Makiah Merritt

March 13, 2016

CS 362 Final Report

Test Report

For this final portion I inspected Steven Powers' and Griffin Gonsalves' dominion.c implementations. I begin by describing the various tests that I have prepared throughout the term. There are four unit and card tests and three random tests all of which were run on each individuals' code. My unit and card tests begin by instantiating a game state that is passed, with other required variables, into a custom function used for performing the actual test. Note, dominion implementations were mainly checked using Visual Studio Community 2015 while the OSU server was used to gather code coverage results. The overview examination of each test should provide a good picture of what code it covers.

Tester Processes

Unit Tests

My first three unit tests average 20-25% code coverage on dominion implementations because the path the test follows is hardcoded. Unit Test 1 tests the dominions playCard function. For this I don't vary which card is being played, beyond simply walking through each card in the players' hand as it was initialized because I want to make sure that the function returns the expected value given invalid and valid circumstances. Of course these circumstances being conditional to the player: whether or not they are in the proper phase, whether or not they have enough actions, whether or not playCard exits because the card referenced by handpost is not an action card, and finally whether or not the player's action count is decremented after a card is successfully played. As might be expected the lack of variation from this process means coverage will be lacking unless the card is one of particular interest.

Beginning with a process similar to Unit Test 1, Unit Test 2 is responsible for testing the dominion implementations updateCoins function. Only needing to test each coin (copper, silver, gold) the overall process of this function is tightly limited as far as code coverage is concerned because there is no deviation from the updateCoins function as there is with the playCard function. As an aside it is surprising code coverages between Unit Tests 1 and 2 are so close, with the latter covering .34% more code. Unit Test 3 performs a test on the dominion implementations fullDeckCount function. During this test it covers calling fullDeckCount with invalid cards and all cards in the CARD enumeration (0 - curse to 26 – treasure map). Finally, the last Unit Test, Unit Test 4, checks the dominion implementations buyCard function. Unit Test 4 offers the greatest coverage of this suite averaging about 33%; its process covers whether or not the player has enough buys, whether or not the desired card has enough quantity in the games supply count, whether or not the player has enough coins, and lastly runs through successful purchases. Because Unit Test 4 tests the dominion buyCard it is able to branch into more of the core functions of dominion: supply counts, checking card costs, and finally gaining cards.

Card Tests

The four card tests run in my test suite test the dominion implementations steward, smithy, great hall, and village cards. Card Test 1 tests the steward card by checking the result after the players choice changes: 1 – player draws two cards, 2 – player gains two coins, or 3 – two cards are trashed from the players hand. Card Test 2 tests to make sure that the player has gained three cards after playing smithy and that smithy is the last card in the players played cards pile. Card Test 3 tests to make sure the player gains a card and action then to make sure that the smithy is in played cards. Lastly, Card Test 4 make sure the player gains a card and two actions before adding village to the players played cards pile.

Random Tests

The random adventurer test varied slightly from what was provided as a base case; the expansion included adding check a check to make sure the last two cards in the players hand are both coins (copper, silver, gold). Not basing it off of a predefined test, the random remodel test is more thorough. First it sets up the game state with random values and places the remodel card somewhere in the players hand. After calling the playCard function the test verifies: the players actions are reduced, choice2's supply was decremented, and a copy of choice2 resides as the last card in the grave yard, and finally that the played card count is incremented by two whilst hand count decremented by two.

Despite using the rand function as it's number generator my random dominion game tester should in theory poses the ability to cover many scenarios that could be found in the dominion game as no choices are pre-determined. However, partially because of while loops and partially because of the random nature I have found that this tester has the potential to find and enter infinite loops, which I attempt to catch by process timeouts. Before running a full match this random tester chooses the number of players, what cards are inside of the kingdom card set. Once the game state is randomized it starts at the beginning of a players turn, action phase, and progresses to the end, buy phase. During the testers action phase it finds a random kingdom card from the players hand which it will then choose whether or not it gets played; if the tester chooses not to play the card it progresses to the players buy phase, if it does play the card it randomizes the players choices (1, 2, and 3) calls the playCard function and then continues this cycle until the player has no more actions – for events where an action card played provides the player with additional actions. Likewise, the players buy phase makes purchases, determined by randomly picking a supply position, until the player either runs out of buys or if the tester chooses not to make any purchases. Of course after the action and buy phases the endTurn function is called.

Evaluation of Implementations

Having examined in detail the various tests run on Steven's and Griffin's dominion.c implementations I feel it is now appropriate to begin my evaluation of their reliability. Note table 1 which displays the best code coverage I found while running the full test suite on Steven's and Griffins implementations. Both implementations went through all non-randomized tests without issue. This leads me to believe that both implementations provide most of the basic functionality required to play a match of dominion. However, once the randomized tests began I immediately had an issue with Griffins code segfaulting when I ran my random adventurer test. The tester was sending an invalid hand position to the discardCard function. Additionally, both tests faired similarly with my randomized full dominion tester. With my random tester both implementations faced issues of potentially finding infinite loops when

attempting to play or purchase certain cards. These were similar to the problems faced by my own implementation; as such I believe the tester's structure to be the main culprit.

As observed in the code coverage table below the randomized dominion test covers a much larger portion of a dominion implementation when compared to my other tests. Not finding the other two randomized tests particularly successful while testing my rating of each implementation comes from the unit tests – which find and cover basic game functionality, card tests – which extend their functionality into the makings of each players turn, and the random dominion test – which has the potential covers as many scenarios as rand can generate. Because both implementations faired quite well in passing the full test suite both receive a 90% reliability rating from me. Based on coverage from the dominion randomized tester both implementations should handle sensible input very well. I believe the reason Steven's coverage for this test was 73.06%, as opposed to Griffins 76.20%, was that Steven's encountered the infinite loops more frequently. This begs the question as to why both implementations receive a 90%, which is given by Griffins proving a little more troublesome to implement and an immediate segfault I found with the random adventurer test – seemed only to be an issue in VS.

IMPLEMENTATION SUMMARIES		
CATEGORY & TEST	GRIFFINS BEST COVERAGE	STEVENS BEST COVERAGE
Unit Tests:		
1) Play Card	21.58%	20.73%
2) Update Coins	21.92%	21.07%
3) Full Deck Count	23.46%	22.63%
4) Buy Card	33.22%	32.47%
Card Tests:		
1) Steward	38.70%	41.45%
2) Smithy	42.12%	44.73%
3) Great Hall	44.35%	46.98%
4) Village	44.35%	46.98%
Random Tests:		
1) Remodel	28.42%	29.19%
2) Adventurer	16.44%	16.58%
3) Dominion Implementation	76.20%	73.06%

Table 1: Code Coverage