Evan Schallerer
CS362 Test Report

Part I
Classmate1: alzamilb

The original dominion implementation was far from ideal, containing numerous bugs and lacking in any real completeness. Ideally, the way to fix this would have been to start over, using the knowledge learned from making the first one as a starting point and reference. The documentation was little to none, and on top of this, many of the functions were incorrect, did not interface together well, and did not lend themselves to testing or running the program in an effective manner.

Fixing this dominion implementation is a daunting task to be placed on anyone, and credit must be given to anyone who takes on this task. In alzamilb's version of the program, it is quite similar to the original as there were no requirements to actually fix the original code. This person has, however, fixed some of the original bugs in the program. On top of this, there are also some new bugs, such as in the smithy card.

Before this code could be considered reliable it would need a lot of clean-up and bug fixes. It does pass some of the tests that are run, but many of them are failing. The tests are, of course, not comprehensive for a number of reasons. First, there was not enough time nor resources to build a full test suite that would adequately find all of the bugs in dominion and produce enough information to find them. Secondly, many of the bugs are very subtle and would require essentially rewriting dominion to test for them. On top of this, the documentation for the code is very light and does not allow someone to test it effectively without access to the author or better documentation. This was a big issue when testing the code.

Because dominion is completely card based, most of the tests focused on playing the cards in the code. One large piece that detracts from the reliability and usability of this dominion implementation is the presence of the large switch statement that handles playing the cards and determining what they do. To have reliable code, it is technically possible for this to be effective, but it is not maintainable, and the chances that it will work perfectly are incredibly low. As we can see through test results, it does not work very well. Refactoring the code, as some was done in this implementation helps to clean it up and find bugs and other problems, but this is a difficult task.

For the most part, this dominion was fairly good at not having serious issues that cause it to crash, which adds to the reliability aspect. The planning may not have been sufficient to get all of the pieces working together in an easy to understand manner, but the fact that most of the tested code will run without any sort of serious failures is definitely a good thing. The tests that were written for dominion covered close to seventy five percent of dominion.c, which seems respectable. This aside, there is a lot more that could be done on the testing front, as well as a lot more that would need to be done, before this code could be considered fully tested and ready for reparations. The code coverage results were:

File 'dominion.c'
Lines executed:74.95% of 579
dominion.c:creating 'dominion.c.gcov'

Part II
Classmate: luans

Similar to the last implementation of dominion that was evaluated, this code is not much different than the previous implementations of dominion. It has many bugs in it still, and needs a lot of work before it could be close to being considered ready for any sort of production. Also similarly to the previous version, it would be ideal to rewrite dominion using this as a starting point, rather than trying to fix it to make it functional. Most likely, it would be less work to go this route than to make this code into a functional production piece.

There were not any glaring issues in terms of crashing that were found during testing this dominion. It seemed to handle memory well and not cause segmentation faults, but there were plenty of bugs in the program that caused it to not behave properly when handling cards. Because dominion deals solely in cards, this means that it does not play properly and is therefore not functional. Proper planning and setup could have prevented a lot of the problems in this code, such as the very large switch statement that determines the behavior of each card. The lack of documentation also does not make it easy to fix and refactor this code into a functional program.

The testing for this program was also the same as the last one, and was not in any way a complete test suite. Each card should be thoroughly tested with a number of different cases, which would be very time consuming, and not worth it for a program that is in such need of repair such as this one. Many of the tests fail, but there are a few that pass that did not pass in the original dominion implementation. The tests achieved a coverage of seventy two percent, which is slightly lower than the last implementation that was evaluated. We can see this here in the GCOV output:

File 'dominion.c'
Lines executed:72.69% of 575
dominion.c:creating 'dominion.c.gcov'

This is a little low to find anywhere near all of the bugs in dominion, but it is enough to determine that there are problems in the code that need to be dealt with. A good step towards making this code usable would be to go through and follow a consistent coding convention, as well as adding documentation for each function. This would make refactoring easier, it would make combining each piece to work together easier and more effective, and it would make maintaining the code much less prone to errors. I do not think that any use of the original dominion code would be justified due to the amount of work that it would require. Even with a substantial amount of work in refactoring and testing, this code would not be suitable to implement as an actual game. It would be less work to start from scratch and use the lessons learned building the original dominion, learning from what worked well and what did not in this version of dominion.