Examining the Strength of the `diff` Command for Patching in a Command Line Interface

Brian Hazzard

June 9, 2015

Advisors: Chris Fernandes, Aaron Cass

Abstract

Frequently used in a command line interface, the `diff` command has been used for nearly 40 years as a file comparison tool [5]. In this study, we test the effectiveness of the `diff` command line tool as a way to preview changes to different actions, or as a patch. We specifically look at the effectiveness in regards to selective undo: a manner of undoing previous actions in a non-linear way. Results found that users who were simply shown what the outcome of their patch would be were able to navigate the selective undo visualizations faster than than those who were shown diff statements.

Contents

List of Figures

List of Tables

1 Introduction

1.1 File Comparison

When editing files on a computer, seeing the difference between certain files clearly is beneficial. Especially when concerning ourselves with revisions of the same file, users wish to know what has changed from revision to revision. There are many different approaches to file comparison tools. The most well known is the use of J. W. Hunt's and M. D. MacIlroy's algorithm[5] to display changes between two files. This approach, commonly known in the UNIX universe as a `diff` command, displays changes to files by noting additions and deletions. There are a few different ways that a `diff` can show differences between files [1], although for the purposes of this study, we will be considering additions to a file to be denoted by a $+$ sign, and deletions to be denoted by a $-$. `diff`'s strength in showing information is more about showing the changes between two files, rather than showing what a final file may look like. So, it is great for file comparisons, and adequate for patching files as well.

1.2 Patching Files

Patching a file means applying a set of changes to an original file to obtain a revised version. Patching is used in UNIX [6] as a means to revise files according to specific changes, as well as in version control systems like Git [4] for the same reason. Usually, these programs implement their own way of creating and applying patches that can be used to revise existing files. Patching can be thought of as a way to apply certain changes to a file without changing the entire thing.

1.3 Rationale for Study

We look to extend previous research [5], [1], [6] that shows diff as an excellent file comparison tool, and to examine the strength of `diff` as a file patching tool. We will use selective undo [3] [2] as an example of how the `diff` command can be used. We hope to determine whether or not `diff` is the most effective way, determined by primarily by speed of completion of tasks, to display file patches in a command line interface.

2 Project Final Design

2.1 Design Requirements

Implementing this experiment as is requires the researcher to be proficient in command line. However, extensions of this project would require a sufficient understanding of the following: command line and bash, python, git, and undo theory. In the program, the git library of python is used extensively, and exposure to their documentation would be helpful if one wishes to make changes to the code.

Volunteers used in this study should be knowledgeable of basic command line processes, and know how to generally navigate file systems using Terminal. If not, the administrator of this study should adequately explain the workings of command line so that the volunteers understand how to use these commands without issue. Testing area for this experiment should be quiet and without distraction.

2.2 Brief Overview

The final design of the program that integrates selective undo is not the same as the one being used in testing. The final program does not have integration with the visualization, as attempts to combine the visualization and the program have been unsuccessful, mainly due to the limitations of the curses python library. So, for the experiment design, each question is asked separately with a visualization accompanying it. This visualization is hard coded to show correct user input, even if the participant has not input the commands correctly. This is so the user can still be tested on problems relating to undo even if they are having trouble with command line in general.

2.3 Implementation Details

The program that we are implementing is a wrapper for command line using the Python coding language. The wrapper accepts certain simple commands listed in the glossary of Appendix A. User input is collected and stored via git to a local repository. Most of the wrapper is implemented using the GitPython library [7] as a necessary way of gathering and storing information into a local repository. GitPython takes git commands such as `git add` or `git commit`, and introduces a way to use these commands in an accessible

way, usually by making objects out of things that we need to access (like commits or the local repository).

The repository object (repo) in GitPython can do many commands that a user would expect to be able to do through command line. For example, the repo takes in the path to the current Git repository, and provides an index which follows and logs tracked files through the use of `repo.index.add(file_name)`. Further, the repo object is responsible for recognizing common commands and using the Git version of these commands so that the files are tracked by git. See Figure 1 to see the implementation. Note that when checking what the command is, there is a return value. This will return `True` if this is an action that should be committed to our repository and `False` otherwise. Some examples of commands that would return false are: `ls` and `cd`.

When there is something committed to the local repository, the repo object adds it to a list. This is a list of **all** commits that have been done in the local repository can be accessed with `repo.iter.commits`, and can be limited to a smaller number of more recent commits using the `max_count` parameter (so the user doesn't accidentally revert to the original commit of the repository. `repo.iter.commits` returns a list of the commits each identified by its unique SHA hash key, with the most recent commits being first.

Undo is implemented in this project through the use of `git checkout` functionality. `git checkout` is a way to navigate between different commits and `branches`, allowing a user to revert to a different state if they wish. The `branching` system in git is worth investigating for other projects, but we won't go into that as our project basically ignores `branching` and works as though it didn't exist. In GitPython, we use `git checkout` by first iterating through the repository and getting a list of all commits. Then, we ask the user which element they would like to undo. Whatever they select, the program will undo every action up to **and including** the action that they specified. Since the commits are ordered from most recent at the beginning to oldest at the end, we simply fetch what the user specified in our list of commits, and call the built in checkout command `repo.git.checkout`.

Git patching is a fantastic way to differentiate between commits. A patch in Git is simply a `diff` file between two commits. As you can see in Figure 2a, we made a change from one commit to another by editing `myfile` to add "hello world" which is denoted by $+$ signs; deletions are similarly marked by $-$ signs. It is easy to make patches across several commits: the notation for making a patch is `git format-patch`.

Our current state is noted by `HEAD` in Git, and since we will always want to be editing from where we are, we specify: `git format−patch HEAD~` and then the number of steps back that we would like to selectively undo. In the example provided by Figure 2c, we would be performing a patch for the state 4 to state 3, 3 to 2, and 2 to 1. This can be seen in Figure 2b. With these patch files, we simply call `git apply` "patch−name" to apply them. So when selectively undoing, we must first use `git checkout` to the state the user wants to selectively undo, and then simply call `git apply` for the states besides the one we're selectively undoing, since the patch file will only change what it has inside of it. If there is a change to a file that we selectively undo, we catch the error and do nothing instead. Afterwards, we delete the patch files. However it may be useful in the future to save them to a secret directory instead in case we wish to use them.

2.4 The Visualization

The visualization for selective undo is not currently implemented, as focus has shifted more to examining the strength of `diff` as a way of conveying information rather than implementing selective undo in our program. However, there is a very high fidelity prototype used for testing these visualizations that can be accessed for one undo before closing.

2.4.1 Visualization Goals

Before we decide on how we wish to show changes in files (using `diff` or not), we decided to use a type of visualization to show cascading selective undo. In our approach to this problem, we look to create an easy to navigate interface that an intermediate user would be able to understand. So, we model our interface after one that users should already be familiar with: a history list. Our goal is that by keeping similarities between a task that users are already familiar with, like browsing a history list, the users will be able to quickly understand exactly what actions are being taken by undoing certain commands. An example of what the interface would look like is Figure 3. As you can see, when the user types in "undo", there is an arrow which points to a command that a user can undo. The commands that have two asterisks appear next to it are going to be affected in some way by this undo, or follow the cascade. We're hoping that suggesting

that something will happen to the actions with an asterisk will be enough information for the user to realize that more than one change may be occurring at the same time.

Once the user has chosen what they wish to undo, they are shown the resulting directory structure after the undo has changed (Figure 4). Again, the arrow represents which file that the user is currently ”selecting” and the user can see more details about the file by hitting the right arrow key. Similarly, the asterisks indicate change in the file they surround.

Our study is attempting to figure out whether or not the `diff` style of display is a good option for showing changes in a text file. Examples of the `diff` and final result visualizations are found at Figures 5a and 5b respectively.

The main argument in favor of the `diff` approach is that it is a more comprehensive description of what is changing in the file system. It displays what is added (using $+$ symbols) or discarded (using $-$ symbols) and could therefore lead to a better understanding of what changes in files. However, a non-`diff` approach does not require the user to mentally figure out what is changing in a file; it simply shows the user what the file will look like. In our experiment, we look to isolate these two ways of showing data and try to figure out which is more efficient.

3 Experiment Design

In this study, we measured the effectiveness of the `diff` style of visualization. To do this, we implemented a visualization for using selective undo in command line as this is a topic that our subjects are likely not well versed in. Even if a participant is more familiar with command line than others, we hope to make it equally as challenging for that user by using selective undo as a pretense. All participants were students of Union College, aged 18-22. The demographic consisted of both students with a Computer Science backgrounds and those without one. 16 total students were tested. Before we discuss the methods, it's important to understand the foundation of the experiment: selective undo.

3.1 Selective Undo

Selective undo works well with patching. Selective undo allows a user to undo certain actions that they have taken without simply undoing everything in their history list. Normally, when a user wishes to undo one of the first actions that they've done, they must undo all subsequent actions before undoing the first one. With selective undo, the user could undo the first action that they did, and only those that are dependent on that action will be affected. Having actions besides the one you undo be affected by your undo is called cascading selective undo. Take the following example:

1. User creates a file called fileA
2. User edits fileA
3. User creates a file called fileB
4. User copies fileA. Name of the copy is fileC

In this example, if the user wishes to undo action 2, the edit of fileA, it is clear that action 3 should not be affected, as fileB is a completely different file. However, action 4 is dependent on action 2 as it copies the contents of fileA. Thus, if action 2 is undone it will also affect action 4.

There has been work done in the field of cascading selective undo specifically by Cass and Fernandes [3]. However, Berlage was the first to propose such an idea [2], and Cass and Fernandes expanded upon it. In

our study, we elaborate further into more specific examples than what Cass and Fernandes proposed so that we can use selective undo alongside `diff`, and gather results based on timing data and comprehension.

3.2 Measurements

To measure the effectiveness of the visualizations being tested, the following measurements were taken: timing data, number of clicks from screen to screen, and a questionnaire after each scenario. The questionnaire is included as Appendix A in the same folder as this document.

The timing data was taken only for navigating the visualizations, not when the user was entering the actual commands in our Command Line Interface. Timing data is taken with the idea that the faster a user would navigate through the navigation, the better they understand it. Similarly, the visualizations were built in such a way that it is easy to track how many times the user switches from screen to screen. Because we are testing how well `diff` shows data to a user, an extension of that is how many times a user has to check the details of different files. The last major device we used to gather data from the users is a short questionnaire after each selective undo scenario. These questions mostly ask questions about content of the file system after the undo has occurred which sees how well the user understood and retained what the visualization displayed.

Questionnaires were given out after each question to assess comprehension in the selective undo tasks. Their comprehension should be at least partially linked with how well either the `diff` or non `diff` approaches displayed information in the file. Average confidence level across the scenarios and questions about the scenarios will be recorded as well.

3.3 Procedure

Participants were randomly divided into two groups ahead of time, either the `diff` or `non diff` group. The two groups were read the same instructions, put in the same scenarios, and asked the same questions. The only difference between the two groups was how the changes in a file were displayed.

Participants were walked through a tutorial in the ”Intro” section of Appendix A, and as instructed, they would type in the commands that were listed there. By physically typing in the commands rather than

just reading from a list of commands, we hoped that the participants would be able to better understand the selective undoing process. After they input the given commands, the users are asked to selectively undo an action using a provided visualization tool. Both the `diff` and the non `diff` groups are shown the same command list shown in Figure 3 as well as the same file structure in Figure 4. However, there was a difference in what the users are shown the details of individual files: the `diff` group was shown the changes in a file similar to what you would see in a `diff` command (Figure 5a). This approach showed what would change in that file after an undo occurred. The non `diff` group was shown what the file would look like after the undo without showing the changes that will get them there (Figure 5b). After going through the visualization and attempting to understanding selective undo better, the participants were asked two questions about the content of the file system after they undid the action.

Following this, the participants were given a bit more freedom in a scenario given to them. They were to input commands listed on one of the scenario sheets, and asked to undo afterwards. They were still separated in the same groups of `diff` and non `diff` file displays. After navigating the undo visualization, they were again asked a few questions on content of the filesystem after the edit, as well as to rate their comprehension of the selective undo process in that situation on a scale of 1 to 5. This continued for four total scenarios and afterwards the participants were asked a few general questions relating to previous experience using command line and undo tools.

4 Results

In Figures 6 through 9 it's clear that the non `diff` approach takes both less time and less clicks than `diff` does. In all cases, the average time to complete a scenario is less for non `diff` subjects. Only in scenario 3 does `diff` beat out non `diff`: the average number of clicks is less than a click greater in the non `diff` case (27.25 for non `diff` and 26.5 for `diff`). Strictly on a speed basis, it is safe to say that the non `diff` approach leads to faster time using the visualization.

All users who answered questions about the scenarios well (either 3 out of 4 or 4 out of 4) indicated confidence levels of 3.75 or more which is surprisingly lower than the average confidence level for everyone of 3.9. This means that there must be several situations where a user thought they were correct, but were

NonDiff Timing	Diff Timing	NonDiff Clicks	Diff Clicks
13.58	29.50	14	10
18.98	34.03	15	16
56.72	41.52	17	17
58.65	55.51	17	20
62.19	61.48	18	23
65.20	70.07	19	25
67.21	71.00	35	25
70.40	88.23	60	50

Table 1: Scenario 1

NonDiff Timing	Diff Timing	NonDiff Clicks	Diff Clicks
19.15	27.36	9	12
31.66	41.86	11	18
32.36	58.90	12	27
49.92	68.92	18	28
53.85	73.47	22	30
61.63	74.44	26	32
63.68	80.60	28	50
142.01		60	

Table 2: Scenario 2

NonDiff Timing	Diff Timing	NonDiff Clicks	Diff Clicks
15.74	28.05	9	11
27.57	32.93	16	11
34.87	34.48	17	21
39.15	45.73	21	23
46.10	59.45	35	33
50.60	79.53	37	34
71.71	81.49	37	39
80.81	91.05	46	40

Table 3: Scenario 3

actually misled either by their visualization or understanding of selective undo. There was only one subject that both indicated an average confidence level of 5 (the max) and also answered all questions correctly. Overall, users were more confident in the actions than not in the non `diff` approach, 6 out of 8 non `diff` gave an average confidence level of 4 or higher in comparison to the `diff` users, where only 3 out of 8 users indicated an average confidence level of 4 or higher. Full results are displayed in table **??**.

NonDiff Timing	Diff Timing	NonDiff Clicks	Diff Clicks
10.41	15.40	7	8
12.26	18.34	12	8
15.79	19.02	12	11
18.60	24.49	12	12
31.03	25.02	13	13
33.69	25.55	14	13
36.55	42.85	16	14
38.39	95.84	21	15

Table 4: Scenario 4

Type of Visualization	Average Confidence Level	Questions Answered Correctly
diff	3.75	4
diff	4	4
diff	4	4
diff	3.75	3
diff	4.5	3
diff	3	2
diff	3.5	2
diff	2.75	1
non-diff	4	4
non-diff	4.25	4
non-diff	5	4
non-diff	3.75	3
non-diff	4.5	2
non-diff	3.5	2
non-diff	4.25	1
non-diff	4.5	1

Table 5: Average Confidence Level and Correctness

5 Conclusions

`diff` has been used frequently as a way to compare two files against each other [5] for a few decades, and is also integrated with the UNIX patch command. By examining the strength as a patch tool in environments where users aren't too familiar with the tools that they're using we can isolate the `diff` visualization as a variable and test it against a control (non `diff` presentation of data).

We hypothesized that users would be able to more quickly navigate our visualization using the non `diff` approach as the `diff` approach requires the user to figure out for themselves what the final file will look like. Conversely, the non `diff` approach simply shows the user what the file will look like after the

undo. The hypothesis was supported by our data where we measured superiority to be a faster completion time of the tasks and less clicks from screen to screen.

Based on the empirical data that was collected, time and number of clicks, it appears that a non `diff` approach is superior to a `diff` approach when previewing undo commands. The non `diff` approach had users that were both faster and required less number of clicks on average (Tables 1 through 4). Since the `diff` approach requires the user to take extra time to figure out what is being added and subtracted to any of the files, it makes sense that on average a non `diff` approach is faster. However, we're not sure that this implies that a non `diff` approach is better than a `diff` approach for previewing the consequences of actions, it simply means it is done faster.

It's important to notice that in our results, there were 2 users who answered only questions about 1 scenario correctly yet indicated average confidence levels of 4.25 and 4.5 (Figure **??**). These users both used the non `diff` visualization. So, although these users worked faster than their counterparts who used the `diff` approach, they had a lower than average amount of questions answered correctly indicating less of an understanding of the experiment material.

It's important to realize that `diff` is primarily used as a comparison tool between two files, and we attempted to extend that to seeing differences in actions. Since participants were only tested on final results, it makes sense that they could better recollect data that was plainly displayed to them (the non `diff` format) rather than having to remember what was added or subtracted to a file.

5.1 Implications

Based on our results, it appears that our users are able to navigate visualizations more quickly when showed data that is not in a `diff` format. Therefore, our work can be extended to other diff implementations: if a program views how quickly a user can interact with it as a strength, then it would be better to use a non `diff` approach.

5.2 Limitations of the Study

One reliability threat is that there was an error in collecting one participant's data (Scenario 2, `diff` user). The participant went ahead of the directions without asking questions and entered a command that caused the program to not write data correctly. Unfortunately, the data was not retrievable and we could not ask the user to do the question again as it would not give accurate results a second time through.

As previously mentioned, one major concern of the study was figuring out how to evaluate a visualization well. After analyzing the results, it has become clear that these results should be accepted cautiously. We are not convinced that time and number of clicks are a good measurement of a visualization, as users would often signal that they are done with a task before completely checking the file system, they would get an answer wrong in one of the questionnaires, or they would give themselves a medium/poor comprehension ranking.

Another reliability threat that may have skewed the results was that it was clear that some users thought about what they would undo before entering the visualization (where the timing starts) while others did not.

Finally, our questions only asked about the final state of the file system after we undid a command. This could definitely skew results in favor of the non `diff` approach which simply states what the final state is. The benefits of displaying a `diff` style visualization are mostly to help with comprehension in what is changing in the moment rather than what the file will look like after. We believe that by asking more questions that had to do with comprehension of selective undo or about a specific change, we would get better results for the `diff` approach because users would have to think more about the process of the undo and less about the final result.

5.3 Future Work

One extension of our work is to decide on an alternative method of data collection. Using selective undo to evaluate the strength of the `diff` command helps in that it introduces a medium that most have never heard of so there is no advantage to one participant over another. However, if a user has a lot of trouble understanding the core concepts of selective undo, it is easy to see how their data might reflect more on

their confusion with the testing technique rather than reflect the strength of the visualization.

Similarly, time and number of clicks have shown to be not the most effective means of gathering data. More research should be done into figuring out the best way to gather data from a visualization. Different questions in the questionnaire would be helpful, as currently they only asses how well the user remembers what the final state of their file system will be after the undo.

Another extension would be to look at using diff in text-based applications such as Microsoft Word which should share qualities with our text-based study here, even if terminal navigation isn't the same as text editing.

5.4 Conclusion

Although the results of our experiment may not be truly indicative of whether or not a `diff` visualization is stronger, results clearly show that in all scenarios asked, participants were faster, on average, while navigating non `diff` environments. Further research is necessary to definitively conclude that our results are valid, but this is definitely inspiration to continue this line of research.

References

[1] Thomas Ball and Stephen G Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[2] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 1(3):269–294, 1994.

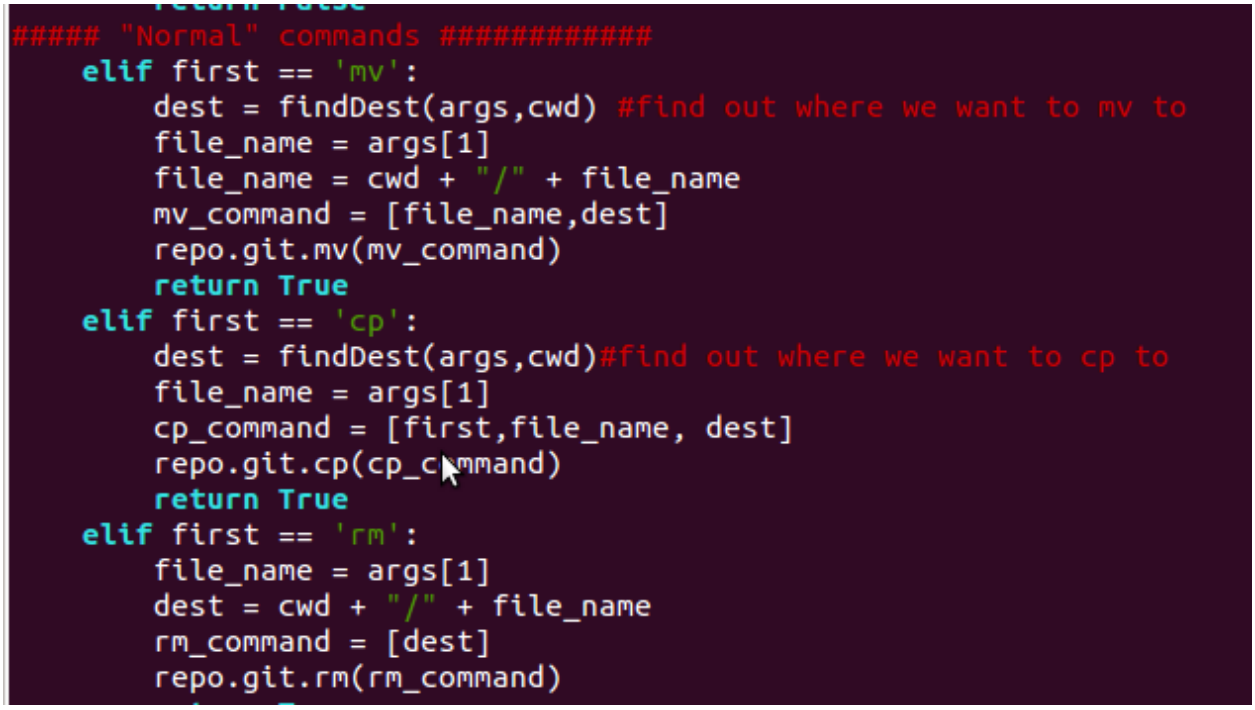
[3] Aaron G. Cass, Chris ST Fernandes, and Andrew Polidore. An empirical evaluation of undo mechanisms. In *Proceedings of the 4th Nordic conference on Human-computer interaction: changing roles*, pages 19–27. ACM, 2006.

[4] Git. http://git-scm.com/.

[5] James Wayne Hunt and MD MacIlroy. *An algorithm for differential file comparison*. Bell Laboratories, 1976.

[6] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[7] Sebastian Thiel. https://github.com/gitpython-developers/gitpython, 2009.



```
##### "Normal" commands ############
    elif first == 'mv':
        dest = findDest(args,cwd) #find out where we want to mv to
        file_name = args[1]
        file_name = cwd + "/" + file_name
        mv_command = [file_name,dest]
        repo.git.mv(mv_command)
        return True
    elif first == 'cp':
        dest = findDest(args,cwd)#find out where we want to cp to
        file_name = args[1]
        cp_command = [first,file_name, dest]
        repo.git.cp(cp_command)
        return True
    elif first == 'rm':
        file_name = args[1]
        dest = cwd + "/" + file_name
        rm_command = [dest]
        repo.git.rm(rm_command)
```

Figure 1: Example of the Repository implementation

```
brian@brian-VirtualBox:~/thesisTesting$ cat 0001-edit-myfile.patch
From a886302e59f1f882430755d028dbf57b8fdceea4 Mon Sep 17 00:00:00 2001
From: Brian Hazzard <hazzardr@union.edu>
Date: Thu, 19 Mar 2015 03:13:23 -0400
Subject: [PATCH 1/3] edit myfile

---
 myfile |    1 +
 1 file changed, 1 insertion(+)

diff --git a/myfile b/myfile
index e69de29..3b18e51 100644
--- a/myfile
+++ b/myfile
@@ -0,0 +1 @@
+hello world
--
1.7.9.5

brian@brian-VirtualBox:~/thesisTesting$
```

(a) What `format-patch` produces

```
/home/brian/thesisTesting >> undos
1)  "rm otherfile"
2)  "mv myfile renamedfile"
3)  "edit myfile"
4)  "touch otherfile"
5)  "touch myfile"
Please type in the number of the command that you would like to undo selectively: 3
0001-edit-myfile.patch
0002-mv-myfile-renamedfile.patch
0003-rm-otherfile.patch
/home/brian/thesisTesting >> ls
0001-edit-myfile.patch  0002-mv-myfile-renamedfile.patch  0003-rm-otherfile.patch  first_file  myfile  otherfile
```

(b) `format-patch` creating multiple patches

```
/home/brian/thesisTesting >> undos
1)  "rm otherfile"
2)  "mv myfile renamedfile"
3)  "edit myfile"
4)  "touch otherfile"
5)  "touch myfile"
Please type in the number of the command that you would like to undo selectively: 3
```

(c) Example of selective undo implementation

Figure 2: Selective Undo

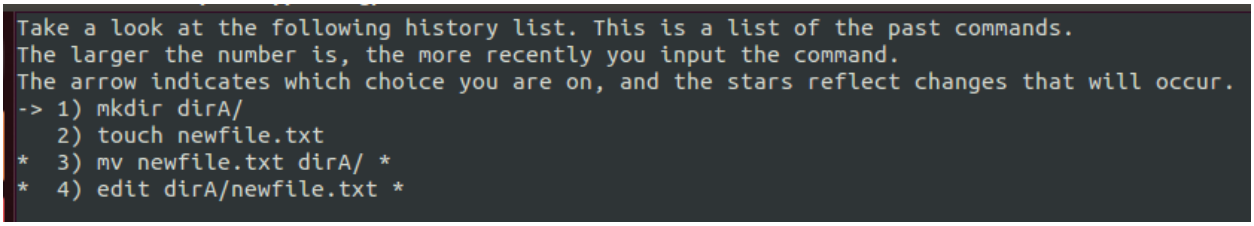


Figure 3: The User is Attempting to Undo Command 1

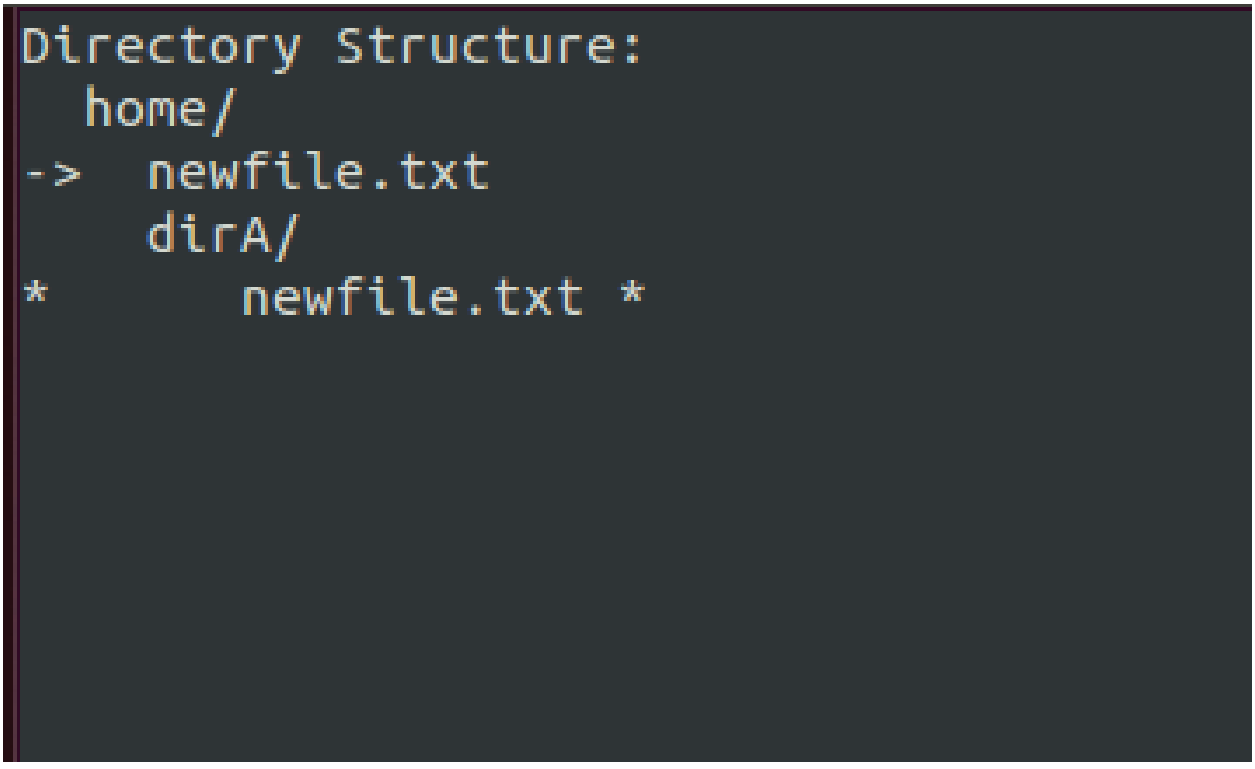
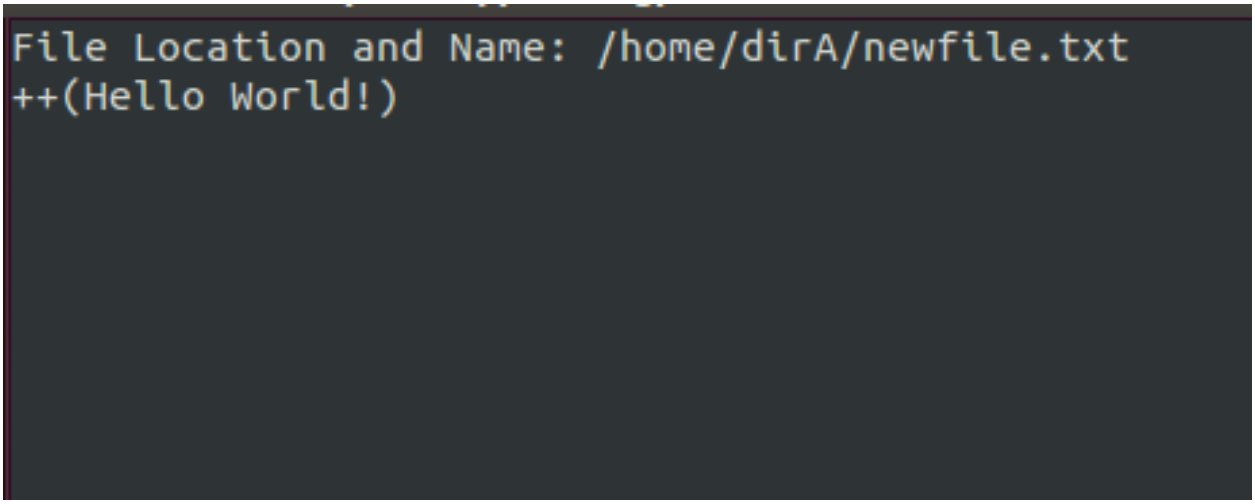
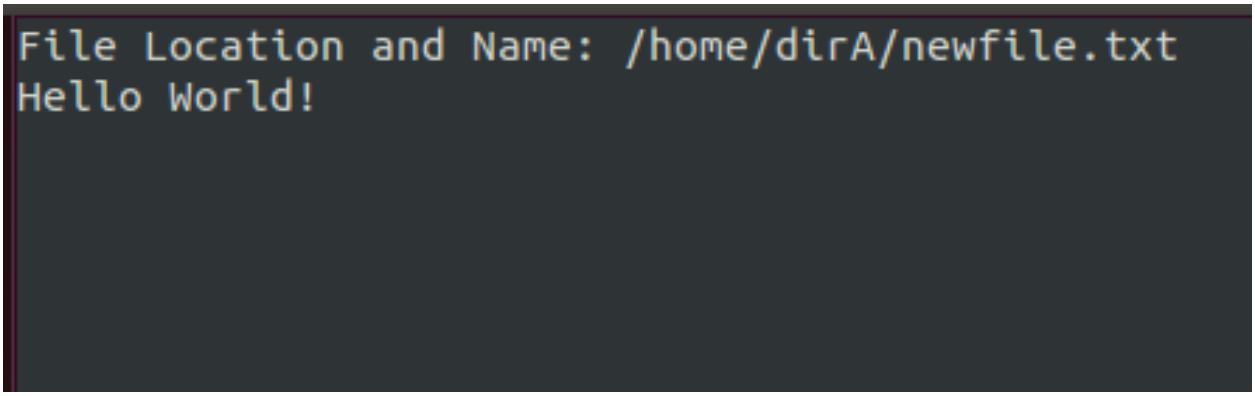


Figure 4: The Directory Structure After an Undo



(a) `diff` Showing Differences in a File



(b) Showing the Final Result After an Undo has Occurred

Figure 5: Different Visualization Methods

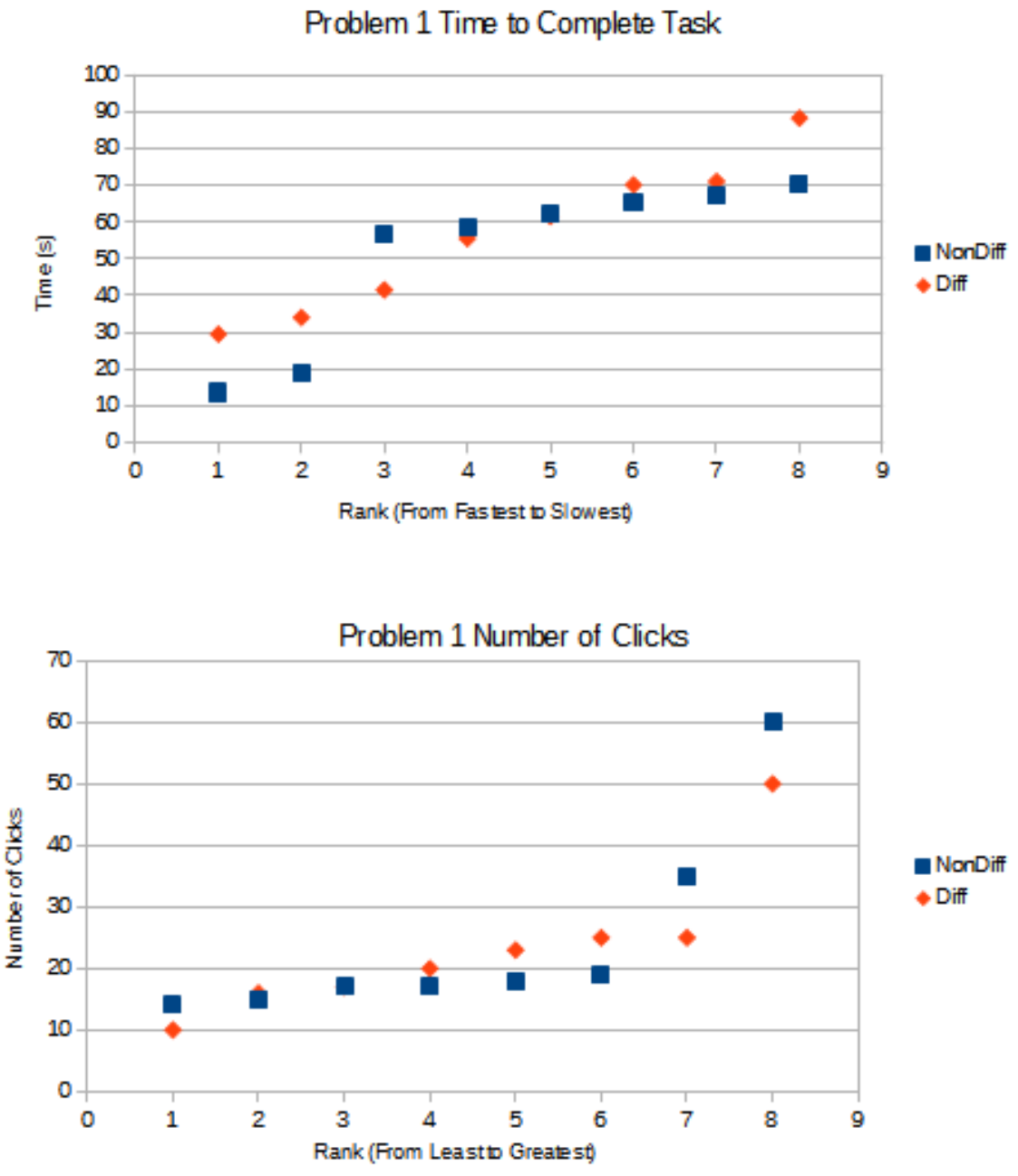


Figure 6: Data Collected for Scenario 1

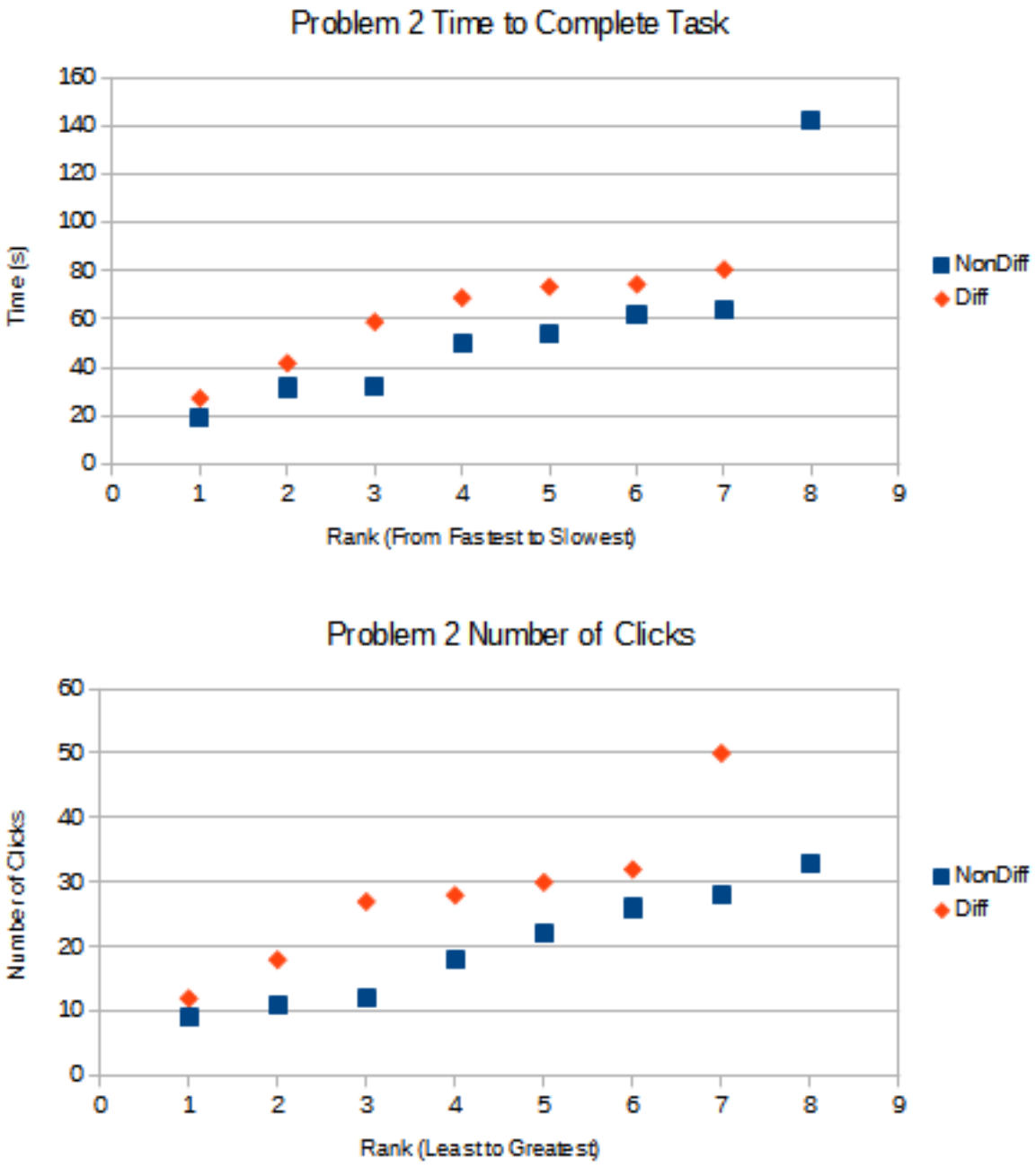


Figure 7: Data Collected for Scenario 2

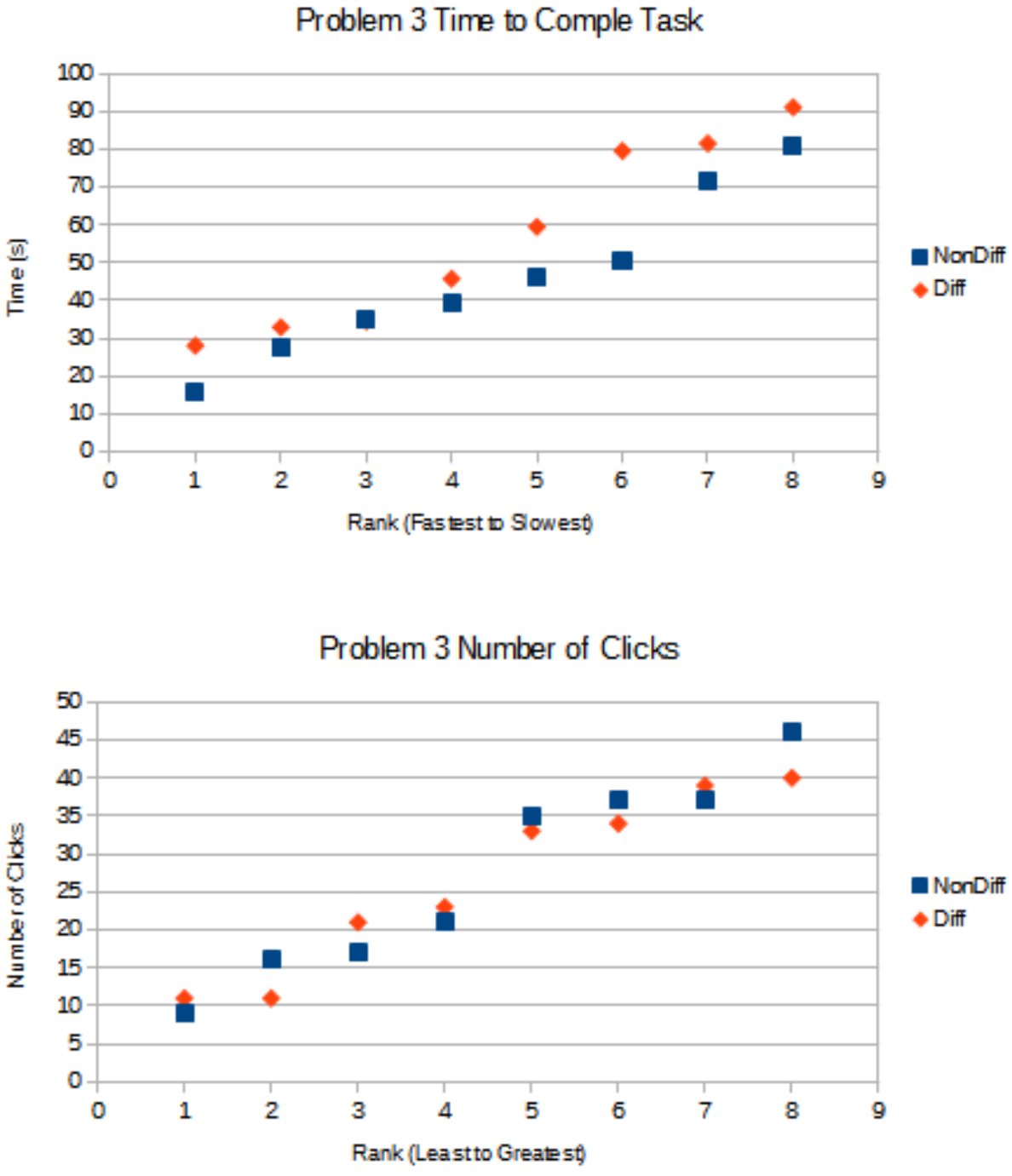
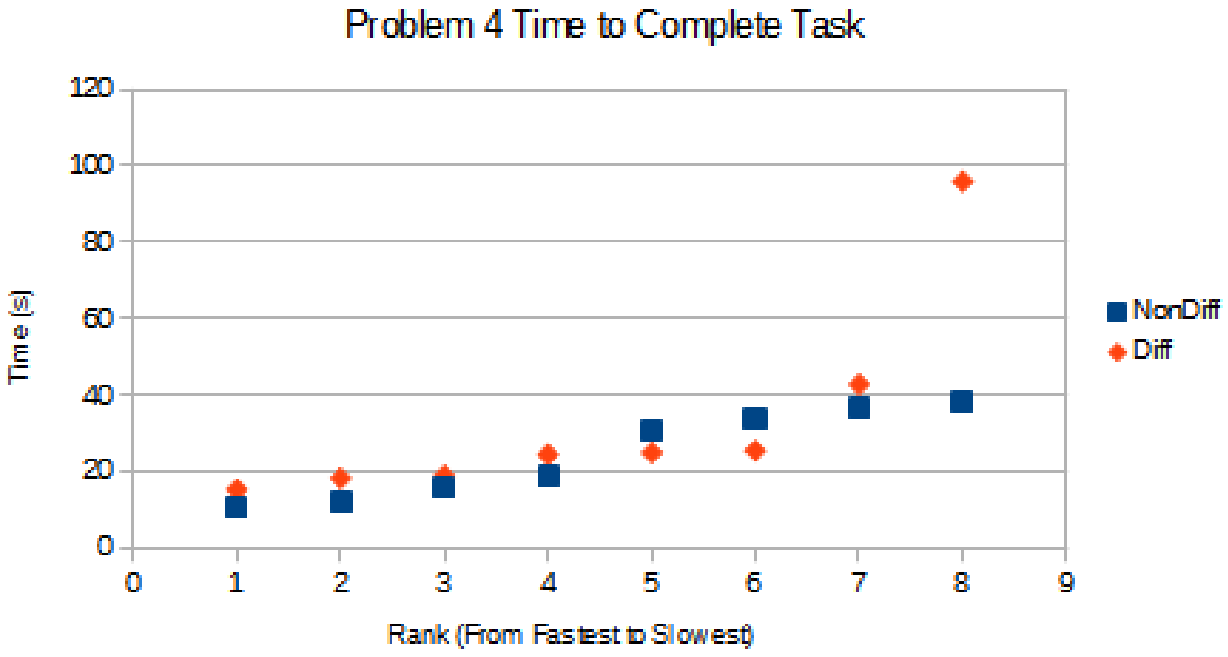


Figure 8: Data Collected for Scenario 3



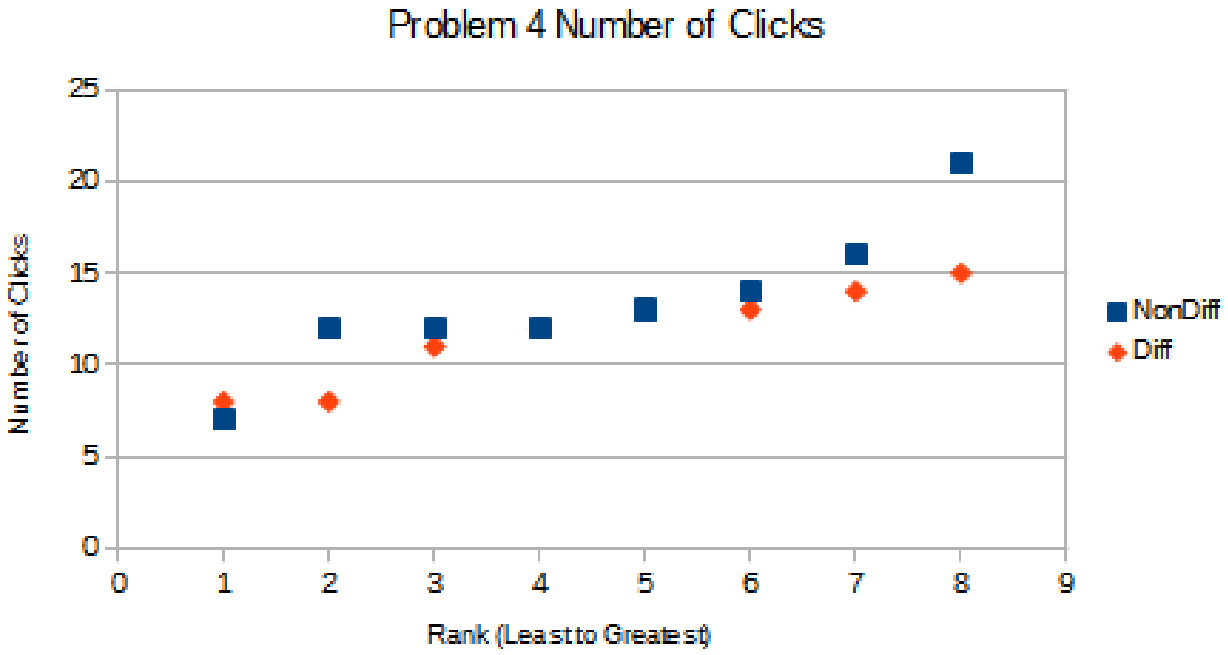


Figure 9: Data Collected for Scenario 4