# Test Report

Overall, testing the given Dominion implementation has been both frustrating and enlightening. I have never really done any sort of automated testing before this class, so everything was new to me. At first, I didn't even really see how testing was even supposed to work, since it seems like you would either have to write manual input/output maps (which would be tedious and narrow-reaching), or for automated testing, basically reimplement the code you're testing for the test itself. I now realize though that in many situations, tests can actually work at a higher level than the code being tested, and it can in fact be easier to write a test for a function than the function itself; there is often no need to reimplement it completely.

Additionally, I was introduced to various new debugging techniques, such as using functions like "print" and "step" for gdb, or using gcov to implement Tarantula-like processing to identify suspicious lines of code. The various testing techniques are quite novel as well; creating a state, copying it using memcpy, and performing on operation on one of them so that they can be later compared is a genius idea that had never crossed my mind before. Another one of these ideas that I had not considered heavily was the use of seeds for producing the random numbers needed for the game; I would usually seed with the current time before this class. Now I realize how useful it can be for debugging purposes to not only be to able to reproduce the error on demand (as in Agans' Rule #2, "Make it fail"), but also to be able to test a potential bug fix in the same exact situation as where it had previously failed to ensure that the bug had indeed been fixed (as in Agans' Rule #9, "If you didn't fix it, it ain't fixed").

The frustration, however, lies largely in the quality of the codebase. I am aware that the given Dominion implementation is not meant to be anywhere near perfect or even good, but still it bothers me how little sense the codebase seems to make, with not only poor formatting, but also poor architecture and even downright poor code in many places (for example, unused variables, unnecessary control statements, duplicate code–the list goes on and on). What eased this frustration a bit though were some of the premade tests, particularly playdom.c, which I was able to use the basics of to form the basis of my own testing. This was particularly helpful because I had never played or even heard of Dominion prior to this class, so I was able to not only glean some of the game's flow through the game play implementations, but also get a feel as to how a test is supposed to be laid out in general. Also useful was the state initializing code, which is a little mind-blowing to me; I found the entire concept of randomly generating a program state and running with it to be entirely outlandish. Even now, the idea still feels a little overwhelming to me, but I was at least able to implement it far enough to create my own testdominion.c full game tester (as well as some of the other tests, such as randomtestcard and randomtestadventurer).

In the end, I think I not only improved my C, Python, and general programming skills, but also definitely my problem solving, testing, and debugging skills. This progression is apparent even over just this term by looking at what I turned in for each of my assignments. The first tests were ugly, narrow, and broken, but as time progressed, I learned and grew, leading to now what I consider to be one of my best unit tests so far, minion_tarantula.c; not only is it the most elegant one I have so far, but it's also the most reliable and complex one. I was even able to get it working with my tarantula.py to implement Tarantula and further my skills in fault localization and bug isolation.

As for the reliability of my classmates' Dominion code bases, I will examine those of Kevin Bergman and Audrey Sullivan.

## Kevin Bergman (bergmank)

Kevin's code base is largely similar to mine, and like it, is also far from perfect. His code, however, does seem significantly more reliable than mine; for example, most of my unit tests fail with errors (likely due to myself writing the tests badly), and my testdominion.c full game tester sometimes gets stuck in infinite loops with certain random seed inputs, even after already fixing one of those types of bugs earlier in this assignment. His testdominion.c implementation, however, does not take a seed as a command line argument, but rather randomly generates numbers differently every execution, preventing the same situation from being tested more than once, even if we need to test a bugfix with values that we knew previously caused the bug to surface, for example. In terms of reliability, his is clearly superior, but in terms of coverage of dominion.c, his tops out at about 45 percent after trying 10 different seeds, while my implementation of testdominion.c reaches around 67 percent coverage of dominion.c with the same 10 seeds.

## Audrey Sullivan (sullivaa)

As with Kevin's, our code bases are largely similar, and Audrey also has more reliable unit tests than me. With regards to the random testers, randomtestcard and randomtestadventurer, however, my first impression is that mine are superior as hers require constant feeding of new seeds, while mine run automatically whilst still being plenty random. Additionally, hers make use of assert statements, while mine tally failed tests so that they may keep running even after one fails, allowing a full, reliable run of the test suite. However, her testdominion.c was more reliable than mine as well, given that it didn't seem to crash, freeze, or loop infinitely at all, unlike mine. However, she also lacks in dominion.c coverage compared to my implementation of testdominion.c; hers seems to top out at about 43 percent after trying 10 different seeds, which is quite low compared to my 67 percent and even slightly lower than Kevin's 45 percent.