Brian Laccone

CS 362

4/29/2018

Assignment-3

Bugs

The unit tests I did for the four functions in dominion.c produced no bugs. However, The four card tests that I created discovered a few bugs.

Adventurer Card (cardtest1.c)

1) Test 1 and Test 4 - Once the adventurer card is complete, the current player should have two more treasure cards in hand but after running the test, not only does the player not have two extra treasure cards but the player does not draw any cards. I believe this bug was introduced by me during assignment 2 because I commented out the part where the player would draw the cards.

Smithy Card (cardtest2.c)

2) Test 1 - Smithy allows the current player to draw 3 cards. After the smithyCardEffect function completes, the current player's hand count doesn't increase by 3 cards. This bug was introduced by myself in assignment 2. I made the for loop, which controls how many cards are drawn, not draw any cards.
3) Test1 - After the smithyCardEffect function completes, the current player's deck count doesn't decrease by 3

Great Hall Card (cardtest3.c)

4) Test 1 - Great Hall allows the current player to draw a card, increase their actions by 1, and have 1 extra victory points by having this card. After Great Hall is played, the current player does not draw a card. This bug confused me because I didn't introduce any bugs

into this function. I’m not sure exactly what is happening with this bug but the current player is not drawing any cards.

5) The current player’s victory points don’t increase by having in their possession

Council Room Card (cardtest4.c)

6) Test 1 - Council Room allows the current player to draw 4 cards and increases their number of buys by 1. This card also allows the other players in the game to draw 1 cards. The current player didn’t draw any cards after running this test and their deck count didn’t decrease by 4. This is because of a bug that I introduced in assignment 2 that draws 6 cards instead of 4.

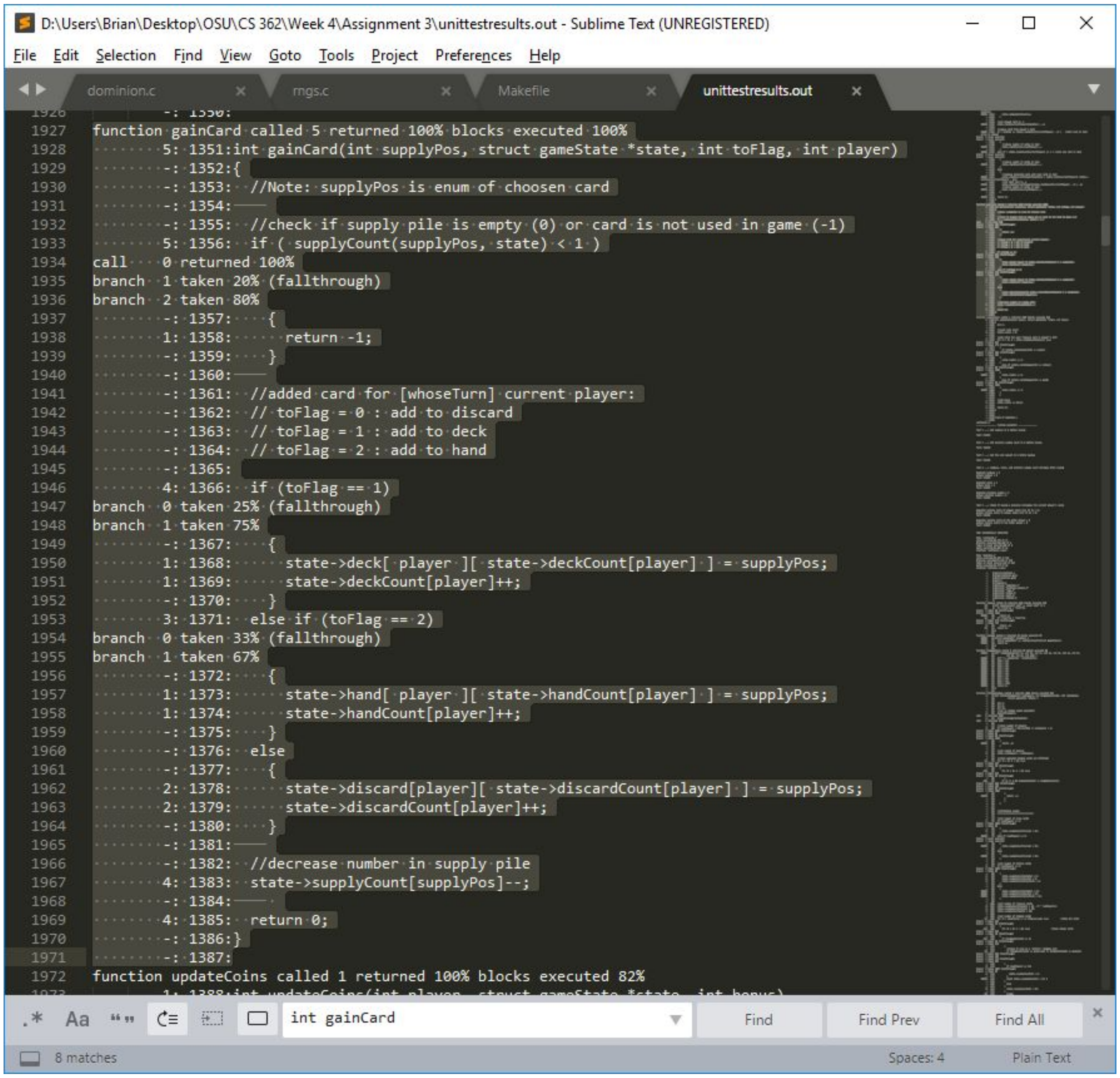
Unit Testing

gainCard() (unittest1.c)

```
File 'unittest1.c'
Lines executed:91.49% of 47
Branches executed:100.00% of 4
Taken at least once:50.00% of 4
Calls executed:90.00% of 30

File 'dominion.c'
Lines executed:18.77% of 554
Branches executed:17.35% of 415
Taken at least once:14.46% of 415
Calls executed:8.51% of 94
```

My first unit test covered the function gainCard(). I believe that I received 100% statement and branch coverage for this function. Unfortunately my overall coverage of dominion.c was only 18.77% with 17.35% branch coverage. I could have feed the function several different states to try and test what it would do with better defined states instead of just the same state every call to it. Although I can’t think of what I would have missed in boundary coverage, I’m sure there is something I could have done to test the boundaries of this function even more. I made sure that I hit every branch by feeding the function supply counts that were empty as well as all three “toFlag” branch options.


D:\Users\Brian\Desktop\OSU\CS 362\Week 4\Assignment 3\unittestresults.out - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
dominion.c
rngs.c
Makefile
unittestresults.out
int gainCard
Find
Find Prev
Find All
8 matches
Spaces: 4
Plain Text

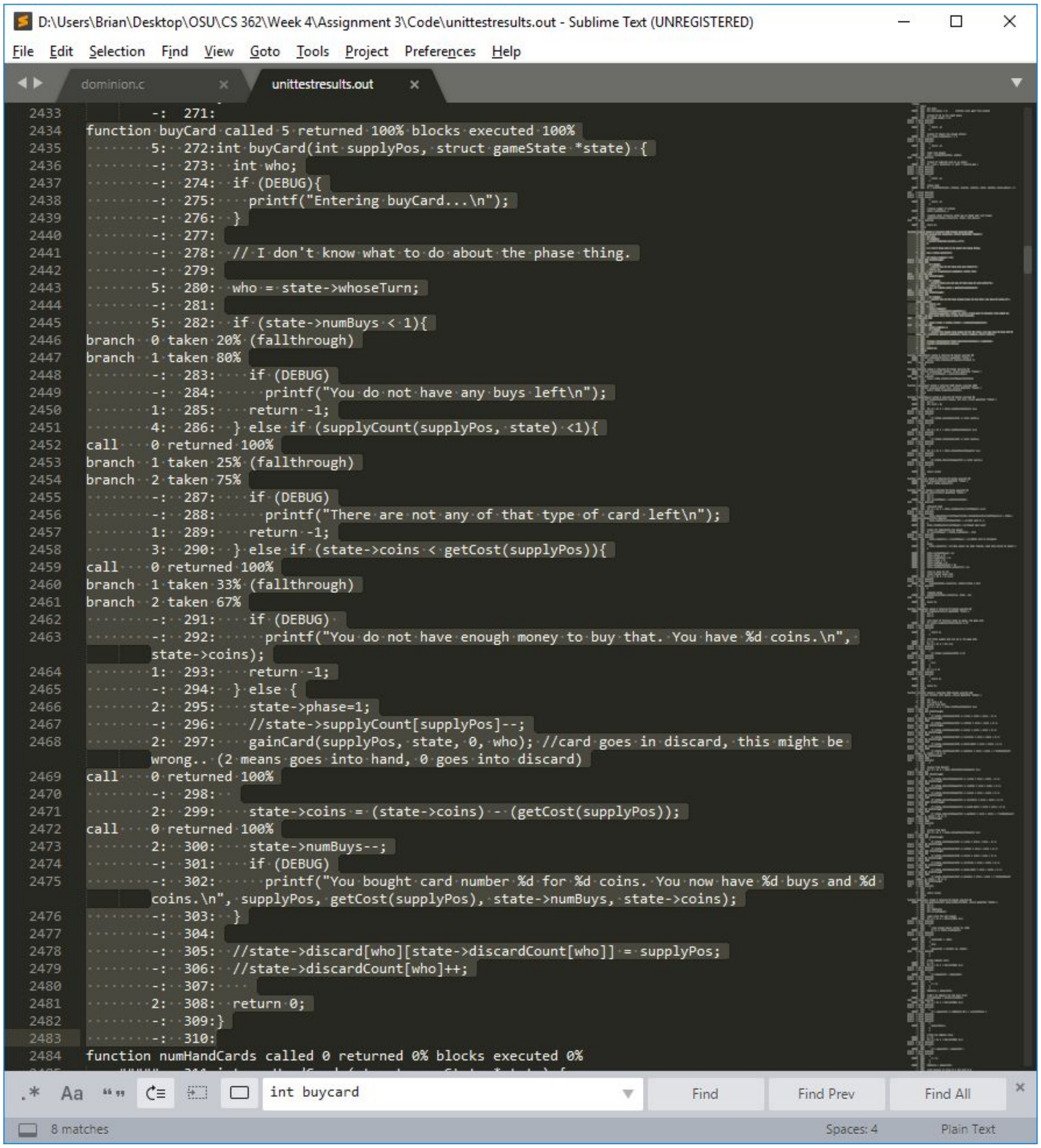
```
function gainCard called 5 returned 100% blocks executed 100%
        5: 1351:int gainCard(int supplyPos, struct gameState *state, int toFlag, int player)
        -: 1352:{
        -: 1353:  //Note: supplyPos is enum of choosen card
        -: 1354:
        -: 1355:  //check if supply pile is empty (0) or card is not used in game (-1)
        5: 1356:  if ( supplyCount(supplyPos, state) < 1 )
call    0 returned 100%
branch  1 taken 20% (fallthrough)
branch  2 taken 80%
        -: 1357:    {
        1: 1358:      return -1;
        -: 1359:    }
        -: 1360:
        -: 1361:  //added card for [whoseTurn] current player:
        -: 1362:  // toFlag = 0 : add to discard
        -: 1363:  // toFlag = 1 : add to deck
        -: 1364:  // toFlag = 2 : add to hand
        -: 1365:
        4: 1366:  if (toFlag == 1)
branch  0 taken 25% (fallthrough)
branch  1 taken 75%
        -: 1367:    {
        1: 1368:      state->deck[ player ][ state->deckCount[player] ] = supplyPos;
        1: 1369:      state->deckCount[player]++;
        -: 1370:    }
        3: 1371:  else if (toFlag == 2)
branch  0 taken 33% (fallthrough)
branch  1 taken 67%
        -: 1372:    {
        1: 1373:      state->hand[ player ][ state->handCount[player] ] = supplyPos;
        1: 1374:      state->handCount[player]++;
        -: 1375:    }
        -: 1376:  else
        -: 1377:    {
        2: 1378:      state->discard[player][ state->discardCount[player] ] = supplyPos;
        2: 1379:      state->discardCount[player]++;
        -: 1380:    }
        -: 1381:
        -: 1382:  //decrease number in supply pile
        4: 1383:  state->supplyCount[supplyPos]--;
        -: 1384:
        4: 1385:  return 0;
        -: 1386:}
        -: 1387:
function updateCoins called 1 returned 100% blocks executed 82%
```

<u>buyCard() (unittest2.c)</u>

```
File 'unittest2.c'
Lines executed:94.37% of 71
Branches executed:100.00% of 4
Taken at least once:50.00% of 4
Calls executed:93.88% of 49

File 'dominion.c'
Lines executed:25.09% of 554
Branches executed:35.66% of 415
Taken at least once:21.69% of 415
Calls executed:12.77% of 94
```

My unittest2.c file contained the unit test for the buyCard() function. The file says that I had returned 100% and had 100% blocks executed. I don’t believe that I had 100% branch coverage because I don’t think DEBUG was set to true. DEBUG was meant to be a visual representation of the sequence of code in the buyCard() function. I didn’t hit the print command in any of the if (DEBUG) {} statements. But I find it interesting that it gave me 100% of blocks executed as well as what looks like full branch coverage. In the future I will look into that and maybe even try to hit every DEBUG conditional. My statement coverage of the whole dominion.c file was 25.09% and my branch coverage was 35%. A test I could have done to better test boundary coverage would have been to set the number of buys and coins to a very high number and then buy several items to see how the function would handle it.


D:\Users\Brian\Desktop\OSU\CS 362\Week 4\Assignment 3\Code\unittestresults.out - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
dominion.c
unittestresults.out
int buycard
Find
Find Prev
Find All
8 matches
Spaces: 4
Plain Text

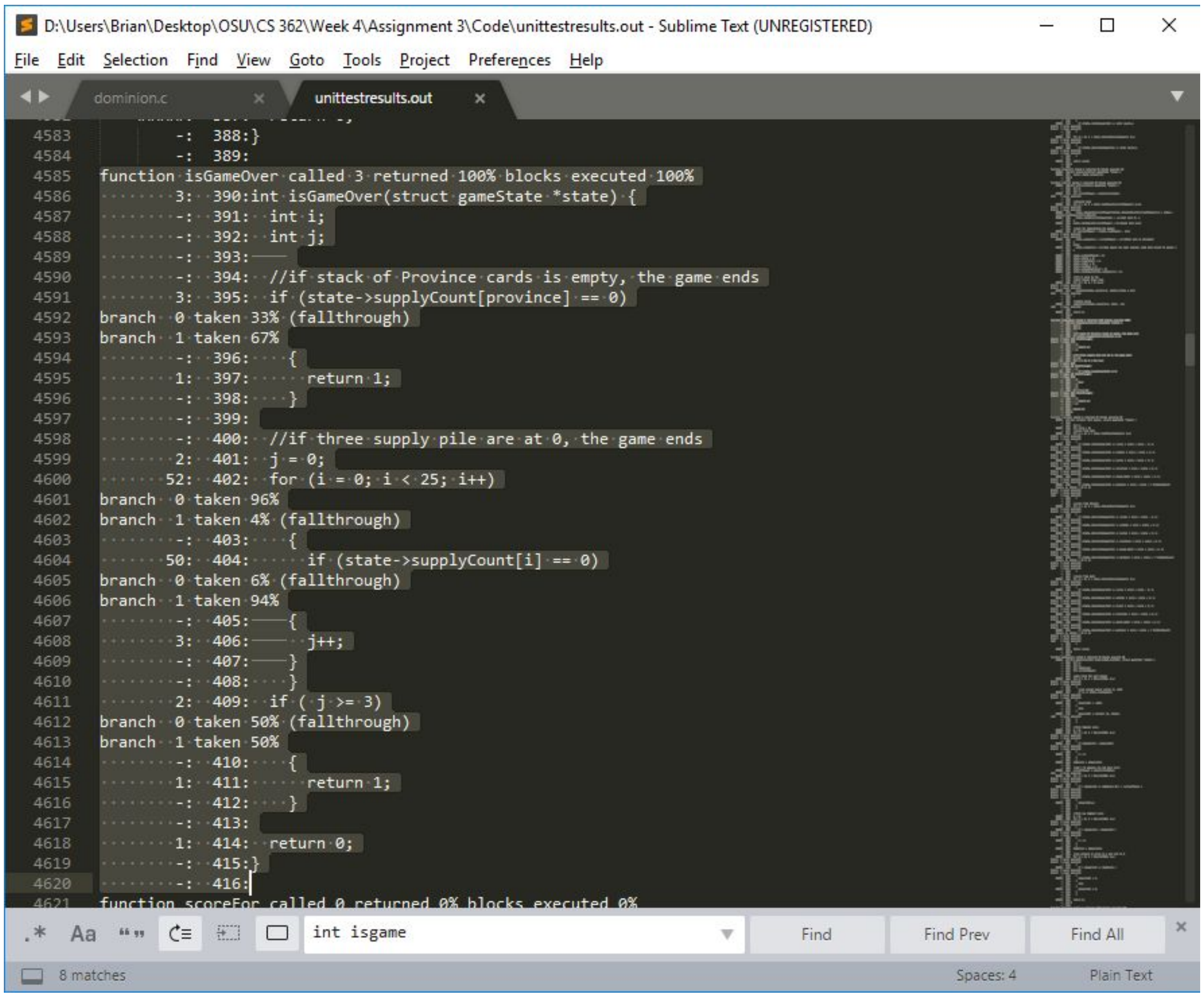
```
        -:  271:
function buyCard called 5 returned 100% blocks executed 100%
        5:  272:int buyCard(int supplyPos, struct gameState *state) {
        -:  273:  int who;
        -:  274:  if (DEBUG){
        -:  275:    printf("Entering buyCard...\n");
        -:  276:  }
        -:  277:
        -:  278:  // I don't know what to do about the phase thing.
        -:  279:
        5:  280:  who = state->whoseTurn;
        -:  281:
        5:  282:  if (state->numBuys < 1){
branch  0 taken 20% (fallthrough)
branch  1 taken 80%
        -:  283:    if (DEBUG)
        -:  284:      printf("You do not have any buys left\n");
        1:  285:    return -1;
        4:  286:  } else if (supplyCount(supplyPos, state) <1){
call    0 returned 100%
branch  1 taken 25% (fallthrough)
branch  2 taken 75%
        -:  287:    if (DEBUG)
        -:  288:      printf("There are not any of that type of card left\n");
        1:  289:    return -1;
        3:  290:  } else if (state->coins < getCost(supplyPos)){
call    0 returned 100%
branch  1 taken 33% (fallthrough)
branch  2 taken 67%
        -:  291:    if (DEBUG)
        -:  292:      printf("You do not have enough money to buy that. You have %d coins.\n",
        state->coins);
        1:  293:    return -1;
        -:  294:  } else {
        2:  295:    state->phase=1;
        -:  296:    //state->supplyCount[supplyPos]--;
        2:  297:    gainCard(supplyPos, state, 0, who); //card goes in discard, this might be
        wrong.. (2 means goes into hand, 0 goes into discard)
call    0 returned 100%
        -:  298:
        2:  299:    state->coins = (state->coins) - (getCost(supplyPos));
call    0 returned 100%
        2:  300:    state->numBuys--;
        -:  301:    if (DEBUG)
        -:  302:      printf("You bought card number %d for %d coins. You now have %d buys and %d
        coins.\n", supplyPos, getCost(supplyPos), state->numBuys, state->coins);
        -:  303:  }
        -:  304:
        -:  305:  //state->discard[who][state->discardCount[who]] = supplyPos;
        -:  306:  //state->discardCount[who]++;
        -:  307:
        2:  308:  return 0;
        -:  309:}
        -:  310:
function numHandCards called 0 returned 0% blocks executed 0%
```

isGameOver() (unittest3.c)

```
File 'unittest3.c'
Lines executed:87.88% of 33
Branches executed:100.00% of 4
Taken at least once:50.00% of 4
Calls executed:82.35% of 17

File 'dominion.c'
Lines executed:17.87% of 554
Branches executed:17.83% of 415
Taken at least once:14.94% of 415
Calls executed:7.45% of 94
```

My unittest3.c file contains the tests for the isGameOver() function. This test got 100% statement coverage and 100% branch coverage for the isGameOver() function. Luckily, this was a very simple function and only took me 3 calls of it to cover 100%. I don’t have a way to know for sure but I feel confident that I covered the boundary coverage pretty well. The only ways that I can think of to stretch the boundary tests would be to feed the function information that wouldn’t be possible in the game. But, I am new to testing so I wouldn't be surprised if I was wrong in saying that.


D:\Users\Brian\Desktop\OSU\CS 362\Week 4\Assignment 3\Code\unittestresults.out - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
dominion.c
unittestresults.out
int isgame
Find
Find Prev
Find All
8 matches
Spaces: 4
Plain Text

```
        -:  388:}
        -:  389:
function isGameOver called 3 returned 100% blocks executed 100%
        3:  390:int isGameOver(struct gameState *state) {
        -:  391:  int i;
        -:  392:  int j;
        -:  393:
        -:  394:  //if stack of Province cards is empty, the game ends
        3:  395:  if (state->supplyCount[province] == 0)
branch  0 taken 33% (fallthrough)
branch  1 taken 67%
        -:  396:    {
        1:  397:      return 1;
        -:  398:    }
        -:  399:
        -:  400:  //if three supply pile are at 0, the game ends
        2:  401:  j = 0;
       52:  402:  for (i = 0; i < 25; i++)
branch  0 taken 96%
branch  1 taken 4% (fallthrough)
        -:  403:    {
       50:  404:      if (state->supplyCount[i] == 0)
branch  0 taken 6% (fallthrough)
branch  1 taken 94%
        -:  405:	{
        3:  406:	  j++;
        -:  407:	}
        -:  408:    }
        2:  409:  if ( j >= 3)
branch  0 taken 50% (fallthrough)
branch  1 taken 50%
        -:  410:    {
        1:  411:      return 1;
        -:  412:    }
        -:  413:
        1:  414:  return 0;
        -:  415:}
        -:  416:
function scoreFor called 0 returned 0% blocks executed 0%
```

<u>shuffle() (unittest4.c)</u>

File 'unittest4.c'

Lines executed:87.18% of 39

Branches executed:100.00% of 6

Taken at least once:50.00% of 6

Calls executed:81.82% of 22

File 'dominion.c'

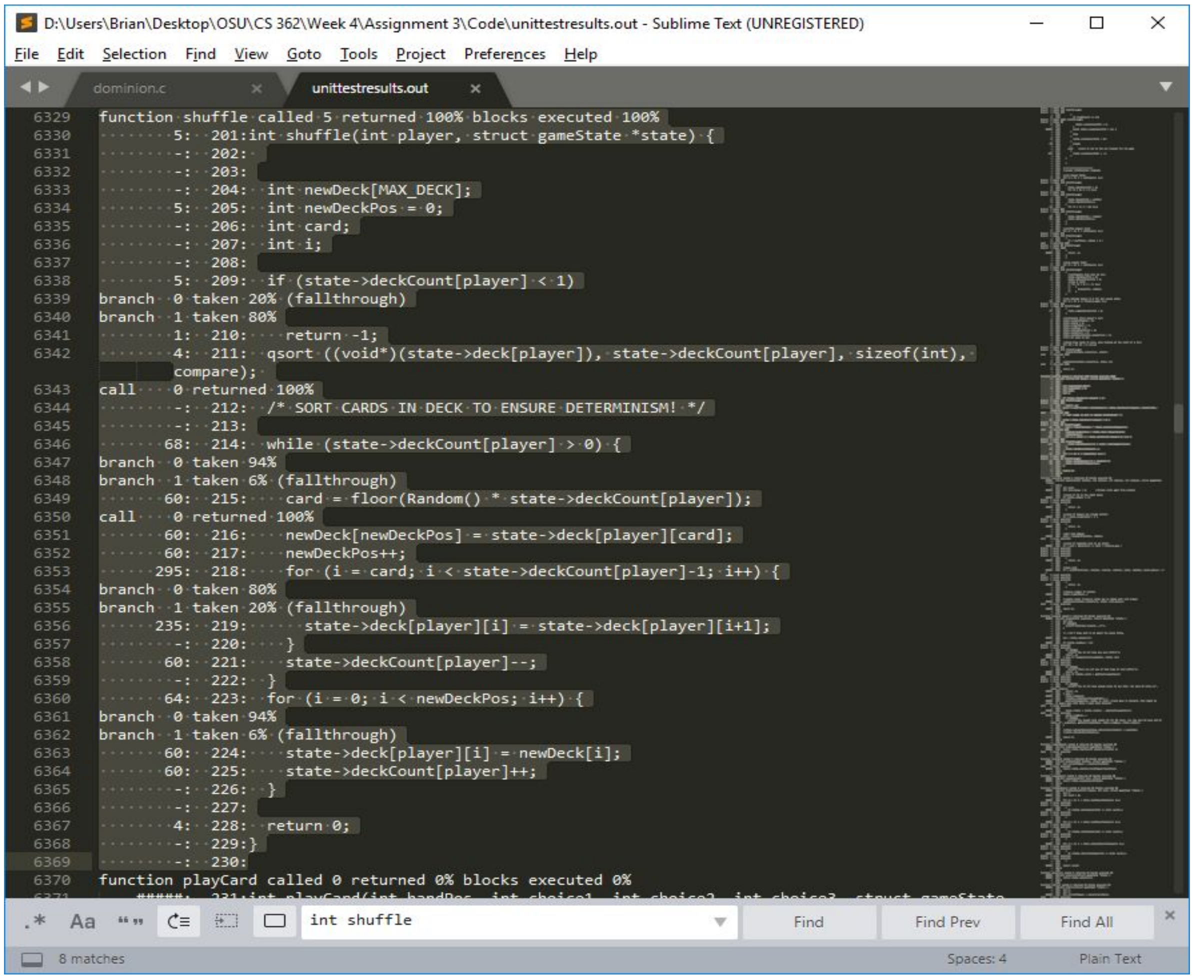
Lines executed:16.43% of 554

Branches executed:15.90% of 415

Taken at least once:13.49% of 415

Calls executed:7.45% of 94

My unittest4.c file was created to test the shuffle() file. The test received 100% statement and 100% branch coverage for the function. Unfortunately it only received a 16.34% statement and 15.90% branch coverage for the whole dominion.c file. The first that jumps out to me is the fact that I called it 5 times to have full coverage. The shuffle() function is not very complicated and could have full coverage with less calls. After looking at the coverage for this function I noticed that I definitely could have done a better job at boundary coverage for this function. I didn't even test the minimum and maximum values. I feel like I could learn to have better boundary coverage on most of my unit tests. I might be missing a crucial bug by foregoing boundary tests.


D:\Users\Brian\Desktop\OSU\CS 362\Week 4\Assignment 3\Code\unittestresults.out - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
dominion.c
unittestresults.out

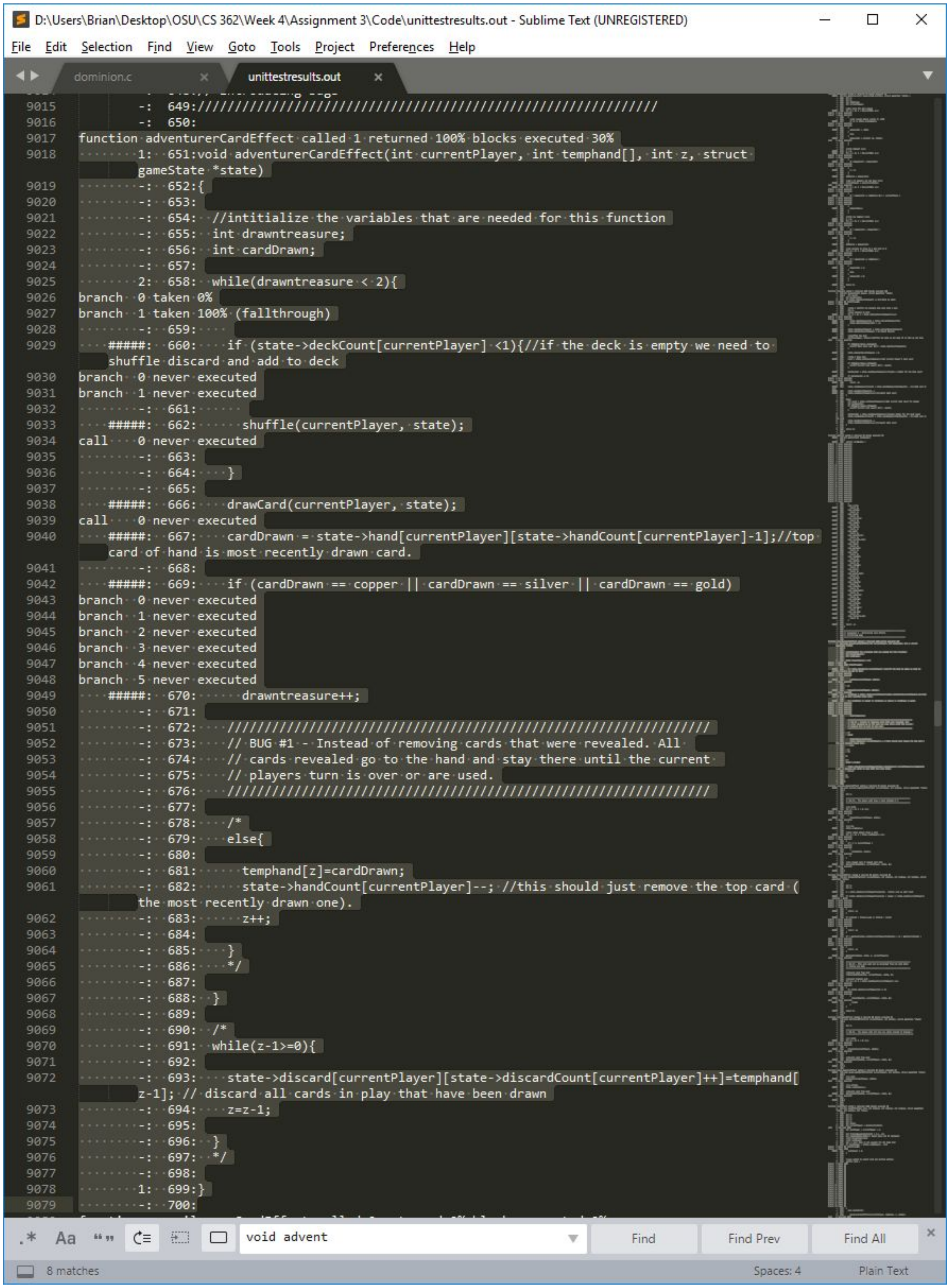
```
function shuffle called 5 returned 100% blocks executed 100%
        5:  201:int shuffle(int player, struct gameState *state) {
        -:  202:
        -:  203:
        -:  204:  int newDeck[MAX_DECK];
        5:  205:  int newDeckPos = 0;
        -:  206:  int card;
        -:  207:  int i;
        -:  208:
        5:  209:  if (state->deckCount[player] < 1)
branch  0 taken 20% (fallthrough)
branch  1 taken 80%
        1:  210:    return -1;
        4:  211:  qsort ((void*)(state->deck[player]), state->deckCount[player], sizeof(int), 
        compare);
call    0 returned 100%
        -:  212:  /* SORT CARDS IN DECK TO ENSURE DETERMINISM! */
        -:  213:
       68:  214:  while (state->deckCount[player] > 0) {
branch  0 taken 94%
branch  1 taken 6% (fallthrough)
       60:  215:    card = floor(Random() * state->deckCount[player]);
call    0 returned 100%
       60:  216:    newDeck[newDeckPos] = state->deck[player][card];
       60:  217:    newDeckPos++;
      295:  218:    for (i = card; i < state->deckCount[player]-1; i++) {
branch  0 taken 80%
branch  1 taken 20% (fallthrough)
      235:  219:      state->deck[player][i] = state->deck[player][i+1];
        -:  220:    }
       60:  221:    state->deckCount[player]--;
        -:  222:  }
       64:  223:  for (i = 0; i < newDeckPos; i++) {
branch  0 taken 94%
branch  1 taken 6% (fallthrough)
       60:  224:    state->deck[player][i] = newDeck[i];
       60:  225:    state->deckCount[player]++;
        -:  226:  }
        -:  227:
        4:  228:  return 0;
        -:  229:}
        -:  230:
function playCard called 0 returned 0% blocks executed 0%
```

Adventurer Card (cardtest1.c)

```
File 'cardtest1.c'
Lines executed:98.41% of 63
Branches executed:100.00% of 22
Taken at least once:77.27% of 22
Calls executed:97.50% of 40
```

```
File 'dominion.c'
Lines executed:18.59% of 554
Branches executed:21.93% of 415
Taken at least once:13.73% of 415
Calls executed:9.57% of 94
```

My cardtest1.c was created to test the adventurer card. This test got an appalling 30% blocks executed. I didn’t know this until I finished all the unit tests but it looks like the main while loop that controls never executed. It is no wonder the card never drew any treasure cards. This was definitely a surprise to me. I thought the card was never being drawn was because of the bug that I introduced but it looks like there is even a greater bug that is causing this function to not work properly. The next steps would be to figure out if the bug lies in the actual program itself or did I introduce this bug in my test by not setting up the game state properly before passing it to the cardEffect() function. The implications of this test being not covering much is a big deal because either I messed up my test code or their is a major bug in the system. Either option is bad.


D:\Users\Brian\Desktop\OSU\CS 362\Week 4\Assignment 3\Code\unittestresults.out - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
dominion.c
unittestresults.out

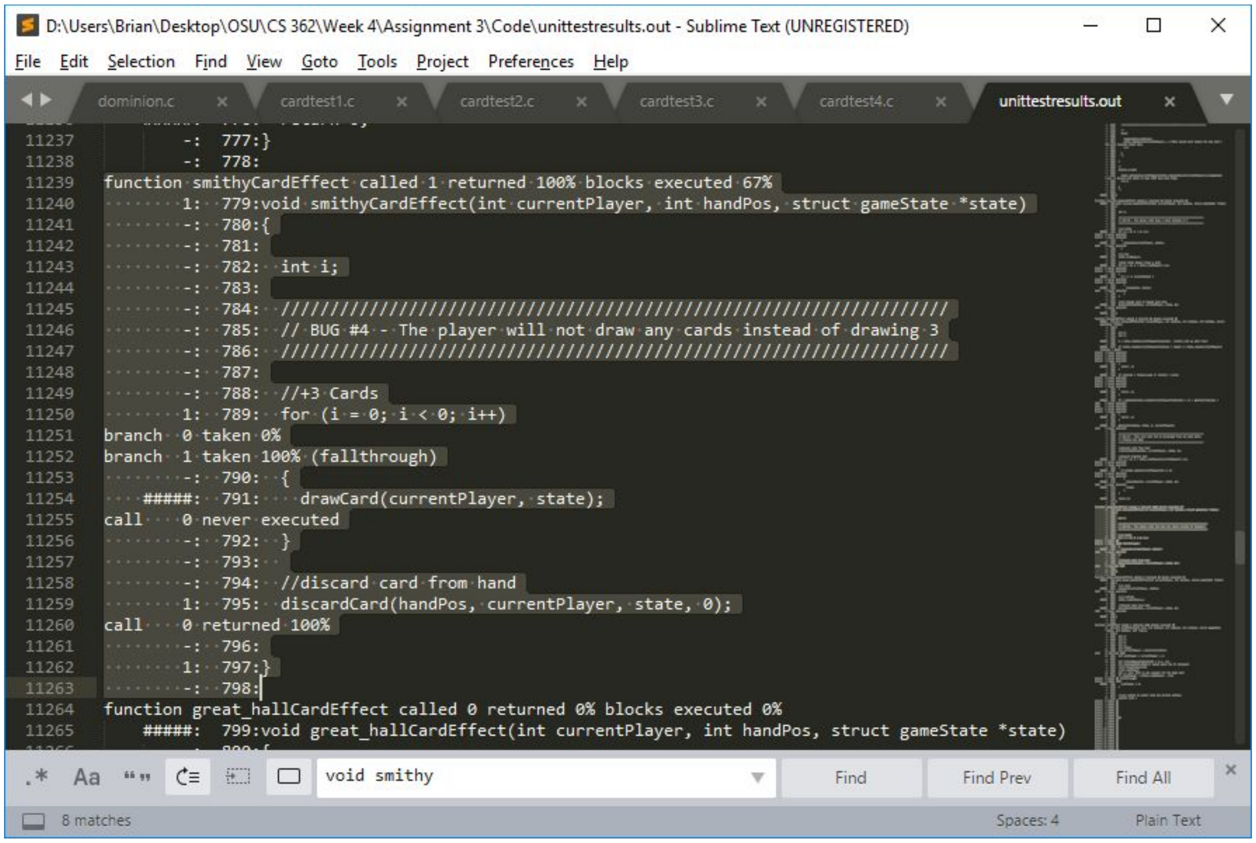
```
        -:  649:///////////////////////////////////////////////////////////////
        -:  650:
function adventurerCardEffect called 1 returned 100% blocks executed 30%
        1:  651:void adventurerCardEffect(int currentPlayer, int temphand[], int z, struct gameState *state)
        -:  652:{
        -:  653:
        -:  654:  //intitialize the variables that are needed for this function
        -:  655:  int drawntreasure;
        -:  656:  int cardDrawn;
        -:  657:
        2:  658:  while(drawntreasure < 2){
branch  0 taken 0%
branch  1 taken 100% (fallthrough)
        -:  659:    
    #####:  660:    if (state->deckCount[currentPlayer] <1){//if the deck is empty we need to shuffle discard and add to deck
branch  0 never executed
branch  1 never executed
        -:  661:      
    #####:  662:      shuffle(currentPlayer, state);
call    0 never executed
        -:  663:
        -:  664:    }
        -:  665:
    #####:  666:    drawCard(currentPlayer, state);
call    0 never executed
    #####:  667:    cardDrawn = state->hand[currentPlayer][state->handCount[currentPlayer]-1];//top card of hand is most recently drawn card.
        -:  668:
    #####:  669:    if (cardDrawn == copper || cardDrawn == silver || cardDrawn == gold)
branch  0 never executed
branch  1 never executed
branch  2 never executed
branch  3 never executed
branch  4 never executed
branch  5 never executed
    #####:  670:      drawntreasure++;
        -:  671:
        -:  672:    ///////////////////////////////////////////////////////////////
        -:  673:    // BUG #1 - Instead of removing cards that were revealed. All
        -:  674:    // cards revealed go to the hand and stay there until the current
        -:  675:    // players turn is over or are used.
        -:  676:    ///////////////////////////////////////////////////////////////
        -:  677:
        -:  678:    /*
        -:  679:    else{
        -:  680:
        -:  681:      temphand[z]=cardDrawn;
        -:  682:      state->handCount[currentPlayer]--; //this should just remove the top card (the most recently drawn one).
        -:  683:      z++;
        -:  684:
        -:  685:    }
        -:  686:    */
        -:  687:
        -:  688:  }
        -:  689:
        -:  690:  /*
        -:  691:  while(z-1>=0){
        -:  692:
        -:  693:    state->discard[currentPlayer][state->discardCount[currentPlayer]++]=temphand[z-1]; // discard all cards in play that have been drawn
        -:  694:    z=z-1;
        -:  695:
        -:  696:  }
        -:  697:  */
        -:  698:
        1:  699:}
        -:  700:
```

Smithy Card (cardtest2.c)

```
File 'cardtest2.c'
Lines executed:98.15% of 54
Branches executed:100.00% of 6
Taken at least once:83.33% of 6
Calls executed:97.44% of 39
```

```
File 'dominion.c'
Lines executed:20.76% of 554
Branches executed:23.37% of 415
Taken at least once:14.46% of 415
Calls executed:10.64% of 94
```

My cardtest2.c file was created to test the smithy card. Although my blocks executed coverage on this function say 67%, I was fully responsible for this lack of coverage. In my bug that I introduced in assignment 2, I made the smithyCardEffect function not able to ever execute the for loop which draws the three cards that the smithy card allows. Without me introducing this bug I'm confident that my statement coverage and branch coverage would have been 100%. I feel that I could have done a better job of testing the boundary coverage of this function by giving the function a different game state.



```
        -:  777:}
        -:  778:
function smithyCardEffect called 1 returned 100% blocks executed 67%
        1:  779:void smithyCardEffect(int currentPlayer, int handPos, struct gameState *state)
        -:  780:{
        -:  781:
        -:  782:  int i;
        -:  783:
        -:  784:  ////////////////////////////////////////////////////////////////////
        -:  785:  // BUG #4 - The player will not draw any cards instead of drawing 3
        -:  786:  ////////////////////////////////////////////////////////////////////
        -:  787:
        -:  788:  //+3 Cards
        1:  789:  for (i = 0; i < 0; i++)
branch  0 taken 0%
branch  1 taken 100% (fallthrough)
        -:  790:  {
    #####:  791:    drawCard(currentPlayer, state);
call    0 never executed
        -:  792:  }
        -:  793:
        -:  794:  //discard card from hand
        1:  795:  discardCard(handPos, currentPlayer, state, 0);
call    0 returned 100%
        -:  796:
        1:  797:}
        -:  798:
function great_hallCardEffect called 0 returned 0% blocks executed 0%
    #####:  799:void great_hallCardEffect(int currentPlayer, int handPos, struct gameState *state)
```

Great Hall (cardtest3.c)

File 'cardtest3.c'

Lines executed:98.65% of 74

Branches executed:100.00% of 6

Taken at least once:83.33% of 6

Calls executed:98.51% of 67

File 'dominion.c'

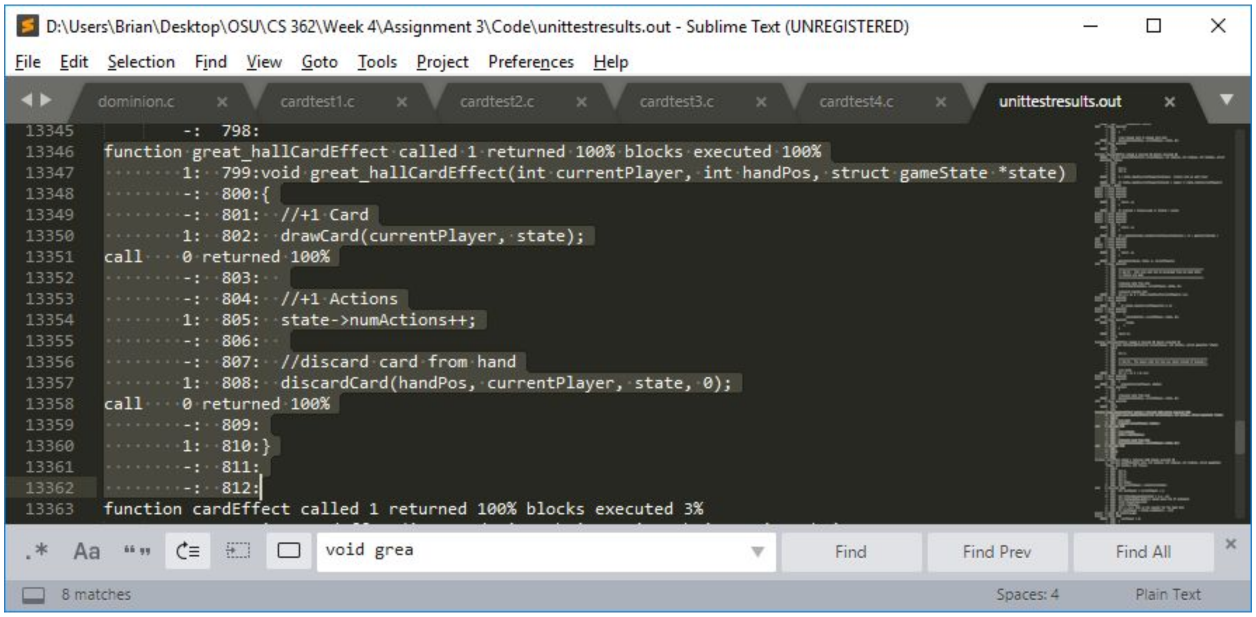
Lines executed:23.10% of 554

Branches executed:27.23% of 415

Taken at least once:16.87% of 415

Calls executed:11.70% of 94

My cardtest3.c was created to test the great hall card. My test received 100% statement and 100% branch coverage of the great_hallCardEffect() function. I didn't introduce any bugs into this function in assignment 2 so I'm not surprised I got full coverage because the function is very simple. I wouldn't even know how to introduce a test to test the boundary coverage better because the function is so simple. I would like to see this card being testing in a system test where it may introduce a problem that a unit test might not be able to see. My dominion.c coverage as a whole surprisingly, got 23.10% statement coverage and 27.23% branch coverage, which is higher than a lot of my other tests on dominion.c as a whole


D:\Users\Brian\Desktop\OSU\CS 362\Week 4\Assignment 3\Code\unittestresults.out - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
dominion.c
cardtest1.c
cardtest2.c
cardtest3.c
cardtest4.c
unittestresults.out
void grea
Find
Find Prev
Find All
8 matches
Spaces: 4
Plain Text

```
        -:  798:
function great_hallCardEffect called 1 returned 100% blocks executed 100%
        1:  799:void great_hallCardEffect(int currentPlayer, int handPos, struct gameState *state)
        -:  800:{
        -:  801:  //+1 Card
        1:  802:  drawCard(currentPlayer, state);
call    0 returned 100%
        -:  803:  
        -:  804:  //+1 Actions
        1:  805:  state->numActions++;
        -:  806:  
        -:  807:  //discard card from hand
        1:  808:  discardCard(handPos, currentPlayer, state, 0);
call    0 returned 100%
        -:  809:
        1:  810:}
        -:  811:
        -:  812:
function cardEffect called 1 returned 100% blocks executed 3%
```

Council Room Card (cardtest4.c)

File 'cardtest4.c'

Lines executed:98.36% of 61

Branches executed:100.00% of 6

Taken at least once:83.33% of 6

Calls executed:97.83% of 46

File 'dominion.c'

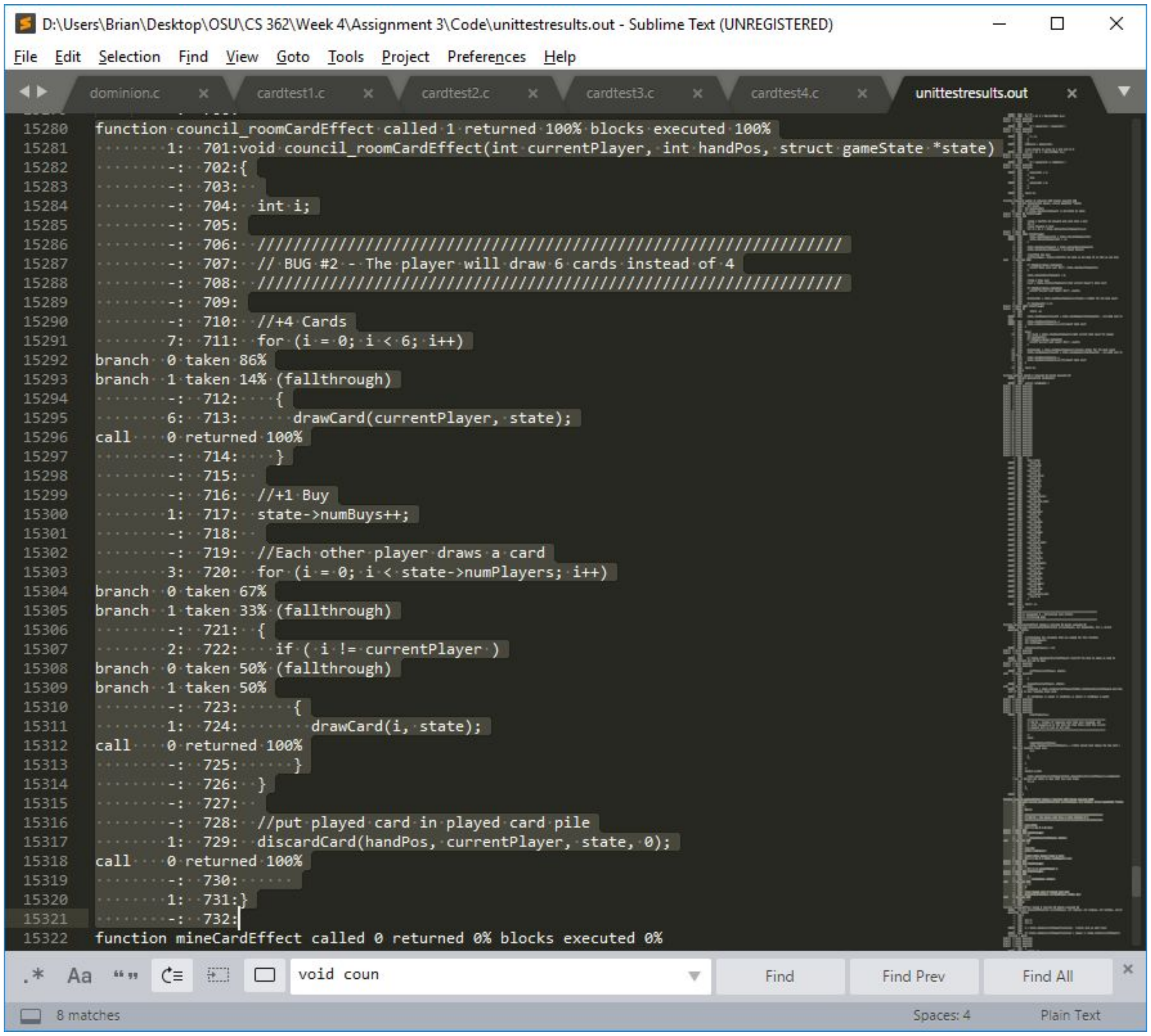
Lines executed:23.47% of 554

Branches executed:25.30% of 415

Taken at least once:16.63% of 415

Calls executed:13.83% of 94

My cardtest4.c was created to test the council room card. My test recieved 100% statement coverage and 100% branch coverage. I feel confident that I fully tested this card being that the only bug I found was the bug that I introduced in assignment 2. I am also impressed that I got full coverage by making one call as there are many things that this card does. Of course problems could occur when this function makes a call to another function but I believe that this card function is well covered.


D:\Users\Brian\Desktop\OSU\CS 362\Week 4\Assignment 3\Code\unittestresults.out - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
dominion.c
cardtest1.c
cardtest2.c
cardtest3.c
cardtest4.c
unittestresults.out
void coun
Find
Find Prev
Find All
8 matches
Spaces: 4
Plain Text

```
function council_roomCardEffect called 1 returned 100% blocks executed 100%
        1:  701:void council_roomCardEffect(int currentPlayer, int handPos, struct gameState *state)
        -:  702:{
        -:  703:
        -:  704:  int i;
        -:  705:
        -:  706:  ////////////////////////////////////////////////////////////////////
        -:  707:  // BUG #2 - The player will draw 6 cards instead of 4
        -:  708:  ////////////////////////////////////////////////////////////////////
        -:  709:
        -:  710:  //+4 Cards
        7:  711:  for (i = 0; i < 6; i++)
branch  0 taken 86%
branch  1 taken 14% (fallthrough)
        -:  712:    {
        6:  713:      drawCard(currentPlayer, state);
call    0 returned 100%
        -:  714:    }
        -:  715:
        -:  716:  //+1 Buy
        1:  717:  state->numBuys++;
        -:  718:
        -:  719:  //Each other player draws a card
        3:  720:  for (i = 0; i < state->numPlayers; i++)
branch  0 taken 67%
branch  1 taken 33% (fallthrough)
        -:  721:  {
        2:  722:    if ( i != currentPlayer )
branch  0 taken 50% (fallthrough)
branch  1 taken 50%
        -:  723:      {
        1:  724:        drawCard(i, state);
call    0 returned 100%
        -:  725:      }
        -:  726:  }
        -:  727:
        -:  728:  //put played card in played card pile
        1:  729:  discardCard(handPos, currentPlayer, state, 0);
call    0 returned 100%
        -:  730:
        1:  731:}
        -:  732:
function mineCardEffect called 0 returned 0% blocks executed 0%
```

Unit Testing Efforts

unittest1.c

In unittest1.c, I tested the function gainCard(). I created 5 tests for this file. In this file, the first thing I tried to do was hit all the branches. I made three separate tests that set the toFlag to 0, 1, and 2, which made the card go to the discard pile, to the deck, and to the hand. Next, I made a call with the a supplyCount as 0 and see if the function would still draw a card. Lastly, I made sure that the supplyCount for a card pile successfully decreased by one after gaining that respective card. I made sure to verify all the tests by checking if the affected variables were the right amount after each test.

unittest2.c

In unittest2.c, I tested the function buyCard(). I created 5 tests for this function. I started this test case by creating 3 tests that all individually set the number of buys, province supply count, and coin amount to 0 while the others where at 10. This covered the first three if statements that verify that the information is correct. Next, I verified that the number of buys, province supply count, and coin amount all decreased by the appropriate amount after a buy had occured. Lastly, I checked if buying a province increased the current player’s victory points and also that the other player’s victory points didn’t increase as a result.

unittest3.c

In unittest3.c, I tested the function isGameOver(). I created 3 tests for this function. I first set the province supply to 0 to see if the game would end. Next, I set three supply piles, none which were the province pile, to 0 and tested if that function returned a 1. Lastly, I set all the supply piles to not equal 0 and checked if the function returned a 0.

unittest4.c

In unittest4.c, I tested the function shuffle(). I created 3 tests for this function. First, I set the current player's deckCount to 0 and tried to shuffle the deck. I checked to see if the function would return a -1. Next, I checked if the deck was actually being shuffled. I saved the deck before the shuffle and then compared it to the deck after the shuffle to see if the cards were actually in different positions. Lastly, I checked to see if the player had the same amount of cards in the deck before and after shuffling their deck.

cardtest1.c

In cardtest1.c, I tested the adventurer card. I created 5 tests for this card. First, I checked to see if the player would actually draw 2 cards after playing this card. Next, I made sure that playing this card didn't affect any of the victory piles or the kingdom piles. I also checked to see if the 2 cards that the player drew were actually treasure cards and not another type of card. Lastly, i checked to see if the other player was affected by playing this card by checking the other player's deckCount and handCount.

cardtest2.c

In cardtest2.c, I tested the smithy card. I created 4 tests for this card. First, I checked to see if the current player drew three cards and that those cards were from their own deck. Next, I made sure that playing this card didn't affect any of the victory piles or the kingdom piles. Lastly, i checked to see if the other player was affected by playing this card by checking the other player's deckCount and handCount.

cardtest3.c

In cardtest3.c, I tested the great hall card. I created 6 tests for this card. First, I checked to see if the current player drew one card and that the card was from their own deck. I checked the other player’s deck to make sure nothing happened to their deck and hand count. Next, I checked to see if the current player’s number of actions increased by one compared to what they had before. I also made sure that playing this card didn’t affect any of the victory piles or the kingdom piles. Lastly, I checked if the either player’s victory score was affected by having this card in the current player’s hand or deck.

cardtest4.c

In cardtest4.c, I tested the council room card. I created 5 tests for this card. First, I checked to see if the current player drew four cards and that those cards were from their own deck. Next, I tested whether other player’s drew a card as a result of playing this card. I also checked to see if the current player’s number of buys increased by 1 as a result of playing this card. Lastly, I made sure that playing this card didn’t affect any of the victory piles or the kingdom piles.