Dylan Camus
CS362
Final Report

Final Report

Experience

I did not find dominion to be an enjoyable experience testing and debugging. I found the source code to be so ridden with bugs that it made testing very difficult. For example, there were many times where my random tests would get stuck in infinite loops that made it difficult to properly get gcov information or do diffs on files. However, I do believe that this struggle has prepared me for the challenges I may face in my future career. Not every bug is as simple as a number being incorrect. Many of them are much more subtle. If it were not for this class I may never have learned the skills needed in order to find and fix these edge case type bugs.

I learned many important skills from this class. First, I learned that print statements for testing are not bad, as I had always been told. I found print statements to be invaluable when trying to find where in a test program I am getting stuck. I also learned that brute force testing is an effective strategy. Many times a bug can be hidden deep in code that is rarely executed. By running a huge number of tests, it is much more likely that that bug will show itself. Another important skill I learned in this class was how useful gcov was. Being able to visually see which lines of code are not being executed is really helpful when looking for bugs and cleaning up code.

Taking this class has inspired me to work on a summer project. I have always loved dominion, and I want to create my own implementation of dominion. Testing the dominion code provided in class was frustrating because of how poorly it was written. I plan to write my dominion code in a much cleaner fashion. This will be interesting, as the bugs I will find will likely be much less frequent and less obvious. I will need to use the skills I learned about unit tests in order to reduce the frequency of obvious bugs, and random testing in order to find the edge cases.

Code Coverage

I found my unit tests to have rather low code coverage. This makes sense, however, as they each only test one card. Unittest1.c had a code coverage of 18.56%, unittest2.c had coverage of 26.62%, unittest3.c had coverage of 20.32% and unittest4.c had coverage of 20.67%. I believe these numbers good for what they test. Since each only tests one card, there are many functions which are not called. Moving on to the bigger tests, testdominion.c. I found that testdominion.c had code coverage of 71.10%, which was a little low. I would like to go through a few of the places in dominion.c that were not executed.

The function adventurer_effect was never called. This was due to the card causing an infinite loop. While working on testdominion.c I found this bug, and in order to keep the test moving, I simply removed adventurer from the list of kingdom cards that were available to be randomly put in the supply. I found this to be a good solution as I already knew that adventurer_effect was

bugged and I wanted to let the test terminate without getting stuck in a loop. Additionally, there were a few other card_effects that were never executed for the same reasons as adventurer. These were smithy, feast, embargo, and tribute.

Another reason for my low code coverage was useless functions. One function that was never executed was kingdomCards. This function does not serve much purpose and I see no reason to use it. Another function that was never executed was fullDeckCount, which I never called in my test and is not called by any other function in dominion.c. Other functions include scoreFor and getWinners.

There were also a few error cases that never came up, such as the return -1 statements in the initializeGame function. An interesting line that was never executed was line 487 of dominion.c which is the return statement that ends the game if there are no more provinces. This is likely due to the way I wrote testdominion.c, which is not very realistic to a real game. There is no artificial intelligence so the only way that the provinces would run out is by random chance. The games ending 100% of the time due to 3 supply piles being empty.

Looking at the code coverage was very useful to me. I now know that if I wanted to improve my testdominion.c coverage I would need to make calls to the unused functions. I also know that I could increase my coverage by quite a bit if I fixed the infinite loops in the card_effects that cause them I could then include them back into the test. I believe that if I were to make these changes, I could get my code coverage up to at least 95%.

Reliability

Gassa:

I found through running my tests on gassa's code that many of the same bugs as in my code were present. For example, the bug that causes cards that have been played not to enter the discard was still present. However, I found different bugs in the card_effect function than were in mine. For example, the smithy bug that caused an infinite loop in my code was not present in gassa's. Additionally, I found a bug in gardens that gardens were not correctly counting for points. This was due to the gardens function containing no code other than a return -1 statement. I also found a bug in the mine card were the card that was supposed to be trashed was not actually ever put into the trash, but rather just put into play.

Olsenw:

Again, I found many of the same bugs were in olsenw's code as were in mine. This makes sense because the only parts we changed were in card_effect. Running through my tests I was able to identify a bug in remodel that was causing the remodel card_effect to not properly trash the card that was supposed to be trashed.

Overall, I found a number of bugs in both gassa's and olsenw's code, and I would not say that the code is at all reliable. Running testdominion.c with their code and diffing it with my own code was causing huge discrepancies. I ran through a few different seeds and each time I was

seeing upwards of 70% of the lines being different. I believe this is due to the fact that a single bug can change how many cards are drawn, for example, and this will then have an effect on every other action that is made during that test. Once one difference appears, the rest of the test will tend to be different as well.