Matt Huang

CS362

Test Report

Summary

Ove the past few months, I have used multiple unit tests, random testers, and debuggers to test various implementations of Dominion. I also used gcov and mutant testing to confirm that my testers were correct. In this report, you will find the process and the results of this testing.

Process

To start, I applied Agan's principle #1 which is to understand the system. Since I had never played dominion before, it was vitally important to know how the game worked so that I could identify what were bugs and what weren't. To gain knowledge of the system, I ran through a couple of games of dominion until I was familiar with it. I also used the internet to confirm the functionality of some cards if I didn't know them already.

After I understood the system, I used simple unit tests to test various card effects and functions within dominion. This could be considered applying Agan's 2nd and 4th principle which were to "make it fail" and to "divide and conquer" respectively. I also used gcov to test how much of dominion each test was covering. Here are the results of these unit tests.

Results from unit and card tests

UnitTest1

Lines executed: 15.97% of 576

Nearly every line in shuffle was executed some more than others. Specifically the things in the while loop were executed (30-35) times.

Unittest2

Lines executed: 19.27% of 576

Most lines in end turn were covered, however they were only covered 1-5 times at the most.

unittest3

Lines executed: 17.36% of 576

Aside from the loop, most lines are only covered 1-5 times.

unittest4

Lines executed: 21.70% of 576

Most lines of getwinner called but not very many times

cardtest1

Lines executed: 22.57% of 576

feast case is executed, lines called very few times

cardtest2

Lines executed: 20.14% of 576

similar to cardtest1 but with smithy

cardtest3

Lines executed:20.14% of 576

village case called once

cardtest4

Lines executed:20.49% of 576

baron case called once

Bugs found:

An infinite loop occurs when feast is given a card that is too expensive or is out of supply. Feast also does not trash itself.

As you can see, few bugs were found and little of the dominion code was tested. To focus on both issues (but mostly the first), I decided to randomly test a couple card effects. Random testing allows for many different inputs which could hopefully reveal more bugs. I also used mutant testing on my tester to make sure it was actually catching bugs. Here are the results of that testing and some more in-depth summary of how I implemented each tester.

Results from using random tester on cards:

Randomtestadventurer:

I began by trying to create a very random game state. I did this by modifying the hand counts, deck counts, and discard counts of the player playing the card. No other player needed to be varied because the card didn't affect them. Modifying all 3 cards themselves doesn't actually give the player cards, but that's ok. This is the case because every player starts with having enough copper for the card to have its effect. If the random number generated for say the deck count was higher than the amount of cards actually in the deck, it'd just be like giving the deck a bunch of blank cards. On the other hand, if the random number was lower than the actual deck size, then it would account for the cases where the player has few or no treasure cards. The number of treasures in hand, deck, and discard are all being kept track of so that we can compare them later. A new assert was written so that we could continue in the case of a failing case.

Running the test by itself gives this coverage:

Dominion.c

Lines executed: 24.83% of 576

File 'olsendominion.c'

Lines executed: 25.52% of 576

File 'kaiserhdominion.c'

Lines executed:26.82% of 578

Since we are only calling one or two functions, this was expected. The coverage was higher for this than for the unit tests. All lines in cardAdventurer are called many times.

Randomtestcard.c

I choose to test council room which gives the user +4 cards and +1 buy but also gives all other player +1 card. I started the same as the other test, I randomized the game state by modifying the counts. This time however, I modified the deck counts, discard counts, and hand counts of all the players since all of them are being affected by the card effect. I check whether each player has the correct hand count, that the number of buys has increased, and that the council room card has been discarded before I allow the test to pass. I also provide separate cases for if a player can't draw a card due to having too few cards in deck and discard. Again new assert is used to allow it to keep running despite any failing tests.

Running the test gives this coverage:

Dominion.c: File 'olsendominion.c'

Lines executed:25.34% of 584 Lines executed:24.83% of 584

File 'kaiserhdominion.c'

Lines executed:25.26% of 578

This is similar to the other random tester. Lines in council room were execute many many times. Lines in other places such as shuffle and initialize game were also run many times. Coverage difference was negligible.

Bugs found:

When not modifying the dominion code at all dominion.c and olsendominion.c this bug was found when running test adventurer:

handcountbefore: 3

deckcountbefore: 0

discardcountbefore: 31

handcountafter: 1

deckcountafter: 0

discardcountafter: 33

test 65 failed

Instead of gaining 2 cards, the player loses 2 for an unknown reason.

Kaiserhdominion.c however failed every test. Most of the time the total card counts along with hand counts before and after were incorrect.

All of them passed when council room was ran.

This random testing was very good for testing just those 2 cards. However, they took a lot of time to create since you must vary every variable (or most) and you must assert correctly for every situation. This added complexity adds a bigger risk of introducing bugs into your test code which could skew your results if you're not careful. Also, our coverage still wasn't very good (only about a quarter of our dominion code) so we still had to solve that. To solve this issue, I decided to use a more general approach and started testing entire dominion games.

In this dominion game tester, we simply play a random game of dominion by randomizing the number of players, what they buy, and what they play. It's important to note that very few things are asserted so unless a fatal bug (seg fault, infinite loop) occurs in one of the functions, it will be nearly impossible to identify a bug. To alleviate this we use the random tester on 2 different implementations of dominion and then diff the logs (what's printed to the screen ie: player x played this card). Comparing the different outputs allows for us to test functionality without directly asserting the correct result.

Results from testing entire games and diffing the logs:

dominion.c 'olsendominion.c'

Lines executed:55.99% of 584 Lines executed: 57.81% of 576

File 'kaiserhdominion.c'

Lines executed:56.57% of 578

Bugs found:

For many seeds, test dominion goes into an infinite loop caused by feast mostly. There are other cases where it is also caused by embargo.

Many differences were found when comparing to a few different dominion implementations, here are a couple of the bugs determined in those.

Olsendominion: gardens when played, returns a 1 instead of a -1.

Kaiserhdominion: Sea hag reduces deck count of all other players when it should not be. It is true that all other players discard the top card of their deck, but they also gain a curse to replace it meaning the deck count stays the same and only the discard count should increase. This was found using my random tester, 2 out files, and the diff produced by them. In the diff, you can observe the difference after sea_hag was played making it easy to pinpoint where the bug was.

I found this method to be the most effective. Not only did it cover the most code, but it also is a very flexible tester. For example, you can set a bias on it so that each player only plays a certain card allowing for you to more thoroughly test that card. It still has the down side, however, of being at risk for bugs.

You'll notice that the coverages for each of them were different in this test and the test before, this could be accredited to the fact that each program has been edited so the number of lines in each is different. As you can see, the number of lines in each doesn't differ by much, but at the same time, the percentages don't either.

I'll mention that I briefly used delta debugging to minimize a failing test case. Specifically I used it on the feast card effect since it was so prominent. To briefly summarize, the delta debugger isolated that the error was cause by me assigning rcard(the card the is bought or played) to feast. Since I was always buying something that was too expensive for feast, this was to be expected. Delta debugging was alright, but it takes a lot of time to produce a workable good test case to feed it.

Fixing the bugs

Not a ton of time was spent simply fixing bugs since most of it was spent finding them. I'll just say that fixing most of the bugs was a simple case of applying agan's 3rd and 1st principle; "quit thinking and look" and "understand the system". In other words, to fix most of the bugs that I found, all I had to do was look at the source code for the function that the bug was in, understand what the function was supposed to do, and fix it accordingly. This may sound overly vague and general but it was really that simple since most of the functions didn't need to be very complicated.

Conclusion:

Now the question is how reliable I believe my classmates and my implementations of dominion are. Well, personally, I believe that the 3 implementations that I've tested today (kaiserhdominion , olsendominion, and dominion(mine)) all aren't very reliable. In literally all of them, the scoring was inconsistent and in all of them, the infinite loops weren't taken care of. Also, despite playing entire games of dominion, I was only able to get about 56% coverage which is barely above half of the dominion code. So, to conclude, the dominion implementations I've tested are faulty, buggy and not reliable.