Facilitating Dependency Exploration in Computational Notebooks

Colin Brown
colinjbrown@niu.edu
Northern Illinois University
Dekalb, Illinois, USA

Hamed Alhoori
alhoori@niu.edu
Northern Illinois University
Dekalb, Illinois, USA

David Koop
dakoop@niu.edu
Northern Illinois University
Dekalb, Illinois, USA

ABSTRACT

Computational notebooks promote exploration by structuring code, output, and explanatory text, into cells. The input code and rich outputs help users iteratively investigate ideas as they explore or analyze data. The links between these cells–how the cells depend on each other–are important in understanding how analyses have been developed and how the results can be reproduced. Specifically, a code cell that uses a particular identifier depends on the cell where that identifier is defined or mutated. Because notebooks promote fluid editing where cells can be moved and run in any order, cell dependencies are not always clear or easy to follow. We examine different tools that seek to address this problem by extending Jupyter notebooks and evaluate how well they support users in accomplishing tasks that require understanding dependencies. We also evaluate visualization techniques that provide views of the dependencies to help users navigate cell dependencies.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**.

KEYWORDS

Computational notebooks, output-driven analysis, cell dependencies, notebook visualization

ACM Reference Format:
Colin Brown, Hamed Alhoori, and David Koop. 2023. Facilitating Dependency Exploration in Computational Notebooks. In *Workshop on Human-In-the-Loop Data Analytics (HILDA '23), June 18, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3597465.3605222

1 INTRODUCTION

Computational notebooks have become important components of computational science, providing a scratchpad for efficient, persisted analysis. Notebooks weave text, code, and outputs in a single document, with outputs immediately following the code that generates them. While notebooks can be used for quick analyses, they also provide important documentation of past investigations, helping authors and collaborators see exactly how an analysis was completed. Furthermore, notebooks are also being used in published results (e.g., [22, 34]). As notebooks are used for reflection and inspection, it becomes important that users can understand their past work and share it with others. Thus, we wish to minimize barriers to understanding and reuse.

In current notebooks, the freedom to write code often overlaps with the burden of later understanding it. While breaking code into smaller snippets interspersed with outputs and text makes finding components of a computation easier, the challenge is connecting the pieces together. All variables in a notebook are, by default, global, and thus a variable from one cell can be referenced by any expression in another cell. This extends to position; a reference in a cell above the cell where a variable is defined is allowed, though articles on best practices advise against this [27, 30]. These variables and their references implicitly define dependencies between the cells, but those references may be ambiguous.

Adding more structure to the cells, then, can potentially provide better experiences for users and readers alike. In dataflow notebooks, we let users explicitly indicate those variables they will use elsewhere while still providing a means to disambiguate multiple definitions. To address the ambiguity this introduces, we automatically track and persist the cells each reference is associated with and allow users to designate a particular "version" of the variable when a previous version is desired. This can be augmented with new operations–for example, one that reruns cells as variables were originally referenced and another that uses the current variable values. Because we persist the exact references, we can deterministically reproduce the computations as they were originally rerun.

While outputs help users search and locate, they still need to understand the entirety of the notebook, both as an overview of the work and as they work on specific tasks. This is especially important for readers who did not themselves create the notebook. With well-defined dependencies, we can build a dependency graph or other aids like a minimap [26] of cells and variables to facilitate overall understanding and navigation. At the same time, when a user is working in a particular cell, such visualizations highlight the specific connections to and from that cell. Note that users can shift between the visualization and the notebook, balancing the linear structure of the notebook with the non-linear explorations that occur.

The objective of our work is to evaluate approaches that extend the Jupyter Notebook framework [12, 18] to clarify or eliminate ambiguity in cell dependencies. In addition, we examine visualization techniques that assist users in understanding references between cells and navigating the notebook.

2 DEFINITIONS

A *computational notebook* is a sequence of code and text blocks called *cells* (see Fig. 1). Generally, a user executes individual code cells one at a time, going back to edit and re-execute cells as desired. This is in contrast to scripts where all code is executed at once. In

HILDA '23, June 18, 2023, Seattle, WA, USA

Figure 1: In a Dataflow Notebook, cells can have multiple named outputs, and referencing an output in another cell creates a dependency between the two cells.

addition, new cells may be later inserted between existing, already-computed cells, so it is possible that the semantics of a variable change. There exists a variety of different computational notebook environments [1, 11, 18, 25, 26, 35, 36, 43], all of which use text and code cells with computational results shown inline. Generally, these environments mimic paper notebooks that document a scientist's work and include text, computations, and visualizations. When archived, they provide a record of analysis and any discoveries as the notebooks persist both input code and the results.

While notebooks are being used to document analyses and store source code, their utility depends on the ability to repeat the executions and reproduce results. A notebook is *repeatable* if the original authors, who usually have access to the original environments and data, can successfully re-execute it. That same notebook is *reproducible* if others can do the same, given access to the documented resources or sufficient configuration information. Repeatability is, then, a more constrained form of reproducibility. As with source code, notebook reproducibility may be complicated by incompatible libraries or updates to data sources, but it is also complicated by the execution order of the cells [28]. While notebooks allows users to position cells in a linear, top-down order, the *execution order* defines the order that the cells were executed in. In addition, a notebook's execution order may have some cells unexecuted and others executed multiple times.

The dependencies between cells are important aids in helping users determine a potentially unknown execution order. If one cell defines a variable x and no other cell defines that variable, any cell that references x needs to be executed after the defining cell. The second cell is *dependent* on the first. When a variable is assigned to a value more than once in different cells, we have a potential *ambiguity* for any other references because those references could be to either of the assignments. We can extend the notion of dependency via transitivity; given a cell c, any cell that must be executed before c is an *upstream* dependency, and any cell that requires c to be executed first is a *downstream* dependency. Together, these form a directed *dependency graph* [10] connecting cells, and this graph is acyclic if we duplicate nodes for cells that are executed more than once. *Immediate* dependencies are directly related. When a cell is modified and executed, each downstream dependency is *stale* until it is re-executed.

In this paper, we will focus on one of the most popular computational notebook environments, Jupyter, and focus on notebooks written in Python [12, 18]. In that environment, cells are identified with a numeric identifier that is incrementally assigned when the cell is executed. However, executing a cell after it has been executed once overwrites the identifier with the latest count. When the notebook is reopened in a new session, the counter resets, meaning the numeric identifiers are not reliable.

3 RELATED WORK

3.1 Issues in Notebook Environments

There have been a number of papers and talks that have highlighted issues with current notebook environments (e.g. [8, 13]) spanning many facets including environment issues [27], versioning [17], and reuse [40]. One key issue is understanding the dependencies and execution order in notebooks as variables are referenced across cells. This increases the mental load on the user to remember what each variable represents and currently stores [38]. In program comprehension [39], users often refer back to the original assignments to remind themselves of this information [15]. In notebooks, users create cells to output the variable values and often don't use variables more than once [41].

3.2 Improving Notebook Environments

Work on addressing issues in notebook environments includes helping users understand and navigate the existing structure [14, 24, 31] as well as modifying the execution semantics or structure [21, 32, 33, 37]. Nodebook enforces a linear order for cells [33] while Observable [26] and ReactivePy [37] embrace a reactive execution where any cell change triggers all dependent cells to also execute. This reactive style works well for code that executes quickly but can be problematic for larger workflows where users might prefer to make multiple edits before triggering a re-execution. Dataflow Notebooks change execution semantics so that a dataflow dependencies [9] between cells are tracked and upstream dependencies are re-executed when necessary, using persistent identifiers to refer to variables [21]. IPyflow [32] uses program slicing to determine dependencies between cells, highlighting those cells that need to be re-executed. Vizier offers improvements to provenance and versioning which allow for the tracking of dependency information [5].

3.3 Visualizing Notebooks

Other solutions that help users understand notebooks rely on visual representations. Some simplify notebooks by summarizing content. NBSearch [23] displays a list of ranked notebook cells linked to search queries, arranging them based on search results. In contrast, ToonNote [16] aims to provide a visually pleasing overview of notebooks, drawing inspiration from the idea of literate visualization [44], which was influenced by Knuth [19]. Others, however, rely on external tools [7] in order to present a view into the notebook according to a pre-existing format.

To understand dependencies, a common representation involves constructing the dependency graph which can be visualized using a node-link diagram [2, 10]. In general source code, dependency graphs can span the entire codebase [4] or address changes over time [6]. However, dependency graphs can become large and hinder understanding [29]. Dependency graphs for notebooks are derived from variable references between cells, adding a layer on top of standard variable dependencies. Albireo cites the non-linear nature of notebook execution as a reason to use a directed layout based on similarities [42]. Observable's minimap [26] eschews full topology

and uses whiskers to communicate whether cells contribute to or depend on other cells (directly or indirectly). It also allows a user to interactively select a single cell to see links to immediate inputs and references and uses the left or right position to show any cells that indirectly contribute to or depend on the selected cell.

4 DETERMINISTIC DEPENDENCIES DEFINE EXECUTION

With the rich outputs inlined in notebooks, users can connect insights to the code in a cell that generated the output, but the problem of ambiguous dependencies makes it difficult to connect that cell to the code in the rest of the notebook that also contributed to the results. When variables are assigned throughout code, the dependencies between those variables can be difficult to follow. In notebooks, this is compounded by the fact that the dependencies span across cells. Because cells can be executed individually and may be inconsistent with each other, it is difficult to determine the execution order in order to ensure that the variables are properly defined. When references are ambiguous, this is impossible.

Dataflow Notebooks (DFNBs) make important changes to both help users locate variable definitions and ensure that dependencies are not ambiguous. First, each cell is assigned a unique, persistent identifier so that any re-execution does not break references to cells. Second, we build on the existing convention that a cell's outputs are listed in the last line of the cell. This is consistent with IPython's display rule that only the last expression should be displayed. In addition, through simultaneous assignment, cells may have *multiple* named outputs (see Fig. 1). Third, the cell is wrapped in a closure so that any local variables do not leak into other cells; it is not possible to refer to variables local to the cell in other cells. However, references to the exposed variables are now also references to intermediate outputs; instead of storing results in global variables that may or may not be shown, the data passed between cells is now always visible. Finally, we allow users to reuse variable names to mitigate variable recall issues and provide mechanisms to unambiguously refer to outputs with the same name.

Instead of using the mutable Jupyter cell numbers, DFNBs use a persistent identifier that Jupyter assigns to a cell when it is created. This identifier is unique and does not change when the cell is edited or executed. Thus, one can fix bugs or update a computation without worrying about any references to that cell becoming stale, as happens with traditional notebooks. These identifiers also persist across sessions. While the unique cell identifiers are useful internally, the random hexadecimal representations assigned by Jupyter are meaningless to users, so we allow users to add a meaningful *tag* to the cell to identify it.

If we restrict output names to be unique across the entire notebook, this forces the user to remember more variable names that will likely be less meaningful. Instead, we allow variables to be reassigned and output in different cells and use cell identifiers and tags to connect an output to a particular cell. If we define `x` in two cells, `a73bd0` and `9fe143`, we can refer to a output as `x$a73bd0`. Better, if that cell has a tag (e.g. `calcSpeed`), we can refer to it using the tag as `x$calcSpeed`.

We do not require users to use cell identifiers unless they wish to reference an "older version" and resist appending cell identifiers unless the reference is ambiguous. By default, an untagged variable references the output that was generated most recently, which is often the one a user recalls and wishes to reference. However, the notebook will always persist the references because if another version of a variable is created, we need to disambiguate all existing references. We can also add operations to enable re-executions that update cells to reference the latest variable definitions or mutations.

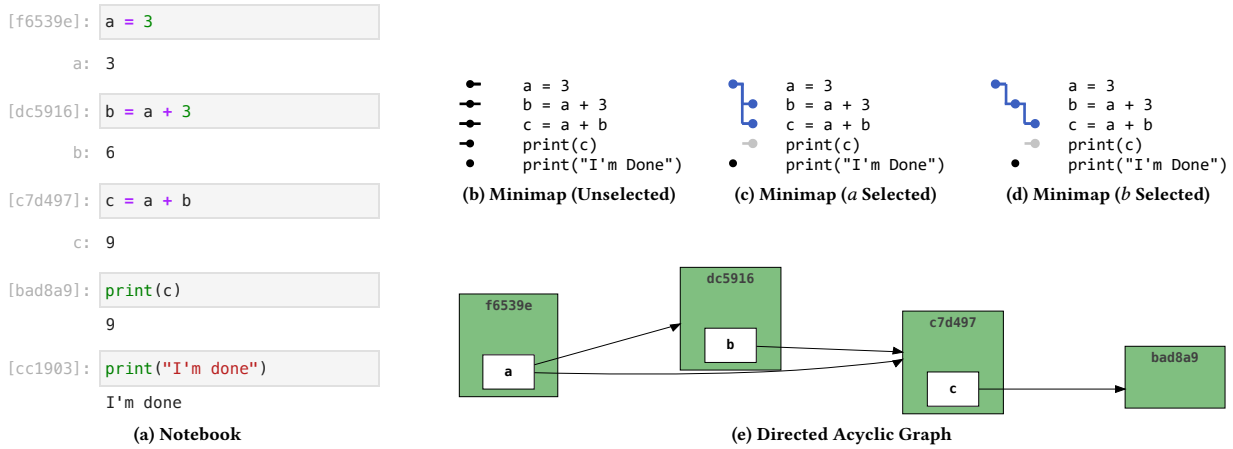
These changes allow the system to build a directed graph of dependencies based on the non-ambiguous references between cells. By disallowing cycles in the graph, we can allow the user to execute any cell, and the system will recursively evaluate upstream cells to ensure that the execution is consistent. Thus, for a notebook where every cell is deterministic and has been executed (to define dependencies), we can ensure that another user executing any cell in the notebook will receive the same result. This assumes that cell executions are side-effect free, and while closures help, this will not totally eliminate reproducibility issues. However, the same side-effect issues exist in the standard Jupyter environment so we are reducing the difficulty of reproducing notebooks.

5 NOTEBOOK VISUALIZATIONS

By disambiguating variable references, users can follow local dependencies in the notebook, but these do not provide an overview of the full structure of the notebook. By aggregating individual dependencies into a directed graph, we can create a visualization that can be used as such a map. We have applied two visualization techniques to Dataflow Notebooks: a node-link diagram that connects variables with edges, and a minimap visualization introduced by Observable [26]. While the node-link diagram is more familiar, the minimap provides a more compact representation that aligns with the vertical layout of the notebook (see Fig. 2). Since both visualizations have components tied to cells, users can navigate from the visualization to the notebook (or vice versa) by selecting those elements.

Since we disallow cycles in the dependencies in the notebook, the graph of those dependencies is directed and acyclic (a DAG). Because each dependency ties an output to its use in the code of another cell, we can construct edges between cells or between an output and a cell. Like Albireo [42], we use output-to-cell connections as they allow more specificity about the variables used and potentially highlight those that are unused. Thus, each cell is mapped to a node, but each of those nodes contains the specific variables (see Fig. 2e). Edges go from these inner nodes to the outer cell nodes. The layout of the DAG allows users to distinguish connected components that likely pair with sections of a notebook.

Unlike the node-link diagram, whose positions do not map to cell positions in the notebook, the minimap is constructed of nodes whose order matches the order of the notebook. Each cell is mapped to evenly-spaced vertical rows that each contain a circle and text with a snippet of code from the notebook. Each circle may have whiskers that extend to the left or right and indicate the presence of upstream or downstream dependencies, respectively. This overview helps users identify cells that are unconnected and those cells that are starting points in executions, often variable assignments. The initial overview shows nothing about any specific dependencies (see Fig. 2b).



(a) Notebook (b) Minimap (Unselected) (c) Minimap (*a* Selected) (d) Minimap (*b* Selected) (e) Directed Acyclic Graph

Figure 2: Both the minimap and directed acyclic graph visualizations can serve as overviews for understanding and navigating global structure and local dependencies of a notebook (a). With no selection (b), each minimap node shows the presence of dependencies (either upstream or downstream) via whiskers, but upon selection (c), (the cell containing b is selected) nodes are moved to indicate the relationship (or lack of) to the selection, and lines show immediate dependencies. Multiple views exist in the minimap for each individual variable that is reflected via selection as can be seen in (c) and (d).

However, the minimap is interactive so if the user selects a cell, the minimap updates to show all upstream and downstream dependencies for that cell by positioning them to the left and right, respectively. In addition, the immediate dependencies are connected with lines, helping users trace inputs back and follow the path of computation. Additionally, dependencies that are non-direct links to cells are highlighted in a grey color (see Fig. 2c) to indicate that there is still a dependence that exists among cells (e.g. *c* depends on *b* but the cell that prints *c* only directly depends on *c*). When selecting a new variable, the dependencies update to the selected variable (see Figs. 2c and 2d). In addition, as cells are edited and executed, the visualizations update to reflect the underlying dependency structure of the notebook.

6 EVALUATION

We conducted a user study to both understand the advantages and disadvantages of Dataflow Notebooks compared to other available techniques, and evaluate the utility of the visualization techniques. Because systems have addressed different issues, we have focused on tasks involving dependencies. The two-part study (a) compares performance on real-world tasks across systems; and (b) investigates whether the interactive visualization techniques improve the understanding of dependencies. In (a), we use data analysis scenarios where users must run and update existing notebooks. In (b), we compare two techniques, directed graph diagrams and minimap visualizations, to see if they improve performance in understanding connections between cells.

6.1 Tools

In the first part of the study, we used four tools: Jupyter Notebook [12, 18], Nodebook [33], ReactivePy [37], and Dataflow Notebook (Sec. 4). While some of these systems were created as prototypes and are not fully featured, we have found that they work well enough to accomplish the tasks we were interested in. Nodebook is an extension to Jupyter Notebook that restricts a user from running cells in a top-down manner so that a notebook will produce the expected results when running all cells [33]. It uses hashes of environments to determine that cells are out of order, and requires a user to *move* the cells to a correct order to execute the notebook. ReactivePy is an IPython kernel that automatically updates other cells that depend on a variable changed in one cell. During our user study we allowed ReactivePy to load a cached execution state so that the reactivity was being tested; Dataflow Notebooks allows for a previous stale execution.

In the second part of the study, we used our own implementations of the directed acyclic graph and minimap visualizations (see Sec. 5).

6.2 Tasks

Part 1: Real-World Re-computation. We tested two common tasks users face when using computational notebooks:(1) opening a saved notebook and re-executing that notebook, and (2) modifying the notebook (i.e., by changing a variable's value) and updating the rest of the notebook to obtain the new results. We structured the tasks so that cells could not be (re-)executed in top-down order, which meant participants had to figure out how to move or execute cells in a non-linear order. When modifying the notebook, they also had to make sure all cells were properly re-executed. In the standard Jupyter Notebook environment, all of these tasks must be manually completed by the participant, but other environments provide tools or enforce restrictions that help users.

All tasks used a portion of a previously written notebook that analyzed Kickstarter data. Participants only needed to execute and modify a single cell of these notebooks to complete the following two-part tasks:

- Compute the average success rate of Kickstarter categories using a given threshold, and then change that threshold and compute the new success rate.

- Compute the average amount pledged for a project, and then change the dataset filename and compute a new average amount.
- Compute the percentage of Kickstarter projects that are not US-based, and then change the excluded countries to include Great Britain and Canada and compute the new percentage.
- Compute the accuracy of a classifier over the dataset, and the change the dataset filename and compute a new accuracy.

Participants used used a different notebook tool for each task, and while the tasks shared the same dataset, they were self-contained. We hypothesized that all of the tools being tested will have improved performance compared to Jupyter Notebook.

Part 2: Dependency Visualizations. We tested whether the DAG or minimap visualizations aid users in understanding relationships between variables and cells. Here, we created notebooks by randomly assigning simple variables to either a literal or a simple arithmetic expression involving other variables; the order of these variable assignments is randomized. We asked participants two types of questions: (1) Does variable b depend on a? and (2) Select the variables that need to be recomputed when variable a is modified. The variable names (a and b) are randomized, and a simplified version of an example notebook is shown in Fig. 2a. Participants were given a static view of either the notebook, DAG, or minimap depending on the task. Each notebook/visualization contained fifteen cells and fifteen distinct variables; no variables were redefined or mutated. We used a pilot study to determine the number of variables for this task and found completion time increased at an extra linear rate. We hypothesized that users will be faster and more accurate using the visualizations.

6.3 Study Design

We randomly divided participants into three groups. In the first part, all participants completed their first task using the standard Jupyter Notebook environment, and then continued with the other tools in an assigned order depending on their group. In the second part, all participants again completed their first questions using the standard Jupyter Notebook environment, and then uses the minimap or DAG in a randomly assigned order.

Participants accessed the study through a web-based application that integrated tutorial pages, videos, and Jupyter environments. Participants had to read documentation and watch tutorial videos for each tool. In addition, there was a help panel for the current tool that was accessible at all times. In addition, an anonymous email form was provided if a participant got stuck on a task so that feedback could be provided. Participants completed surveys about their expertise, demographic information, and impressions of the tools. Timings and answers for each task were recorded. Participants were asked to score each tool being evaluated on a 5-point Likert scale after the completion of each task.

We advertised the study to undergraduate and graduate students in the Northern Illinois University Computer Science Department and recruited 33 participants. However, eleven of these results were dropped to low quality data, including responses taking too much time or implausible answers to tasks. Of the remaining 22 participants, 16 were male, 5 female, and 1 non-binary. There were 17 participants between 18 and 25 years old and 5 between 26 and 35. All participants had some level of experience using Jupyter notebooks and/or Python, and 13 said they were familiar with the notion of tracking variable dependencies.

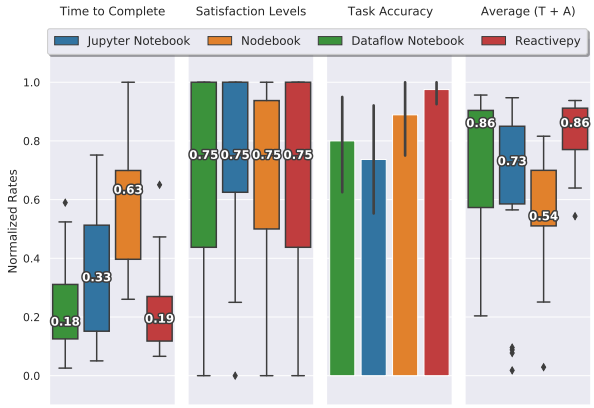


Figure 3: While accuracies were mostly within the margin of error, timings were significantly better in the reactive frameworks Dataflow Notebook and Reactivepy. Nodebook likely suffered because users found the enforced top-down ordering more difficult to understand.

6.4 Results

For the first task, we found that there was a significant difference in the amount of time it took to complete a reproducibility task when comparing most of the frameworks. For the second task, however, we did not reach a conclusive result about the relative performance of the minimap and directed acyclic graph diagrams, and need further analysis. We also report a combined measure that averages normalized time and accuracy measures to attempt to characterize frameworks that supported high-quality, efficient results. This allowed us to balance results where a participant rushed through a task without considering how accurate their answers would be.

Re-computation. In the re-computation tasks, we saw a significant difference computed via pairwise t-tests ($p < 0.05$) between all frameworks except for Reactivepy when compared to Dataflow Notebooks. The t-tests were calculated using the average (time + accuracy) values but also showed differences when looking at just accuracy. Differences in accuracy largely fell within the margin of error as seen in Fig. 3 with only a few participants having difficulties obtaining correct results. All timing values represented in Fig. 3 are scaled using the maximum participant taken time (23 minutes). The fastest task completion time was 41 seconds. Our results are complicated by the varying skill levels. Based on the reported values, we found participants fell victim to known pitfalls where they relied on previous results without re-executing previously executed cells. Nodebook, in particular, had longer completion times because (a) participants had to move cells in the notebook to obtain the proper order before execution, and (b) it incurred overhead doing environment hashing. Using Dataflow Notebooks and Reactivepy, participants achieved faster times with similar accuracy compared to Jupyter Notebook. This would suggest that reactive notebook frameworks offer advantages in dealing with cell dependencies.

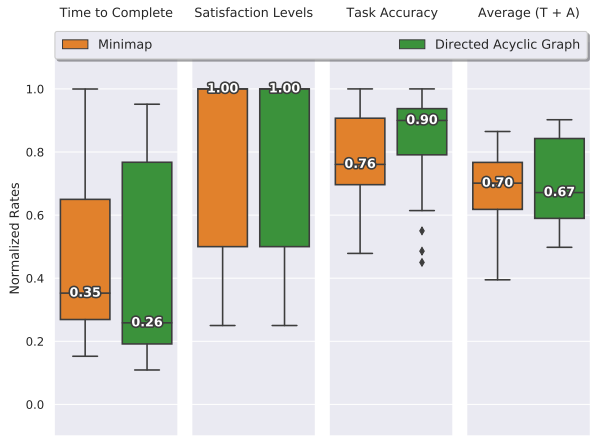


Figure 4: Comparing the time, accuracy, and satisfaction between the minimap and directed acyclic graph (DAG). Several of the users who completed tasks using the DAG faster were also more inaccurate.

Dependency Analysis. Fig. 4 shows that there isn't a significant difference between DAGs and the minimap in either accuracy or time to complete. (Due to differences in how the Jupyter Notebook task was set up, were unable to compare either technique to a baseline.) The timing data was scaled by the maximum participant completion time (23.5 minutes), and the fastest task completion was 41 seconds. Participants were generally more confident using the DAG as they took less time to complete their answers, but this was offset by their poor accuracy on this task as many participants who finished quickly also performed poorly. By combining completion time and accuracy measures into a single measure, we found that the minimap achieved more consistent results.

Feedback. During the study, participants could provide feedback through a contact form, and some of the submitted feedback highlighted interesting issues. One user thought that Nodebook was not working correctly because they did not understand that cells needed to be reordered in order to complete the notebook execution. We know many notebook files are distributed with cells whose executions are not in positional order [27], and we believe many users do not reorder cells to fix this. Other feedback concerned the horizontal layout of the directed acyclic graph and how this did not fit the participant's screen well. This problem was exacerbated by the side-by-side layout of the study, and highlighted how the minimap uses much less screen real estate than the DAG.

Skill Level. When controlling for skill level of the participants, there was no significant difference between participants who self-assessed as having lower proficiency in Python or limited experience working with Jupyter. However, these users were slower to complete tasks overall and represented many of the outliers seen in Fig. 3.

7 DISCUSSION

In general, our results indicate that reactive frameworks help users in reproducing notebook outputs. Even when users are guided through the process of reproducing results in notebook, there is still confusion when notebooks do not execute as expected and time lost spent trying to re-execute others' results. We found that accuracies in Jupyter Notebook were the lowest despite this being a common platform for sharing analyses and results. Perhaps when a notebook editor is also the notebook creator, personal knowledge helps guide changes. However, tools to trace and visualize dependencies through the notebook should aid users in understanding unfamiliar notebooks.

One potential threat to validity is the length of the study and how that affected participants' results, especially in the second part of the study. Initial trials with expert users with experience with directed acyclic graphs indicated that 15 variables was reasonable, but such static analysis is likely a more difficult task for less inexperienced users. Some participants took more than an hour to complete the study, and this may have contributed to mental strain or fatigue. In the future, we want to better understand the fatigue involved with manually doing static dependency analysis.

While our results for dependency visualization in notebooks were inconclusive, we believe the number of tools using DAGs for variable dependencies as well as the development of new techniques like minimap, points to their utility. The minimap seems to solve a key challenge in scalability as it can handle large numbers of cells and dependencies by leveraging interaction to hide dependency links unless a user selects a node that is an endpoint.

The DAG, in contrast, provides a global overview, but edge crossings often make understanding the relationships in large graphs very hard [3, 20]. We believe there is additional work that can be done in enhancing both visualization techniques with respect to dependency-related tasks by clustering related cells.

8 CONCLUSION & FUTURE DIRECTIONS

The development of frameworks and visualizations to help users to understand dependencies is an important step forward in improving notebook reproducibility. Initial results show promise that the changes in notebook extensions like Dataflow Notebook can aid users in understanding the dependencies between cells while simplifying execution. We also believe that visualizations like DAGs and minimaps aid users in navigating these dependencies. More work should be done to understanding when these visualizations work best and why they may perform poorly. We plan to expand our user study to more complex notebooks with more varied dependency graphs that more closely resemble real-world notebooks, and better control for differences in skill sets of participants. We also plan to investigate methods to make these new frameworks backward compatible with standard notebooks as this should help aid adoption.

9 ACKNOWLEDGEMENT

This work is supported in part by NSF Grant No. 2022443.

REFERENCES

[1] Apache Software Foundation. 2022. Apache Zeppelin. http://zeppelin.apache.org.

[2] F. Balmas. 2002. Using Dependence Graphs as a Support to Document Programs. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Comput. Soc, Montreal, Que., Canada, 145–154. https://doi.org/10.1109/SCAM.2002.1134114

[3] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. 2007. The aesthetics of graph visualization. In *Computational aesthetics in graphics, visualization, and imaging*, Douglas W. Cunningham, Gary Meyer, and Laszlo Neumann (Eds.). The Eurographics Association. https://doi.org/10.2312/COMPAESTH/COMPAESTH07/057-064 ISSN: 1816-0859.

[4] Krzysztof Borowski, Bartosz Balis, and Tomasz Orzechowski. 2022. Graph Buddy — an interactive code dependency browsing and visualization tool. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, Limassol, Cyprus, 152–156. https://doi.org/10.1109/VISSOFT55257.2022.00023

[5] Mike Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p13-brachmann-cidr20.pdf

[6] Michael Burch, Christoph Müller, Guido Reina, Hansjoerg Schmauder, Miriam Greis, and Daniel Weiskopf. 2012. Visualizing dynamic call graphs. In *Vision, modeling and visualization*, Michael Goesele, Thorsten Grosch, Holger Theisel, Klaus Toennies, and Bernhard Preim (Eds.). The Eurographics Association. https://doi.org/10.2312/PE/VMV/VMV12/207-214

[7] Lucas AMC Carvalho, Regina Wang, Yolanda Gil, and Daniel Garijo. 2017. NiW: Converting Notebooks into Workflows to Capture Dataflow and Provenance.. In *K-CAP Workshops*. 12–16.

[8] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. 2020. 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*. ACM, Honolulu HI USA, 1–12. https://doi.org/10.1145/3313831.3376729

[9] J. B. Dennis. 1980. Data Flow Supercomputers. *Computer* 13, 11 (Nov. 1980), 48–56. https://doi.org/10.1109/MC.1980.1653418

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[11] Google. 2022. Collab. https://colab.research.google.com.

[12] Brian E. Granger and Fernando Perez. 2021. Jupyter: Thinking and Storytelling With Code and Data. *Computing in Science & Engineering* 23, 2 (March 2021), 7–14. https://doi.org/10.1109/MCSE.2021.3059263

[13] Joel Grus. 2018. I Don't Like Notebooks. https://www.oreilly.com/library/view/jupytercon-new-york/9781492025818/video322524.html.

[14] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. 2019. Managing Messes in Computational Notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–12. https://doi.org/10.1145/3290605.3300500

[15] Derek Jones. 2004. Memory for a Short Sequence of Assignment Statements. *CVu* 16, 6 (2004), 1–15.

[16] DaYe Kang, Tony Ho, Nicolai Marquardt, Bilge Mutlu, and Andrea Bianchi. 2021. ToonNote: Improving Communication in Computational Notebooks Using Interactive Data Comics. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 727, 14 pages. https://doi.org/10.1145/3411764.3445434

[17] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E. John, and Brad A. Myers. 2018. The Story in the Notebook: Exploratory Data Science Using a Literate Programming Tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, Montreal QC Canada, 1–11. https://doi.org/10.1145/3173574.3173748

[18] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, Amsterdam, NL, 87 – 90.

[19] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (Feb. 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[20] Stephen G. Kobourov, Sergey Pupyrev, and Bahador Saket. 2014. Are Crossings Important for Drawing Large Graphs?. In *Graph Drawing*, Christian Duncan and Antonios Symvonis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 234–245.

[21] David Koop and Jay Patel. 2017. Dataflow Notebooks: Encoding and Tracking Dependencies of Cells. In *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*. USENIX Association, USA, 1.

[22] Laser Interferometer Gravitational-Wave Observatory (LIGO). 2016. Signal Processing with GW150914 Open Data. https://losc.ligo.org/s/events/GW150914/GW150914_tutorial.html.

[23] Xingjun Li, Yuanxin Wang, Hong Wang, Yang Wang, and Jian Zhao. 2021. NBSearch: Semantic Search and Visual Exploration of Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Yokohama Japan, 1–14. https://doi.org/10.1145/3411764.3445048

[24] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1093–1101.

[25] Stephen North, Carlos Scheidegger, Simon Urbanek, and Gordon Woodhull. 2015. Collaborative visual analysis with RCloud. In *2015 IEEE Conference on Visual Analytics Science and Technology (VAST)*. IEEE, USA, 25–32. https://doi.org/10.1109/VAST.2015.7347627

[26] ObservableHQ. 2022. Observable. https://observablehq.com.

[27] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, Montreal, QC, Canada, 507–517. https://doi.org/10.1109/MSR.2019.00077

[28] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2021. Understanding and Improving the Quality and Reproducibility of Jupyter Notebooks. *Empirical Software Engineering* 26, 4 (July 2021), 65. https://doi.org/10.1007/s10664-021-09961-9

[29] Martin Pinzger, Katja Grafenhain, Patrick Knab, and Harald C. Gall. 2008. A Tool for Visual Understanding of Source Code Dependencies. In *2008 16th IEEE International Conference on Program Comprehension*. IEEE, Amsterdam, 254–259. https://doi.org/10.1109/ICPC.2008.23

[30] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H. Nguyen, Sara Brin Rosenthal, Fernando Pérez, and Peter W. Rose. 2019. Ten Simple Rules for Writing and Sharing Computational Analyses in Jupyter Notebooks. *PLOS Computational Biology* 15, 7 (July 2019), e1007007. https://doi.org/10.1371/journal.pcbi.1007007

[31] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (Nov. 2018), 1–12. https://doi.org/10.1145/3274419

[32] Shreya Shankar, Stephen Macke, Andrew Chasins, Andrew Head, and Aditya Parameswaran. 2022. Bolt-on, Compact, and Rapid Program Slicing for Notebooks. *Proceedings of the VLDB Endowment* 15, 13 (2022), 4038–4047.

[33] StichFix. 2018. Nodebook. https://github.com/stitchfix/nodebook.

[34] Richard H. Styron and Eric A. Hetland. 2014. Estimated likelihood of observing a large earthquake on a continental low-angle normal fault and implications for low-angle normal fault activity. *Geophysical Research Letters* 41, 7 (2014), 2342–2350. https://doi.org/10.1002/2014GL059335 Notebook version: https://github.com/cossatot/lanf_earthquake_likelihood/blob/master/notebooks/lanf_manuscript_notebook.ipynb.

[35] The Sage Developers. 2022. *SageMath, the Sage Mathematics Software System*. The Sage Developers. https://www.sagemath.org DOI 10.5281/zenodo.6259615.

[36] Two Sigma Open Source. 2022. BeakerX. http://beakerx.com.

[37] California Polytechnic State University. 2019. Reactivepy. https://github.com/jupytercalpoly/reactivepy.

[38] A. Von Mayrhauser and A.M. Vans. 1993. From code understanding needs to reverse engineering tool capabilities. In *Proceedings of 6th International Workshop on Computer-Aided Software Engineering*. IEEE Comput. Soc. Press, Singapore, 230–239. https://doi.org/10.1109/CASE.1993.634824

[39] A. Von Mayrhauser and A.M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (Aug. 1995), 44–55. https://doi.org/10.1109/2.402076

[40] Jiawei Wang, Tzu-yang Kuo, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 138–149. https://doi.org/10.1145/3324884.3416585

[41] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, Seoul South Korea, 53–56. https://doi.org/10.1145/3377816.3381724

[42] John Wenskovitch, Jian Zhao, Scott Carter, Matthew Cooper, and Chris North. 2019. Albireo: An Interactive Tool for Visually Summarizing Computational Notebook Structure. In *2019 IEEE Visualization in Data Science (VDS)*. IEEE, Vancouver, BC, Canada, 1–10. https://doi.org/10.1109/VDS48975.2019.8973385

[43] Wolfram Research, Inc. 2022. Mathematica. https://www.wolfram.com/mathematica/.

[44] Jo Wood, Alexander Kachkaev, and Jason Dykes. 2018. Design exposition with literate visualization. *IEEE transactions on visualization and computer graphics* 25, 1 (2018), 759–768.