CS362: Final Test Report

Devlin Junker 931636867

June 11th, 2012

1 Test Approach

A majority of my testing is done using differential testing that we first covered in class. This seemed like the most logical testing strategy seeing as we have 30 different implementations of dominion in this class, so comparing them to each other will provide ample opportunity for differences in code. I also included some Buy Card and Draw Card random testing and unit testing to check if these specific functions performed up to specification.

My main test suite file, *testall* first calls the Buy Card test 2000 times with random game states and checks to see if the state is correct after the call of Buy Card. I do the same thing with my draw card tests and check to see if the state is correct after the call. After the Draw Card tests have run, I then play 2000 random games, each with 100 turns in them and produce an output file that can be compared against other tester output files from other games of dominion.

My biggest gripe with differential testing is the difficulty of finding the specific fault in the code that caused the error that is visible. I am able to narrow down the problem to a function or block of the code, but it is still necessary to scan through the code to find the specific line that the fault is on.

I handled segfaults and crashes with gdb to backtrace to the line that caused the crash. I then would send a bug report to the person with the line of code that crashed, and my assessment of what could have caused it. In some cases, I was unable to find out what caused it, so I sent a bug report with my best guess.

I saved the output from testing on my own implementation of dominion as *baseline*, this can be used to for differential testing against all of my teammates implementations.

I have included some of the source code for the the random Game Tests, buy card tests, draw card tests and adventurer tests at the end of this report.

2 Test Statistics

I ran the test suite on all of my team members code, so for each there were 2000 tests of buycard, 2000 tests of draw card and 2000 random games played, each game with 100 turns each. The coverage I was able to get on all of the implementations of dominion was right around 50% for all code that compiled. Buy Card, Draw Card and Play Card have all been tested over 50,000 times each for each implementation. Each of the kingdom cards in my test suite (adventurer, council room, feast, gardens, mine, remodel, smithy, village, baron, and great hall) have been tested about 250 times each individuals code.

Coverage: junkerd (51%), turcottm (N/A), gibsonro (56%), luisramg (N/A), nolandr (N/A), gormanv (N/A), genzg (52%), bartoskz (N/A)

A couple of my teammates code didn't compile, with a number of them still containing errors in draw card, a couple of my teammates still had infinite loops after I had sent them bug reports before the last test report, so I'm not sure if they are still in the class or updating their code and a number of them segfaulted so I was unable to obtain coverage numbers for their implementations.

3 Communication

Each bug report was placed in the bug-reports folder with a small description of the bug and a possible solution to it. In some bug reports, I also included other suggestions for the code. I have included an example below:

```
Infinite Loop in cardEffect for Feast

Best Solution: set x = 0 after printf("None of that card left, sorry!\n")

Other Notes: Might want to check choices to see if valid,
my test program sends -1 for choice1 which is invalid
```

A couple of bugs that I found, I was unable to determine the cause of. In one of my teammates code, I was unable to find the flaw that led to a seg fault during the adventurer tests. The segfault was caused by a discardCount of 185275669, but when I turned on debugging with the preprocessor content, at the beginning of the adventurer implemenation, the discard count was only 474. I left the following bug report:

```
Seg Fault in Adventurer Tests

Best Solution: Unknown
```

```
Other Notes:
dominion.c:844
state->discard[currentPlayer][state->discardCount[currentPlayer]] =
state->deck[currentPlayer][i];

During testing, I printed a discardCount of 185275669, however I turned on debugging
and found that at the beginning of that call to adventurer, the discard count was
only 475. I think there may be a bug in your implementation that changes the
discardCount value.
```

I never recieved response from anybody for the bug reports that I issued, so I'm not sure whether or not they were clear enough to solve the bugs I found, or if the bugs were ever solved.

I only recieved one bug report that was sent to the entire class about bugs in the baseline dominion code, I fixed these bugs but didn't reply to the email because I knew that the person who sent it out wasn't testing my code specifically, only informing everyone of bugs they found in their own code.

4 Quality of Code

My code has begun to improve, with more bugs being squashed with more tests being run. I have correctly implemented the adventurer card, and I am positive that draw card and buy card are performing properly. I also am sure that play card will not be succesfull if it is not called with an action card. Differential testing has helped me improve my code, because I can see the differences in output between my implementation and other peoples and notice when there is something that isn't the same in our outputs that should be further investigated.

After testing a number of people's code, I found that a majority of the implementations of adventurer were not correct. I was also suprised by the large number of implementations that wouldn't compile, or would segfault inside of their implementations in their functions.

I found a couple of bugs in the call to shuffle during game intialization that happened in my random tests on one teammates code and during the buy card tests on another. I wasn't sure what the cause was, but it may have to do with my test suite because it happened in two implementations, although the initialize game functions were identical for these two implementations of dominion.

5 Bugs Found

The number one bug that I found was the infinite loop bug when playing a feast card, I included the same bug report for each of my testing team that contained the bug.

The bug report was simple, explaining to set x = 0 if there are no more cards left in the pile. My random games test found this bug because during the random games, when playing a card, the choices are set to -1. I included this detail in the bug report and suggested that before checking the number of cards left, the game should make sure it is a valid card choice.

The other bug I found in other peoples code was an error in buycard where the number of buys is not decremented after buying.

In my own code, I found a bug in buycard where if the number of coins was equal to the cost of the card, it wouldn't be bought. This bug cropped up in my first differential test because my implementation wasn't able to buy something and the other persons was.

I also found a bug in my adventurer implementation. During differential testing I noticed that in one of the games with my implementation, card number -76493526 was in the hand of the player. After looking through the output of the random games I found that this was placed in the deck after playing an adventurer card. In the implementation of the adventurer card, there was a bug in the code such that when searching through the deck for two treasure cards, the temporary variable to hold the cards being searched through was assigned to the entire deck array instead of a particular card in the deck.

Before change:

```
i = (long) state->deck[(state->deckCount[currentPlayer]-1)];
```

After change:

```
i = (long) state->deck[currentPlayer][(state->deckCount[currentPlayer]-1)];
```

6 Future Testers

In the future, more coverage needs to be done of the card effects, this is the least tested part of the implementations because there are a lot of different things that need to be checked based on which card is played. I developed a good infrastructure for running individual types of tests (unit, differential) and being able to include these in an overall test suite. The testsuite is called by running make testdom, the unit tests can be run by calling make testBuyCard or make testDrawCard.

7 Test Code

```
int randomGameTests(FILE* output){
for (seed = 1; seed <= NUM_TESTS; seed++) {
initializeGame(2, cards, seed, &G);
```

```
/* Play Game with NUM_TURNS turns */
for(n = 0; n < NUM_TURNS; n++){
/* Print some Details */
fprintf(output, "[%d, %d] ", n, seed);
printDetails(output, G);

/* Determine Random Move */
op = floor(Random() * 4);
card = floor(Random() * treasure_map);
pos = floor(Random() * G.handCount[G.whoseTurn]);

/*Make Move and Print Details of Move */
switch(op){
case 0:
fprintf(output, "BUY %d (cost: %d) ", card, getCost(card));
result = buyCard(card, &G);
printResult(output, result);
break;
case 1:
fprintf(output, "DRAW ");
result = drawCard(G.whoseTurn, &G);
printResult(output, result);
break;
case 2:
fprintf(output, "END TURN ");
result = endTurn(&G);
printResult(output, result);
break;
case 3:
fprintf(output, "PLAY position %d ", pos);
result = playCard(pos, -1, -1, -1, &G);
printResult(output, result);
break;

}

fprintf(output, "\n");
}

}

int checkBuyCard(FILE* output, int card, struct gameState* post){
```

```
struct gameState pre;
memcpy(&pre, post, sizeof(struct gameState));

int who = pre.whoseTurn;

/* Call Buy Card*/
int r = buyCard(card, post);

/* Set Expected State */
if(pre.supplyCount[card] > 0 && pre.coins >= getCost(card) && pre.numBuys > 0){
pre.discard[who][pre.discardCount[who]] = card;
pre.supplyCount[card]--;
pre.discardCount[who]++;
pre.coins -= getCost(card);
pre.numBuys--;
pre.phase = 1;
/* Check for Errors */
}else{
if(r != -1){
fprintf(output, "ILLEGAL BUY ");
return -1;
}else{
fprintf(output, "BUY FAIL");
return 0;
}
}

/* Compare Expected State with Actual State*/
if(memcmp(&pre, post, sizeof(struct gameState)) != 0){
fprintf(output, "UNEXPECTED CHANGE");
return -1;
}

return 0;
fprintf(output, "BUY COMPLETE");
}

int checkDrawCard(FILE * output, struct gameState *post) {
struct gameState pre;
memcpy (&pre, post, sizeof(struct gameState));

int r;
```

```
int p = pre.whoseTurn;

/* Call Draw Card */
r = drawCard (p, post);

/* Set Expected State */
if (pre.deckCount[p] > 0) {
pre.handCount[p]++;
pre.hand[p][pre.handCount[p]-1] = pre.deck[p][pre.deckCount[p]-1];
pre.deckCount[p]--;
} else if (pre.discardCount[p] > 0) {
memcpy(pre.deck[p], post->deck[p], sizeof(int) * pre.discardCount[p]);
memcpy(pre.discard[p], post->discard[p], sizeof(int)*pre.discardCount[p]);
pre.hand[p][post->handCount[p]-1] = post->hand[p][post->handCount[p]-1];
pre.handCount[p]++;
pre.deckCount[p] = pre.discardCount[p]-1;
pre.discardCount[p] = 0;
}

/* Compare Expected State with Actual State */
if(memcmp(&pre, post, sizeof(struct gameState)) != 0){
fprintf(output, "UNEXPECTED CHANGE");
return -1;
}

return 0;
}

int checkAdventurer(FILE* output, struct gameState* post){
struct gameState pre;
memcpy(&pre, post, sizeof(struct gameState));

int i;
int who = pre.whoseTurn;

int treasure_count = 0;
int t1, t2 = -1;
int pmin = MAX_DECK;

for(i = 0; i < post->deckCount[who]; i++){
t1 = post->deck[who][i];
```

```
if(t1 == copper || t1 == silver || t1 == gold){
treasure_count++;
if(t2 != -1){
break;
}else{
t2 = t1;
}
if(pmin > i){
pmin = i;
}
}
}

/* Play Adventurer Card*/
int r = cardEffect(adventurer, -1, -1, -1, post, 0, 0);

/* Set Expected State */
if(treasure_count >= 2){
pre.hand[who][pre.handCount[who]] = t1;
pre.handCount[who]++;
}

if(treasure_count > 0){
pre.hand[who][pre.handCount[who]] = t2;
pre.handCount[who]++;
pre.playedCardCount+=(pre.deckCount[who]-pmin-2);
pre.deckCount[who]-=(pre.deckCount[who]-pmin-1);
}else{
pre.playedCardCount+=pre.deckCount[who];
pre.deckCount[who] = 0;
}

for(i = 0; i < pre.playedCardCount; i++){
pre.playedCards[i] = post->playedCards[i];
}

discardCard(0, who, &pre, 0);

/* Compare Expected State with Actual State*/
if(memcmp(&pre, post, sizeof(struct gameState)) != 0){
fprintf(output, "UNEXPECTED CHANGE ");
return -1;
```

```
}
return 0;
}
```