, and Aziz Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018.

Appendices

Appendix A

User Study

A.1 User Study Questions

Control Unordered Notebook

1. Please view the 10th and last cell and do not review the contents of any other cells. This cell's contents are "print("The Standard Deviation of this data is: ", stdDev)." Start a timer. Determine what cells must be run and in which order they must be run in order to receive the right output upon execution of the 10th cell. After you have derived the cell execution order, jot it down and record how long it took to perform this procedure.

2. What is the cell execution order for the 10th cell.

3. Please run the cells as according to your execution order. What is the output you received from cell 10?

4. How long did it take you to determine the cell execution order?

5. How many attempts did it take for you to determine the execution order?

6. Please write a short description giving us an insight into how you determined the execution order.

7. From a scale of 1 to 10, how difficult did you find this task? (Likert scale from 1 to 10 with 1 being 'Very Easy', and 10 being 'Rather Difficult'.)

8. If you'd like, you may enter in the justification behind your rating.

Test Unordered Notebook

1. Please view the 9th and last cell and do not review the contents of any other cells. This cell's contents are "print(percentList)." Instead of you manually deriving the cell execution order, you will be using a plugin. Click on the 9th cell, and then click on the button as shown in the image. You should see certain cells outlined in green. If you do not, please refresh the notebook. Start a timer, and with the output from this plugin, determine the cell execution order. Please write down the execution order in the short answer spot below, similarly to the previous section.

2. How long did it take to determine the execution order after receiving the output of the plugin?

3. Please run the cells as according to the execution order. What is the output?

4. How many attempts did it take to determine the correct execution order?

5. On a scale of 1 to 10, how easy was it to determine the execution order after utilizing this plugin? (Likert scale from 1 to 10 with 1 being 'Very Easy', and 10 being 'Rather Difficult'.)

6. If you'd like, you may enter in the justification behind your rating.

Control MultiContext Notebook

1. Please review the 6th and last cell in this notebook. The cell's contents are "print("The data's variability is measured as: ", metric)." How many different execution orders with unique outputs are there in this notebook that end in the 6th cell?
2. Please list the execution orders below. With each execution order on its own line. As in the previous sections the cells are listed numerically.
3. Please write a short description as to how you determined how many execution orders with unique outputs were in the notebook.
4. How much effort was required to determine how many execution orders were in this notebook? (Likert Scale from 1 to 10, with 1 being 'Not Effort At All' and 10 being 'Significant Effort')
5. On a scale of 1 to 10, how much does the organization of the cells in the notebook convey that there are multiple execution orders in this notebook? (Likert scale from 1 to 10 with 1 being 'Not At All', and 10 being 'Perfectly Represents Multiple Execution Orders'.)
6. If you'd like, you may enter in the justification behind your rating.

Test MultiContext Notebook

1. Please review the 5th and final cell of this notebook. Upon viewing the notebook, how many execution orders with unique outputs are there in this notebook ending in the 5th cell?
2. Please list the execution orders below, with each execution on its own line.
3. Please write a short description as to how you determined the number of execution orders in this notebook.

4. On a scale from 1 to 10, how much does the organization of the cells in this notebook convey that there are multiple execution orders? (Likert Scale from 1 to 10 with 1 being 'Not At All' and 10 being 'Perfectly Conveys Multiple Execution Orders')

5. If you'd like, you may enter in the justification behind your rating.