Test Report

Back End Tests

ColoredGraphSolverTests:

testSolver_MonochromaticGrid(): The rationale for this test was to check if the solver would be an empty list if the graph was already one color. When running this test, the solver did have a list that was empty of hints.

testSolver_1-3(): In these tests, the goal was to make sure that the solver would return a winning outcome for multiple levels. In each of the three different levels that were tested, the solver was able to successfully win the game, meaning that the solver was working correctly.

testSolver_ManyChanges(): The rationale for this test was to see if an exception would be thrown if there were too many hints to produce with the solver, as it could take a very long time if there is no minimum amount of steps for the solver to stop at. With this test, it was evident that the solver would produce an exception.

ColoredGraphTests:

testColoredGraph_Default(): The rationale for this test was to test the default constructor for the ColoredGraph class, and so the vertex set should be empty and so should the colorIds. The outcome of this test was that the ColoredGraph default constructor produced an empty graph.

testColoredGraph(): The rationale for this test was to test if constructing a graph that has already had vertexes added to it would have all of the vortexes in the VertexSext. Every vertice that was supposed to be in the VertexSet was tested, and every one was present in the set.

testGetNeighbors(): The rationale for this test was to check if the neighbors of a vertex would be as expected based on the logic of the game. Two different vertices were tested using an already filled in graph, and the list of neighbors contained the appropriate RectangleGridCells.

testAddVertex(): The rationale for this test is to check that, when adding a vertex to a graph, it appears in the vertex set. The outcome of this test showed that they were being added to the set.

testAddVertex_AlreadyExists(): The rationale for this test was to check if adding a vertex when it already exists in the graph should throw an error. The program did return an error, as expected.

testAddVertex_InvalidColor(): The rationale for this test was to check if adding a vertex with a color that does not appear in the colorIds should let that behavior go through. The program did let it go through as expected.

testPruneGraph(): The rationale for this test was to make sure that pruning the graph condensed adjacent cells with the same colors into one vertex. After running this test, the vertex set only contained three vertices, as expected, since there were three groups of color.

testAddEdge(): The rationale for this test was to make sure that when adding an edge to a graph, the neighbors of a vertex will contain the vertex that is being added to it. The outcome of the test was that this was true.

testRemoveEdge(): The rationale for this test was to make sure that when removing an edge from a graph, it no longer belongs in the vertex set. When running this test, we found that the vertex was no longer in the set.

testGetVertexColor(): The purpose of this test was to make sure that getting the vertex color would return the proper colorId. This test resulted in the proper colorId, meaning that the function was working.

testSetVertexColor(): The purpose of this test was to make sure that setting the vertex color to a new color would be successful and result in a new color for the vertex. The test returned true, so the method was successful in setting the color.

testGetNumberOfVertices(): The purpose of this test was to ensure that the number of vertices was correct when calling this method. The outcome of the test was true, meaning that the method was functioning as expected.

testBuildGraphWithAdjacency(): The purpose of this test was to ensure that when calling the method buildGraphWithAdjacency(), the graph would have edges between any cell that is next to each other based on the row and column. By testing all of the cells in a graph, we found that the outcome was true, meaning that the method functioned properly.

testColorFloodFill(): The purpose of this test was to ensure that the color flood fill for colored graphs was successful. By testing all the cells that should be changed when this is called on a particular vertex in a graph, it showed that the method was working properly and the color was spreading to neighboring cells with the same color.

testGetColorIds(): The purpose of this test was to ensure that getting the colorId would be contained in the set of integers representing the colorIds and that colorIds that were not in the graph should not be in the list of integers. The outcome of this test was true, meaning that the method was providing the correct list of colorIds.

ColorRepositoryTests

testConstruct(): the ColorRepository should be initiated with 4 default colors in there. The purpose is to check if the object is initiated correctly.

testListColors_Default(): after adding a new color, that color should show up in the repository. The purpose of this is to test if listing colors is working correctly.

testListColors_Ids(): test if listing color by a list of Ids return the correct colors. The purpose of this is to test if this API behaves correctly.

testAddColor(): test if adding a color behaves correctly. The color should then exist in color repository

testAddColor_AlreadyExists(): test if adding a color behaves correctly. Since the added color is already within the repository, it should throw an error as expected.

testGetColor(): test if retrieving a color by its id works.

ColorTests

testGetRValue(): The purpose of this test was to ensure that the method would return the correct r-value for the color. The outcome of this test was true, meaning that the method was working properly and returning the correct value.

testGetBValue(): The purpose of this test was to ensure that the method would return the correct b-value for the color. The outcome of this test was true, meaning that the method was working properly and returning the correct value.

testGetGValue(): The purpose of this test was to ensure that the method would return the correct g-value for the color. The outcome of this test was true, meaning that the method was working properly and returning the correct value.

testEquals(): This test was to make sure that the equals method for colors was working, as this is very important for the entirety of the program. By running this test, we found that the method was functioning properly and colors that were supposed to be equal were, while colors that should not be equal were not equal to one another.

LevelBuilderFactoryTests

testRegister(): The purpose of the test was to make sure that the register method would properly create a graph if the LevelType existed in the LevelType class. This test is testing both the register and createLevelBuilder, as these work together in the LevelBuilderFactory. The outcome of this test was that there was no error thrown, meaning that the register and createLevelBuilder were working properly in this case.

testCreateLevelBuilder_NoKey(): The purpose of this test was to see if, when no LevelType is registered, an error would be produced since the BuildFactory should not be able to build a LevelBuilder of a type it does not know. In this case, an exception was produced, meaning that the method was functioning properly.

LevelRepositoryManagerTests

testLoad(): The purpose of this test was to ensure that loading a level would work properly when it is in the level repository. To test this, the level was loaded and all key components were checked, with the graph having a correct amount of vertices, number of maximum moves, current color, and that the level would be solved when using the hints. The outcome of this test was true, meaning that loading the level was successful.

testLoad_UnknownFile(): The purpose of this test was to ensure that an exception was produced if the LevelInfo did not already exist in the repository. The outcome of this test was that an error was produced, meaning that the method was functioning as expected.

testSaveBuild(): The purpose of this test was to ensure that, when saving a LevelBuilder object, it is added to the LevelInfo, meaning that it is now in the repository of saved levels. This test showed that it was in the LevelInfo list, meaning that save was working properly.

testSaveHint(): The purpose of this test was to ensure that, when saving a LevelHint object, it is added to the LevelInfo, meaning that it is now in the repository of saved levels. This test showed that it was in the LevelInfo list, meaning that save was working properly.

MoveTests

testGetColor(): The purpose of this test was to ensure that getting the color of a move was working properly given a specific move. The outcome of this test was true, meaning that the method was behaving properly.

testGetRow(): The purpose of this test was to ensure that getting the row of a specific move would produce the correct integer. In the outcome of this test, we saw that the method was behaving properly.

testGetCol(): The purpose of this test was to ensure that getting the column of a specific move would produce the correct integers. In the outcome of this test, we saw that the method was behaving properly.

RectangleGridCellTests

testAdjacentTo(): test if the implemented adjacentTo() of RectangleGridCell behaves correctly. 2 cells next to each other in terms of their positions in the grid should be adjacentTo each other.

RectangleGridLevelBuilderTests

testLevelBuilder(): The purpose of this test was to ensure that the constructor for RectangleLevelBuilder was working properly and that the number of rows and columns was correct, as well as the colors in the colorIds was also functioning. This test proved to be true.

testGetCurrentColor(): The purpose of this test was to ensure that the current color of the builder was initially red, and that when switching the color, the current should then be the color that had been switched to. The outcome of this test was true, meaning that both cases were working.

testSetColor(): The purpose of this test was to ensure that setting the color of a given cell in the LevelBuilder would change the color of the cell. The test proved that setting the color of the cell was successful.

testChangeGridSize(): The purpose of this test was to ensure that the method to change the grid size was working and altering the size of a LevelBuilder's ColoredGraph, which proved to work correctly.

testRestart(): The purpose of this test was to ensure that restarting the LevelBuilder would return all cells to the default color red. The method performed this correctly for all cells that had been changed.

RectangleGridLevelTests

testConstruct(): test if the loaded level is of correct size and have the right number of moves. The rationale is to test check if the level is properly loaded in. Further testing is conducted in other tests

testGetColorAt(): test if getting the color of a cell(vertex) function properly.

testSwitchColor(): test if switching color is functioning properly. The level should let the user switch to a different color.

testSwitchColor_Invalid(): test if switching color is functioning properly. The level should let the user switch to a different color even if it is not inside the level (will be a bad move on the user side, but it should still works)

testGetCurrentColor(): test if getting current color is behaving correctly. Should return the current color of the level

testPlay(): test if play() is behaving correctly. The floodfill should happen from the target vertex. The rationale is that we have to test if playing API (core of the game) is working as expected.

testPlay_noMovesLeft(): When there is no more left, the level should throw an IllegalArgumentException saying the user can't play anymore because they ran out of moves.

testGetColors(): test if getting the colors that are currently in the level behave correctly. The code passed the test by returning the expected colors.

testGetHints(): Test if getHints() return the right hints to the level. The code gives the right hints as expected

testLevelState_Win(): Test the getLevelState() API. It returns WIN when the level only has 1 color left as expected.

MementoCaretakerTests:

testUndoable(): This test is to check to see if there are any mementos in the caretaker that can change the state of an object back to its previous state. When running this test, we find that when there is nothing in the caretaker, the function returns false as expected and true when there is a memento object in the caretaker. Also, when the caretaker uses up all of the memento objects, it returns false for this function.

testUndo(): The purpose of this test was to ensure that the undo method for the caretaker works properly and pops the memento objects off of this stack. The undo function returned the proper states based on the order they were added, meaning that this method is functioning properly.

RectangleHintInputLevelTests

testConstructor(): This test ensures that the constructor functions properly. When creating a LevelHint object, we always use the underlying graph of a LevelBuilder object. When running this test we compare the underlying graphs are identical, and that the two objects have the same dimensions. We also ensure that they have the same color palette. The function returns true in all cases.

testSwitchColor(): This test ensures that the current color field of a LevelHint object can be switched. We use two distinct colors for this test so that at least one implies a change in the data field.

testPlay(): This test checks to see that the play move functions properly by flood filling. We color the first row and column of the LevelBuilder object homogeneously with every color in the color repository. We create a LevelHint with the underlying graph of the LevelBuilder and play a move in the top left corner with a distinct color. We then check to see that the first row and column of the LevelHint are that distinct color. We restart the builder and level, and repeat with the next color in the repository.

testResetGraph(): This test ensures that the reset functionality of the LevelHint is preserved, that is, a HintLevel should be able to return to its original state at construction. We color a LevelBuilder haphazardly and construct a LevelHint. We play a series of moves on the LevelHint and call reset. We then validate that the entries of the LevelBuilder and the LevelHint are the same.

testGetLevelState(): This test ensures that a LevelHint knows when it is won. We create a non-winning LevelBuilder and use its graph to create a LevelHint. We check that it has an "ONGOING" LevelState. We then create a winning LevelBuilder and use its graph to create a LevelHint. We check that it has a "WIN" LevelState.

CommandTests:

testReset(): This test ensures that, after a series of command invocations, the internal command queue of the CommandInvoker is cleared by reset.

testGetCommandQueue(): This test ensures that, after a series of command invocations, the internal command queue of the CommandInvoker maintains each command in the correct order.

testGetCommandQueue(): This test ensures that CommandInvoker invokes the Command object, thereby calling the Command object's execute method. We use a test stub for the RectangleHintInputLevel class that maintains its own record of the commands executed. We use the PlayMoveCommand as our command as it is the only Command object in the program. We checked that the Move object for the PlayMoveCommand was the same as that which was stored in the CommandInvoker queue, and as that which was stored in the test stub's internal move list.

Front End Tests

Testing Menu View and Controller:

In order to test the menu, we first started by examining the layout of the window. This is to ensure that the view is working properly and the layout is as expected. Since the window looks as expected, we know that MenuView is functioning correctly. Furthermore, we can see that all levels are being loaded from the level repository properly onto the menu. Next, to test the controller of the menu, we click on a level. This not only informs us as to whether the stage is

switching the scene correctly, but whether or not the controller for clicking on a level is working correctly. It also shows if the levels are being properly loaded. Since the scene successfully changed to the level and the level is being properly loaded, we know that the controller is working and the view is working. After this, we test the controller method for clicking on the create button. This should bring us to the scene for creating your own level and will show us if the handling of this event is working properly. Since the view does change to the level builder scene, we know that the controller is functioning properly. Furthermore, on the menu screen, we can check to see if the create button is positioned as expected, at the bottom of the lists of levels. In order to test this, we first had an initial level repository size of 3. This meant that the create button should be in line and next to the third level. Since this was the case, we know that this part of view is working. Next, after saving a new file, we check to see if the create button is underneath the third level button. This will prove to us that the create button is being properly placed depending on the amount of levels in the repository. Since doing this produced a proper menu, we know that this part of the view is also working.

Testing Level View and Controller:

The first part of testing the level view was to make sure that everything was in the proper place on the screen. The expected screen should have the grid of colors at the top of the window above any of the buttons. Furthermore, it should have grid lines and be connected to one another, filling the screen. After clicking on a level, we found that the grid was in the proper place, meaning that it is being placed correctly in the view and in the right style. Also, all colors are in the proper place based on the level file, meaning that the view is correctly processing the ColorGrid. Then, looking at the control panel on the bottom of the screen, we make sure that all of the option buttons, colors, and number of moves remaining are present and in the proper place. This will allow us to know if the level view is properly formatting the page and if the sizes are matching the desired size. Since all the buttons are present and in the proper place, we know that the level is rendering the view properly. Next, we test each of the different control panel buttons to make sure that their controller is working properly. Starting with the exit button, we click the button and expect for the scene to switch back to the menu. When doing this test, we find that the controller for this button is working properly. Next we test the restart button. The expected outcome for this restart button is that the grid is reverted back to its original state with all the colors in the grid changing back to their original color. After doing this with both one move made, no move made, and many moves made, we find that the restart button controller is working properly and setting the grid back to the proper state. Then, we test the undo button, which we expect to go back to the previous grid state if a move has been played already. This will show if the undo button controller is functioning properly. When doing this test, for one and many moves, we find that the undo button controller is working successfully and changing the grid state back to its previous one. Also, we test the undo button when there have been no moves already made. When doing this test, we expect that an alert will be shown that no moves have been played prior to the request. This will let us know if this exception is being properly

accounted for. After running this test, we find that the controller for this scenario is working properly. Lastly, we test the hints button. We expect that this event will cause a window to appear with hint options for the amount of hints there are stored. Then when clicking one of these hints, it will change the color of a grid cell and flash. Running these tests proved that this was working properly and successfully reading the hints.

Then we test the number of moves remaining that appears next to the option buttons. We expect for this to be an integer that decreases every time a move is played. Since this successfully works, we know that the view is properly changing the label of how many moves are remaining. Furthermore, when the number of moves remaining is zero, we expect for an alert to appear with a restart and a move to menu button to appear. This works successfully, showing that this exception is being accounted for by the view. Furthermore, these buttons do appear in the alert window and function properly as they did in the prior tests. For the colors being chosen, we test this by making sure that when a color is chosen from the selection and then a color is clicked, it changes to the proper color that has been selected. After testing this for each color, we find that it works as desired, with the selected color properly appearing on the screen.

Then, we test the function of the level view when playing. To do this, we select a color and then click a grid cell. We expect that it will change to the proper color and flood fill adjacent cells with the same color. This will show if the view is changing after a move is being played. The tests run for this, when there are cells around to be filled and when there are not, show that the view and controller are functioning properly when a move is being played. Then we test that a window is shown when the player wins. To do this, we play the correct moves in order to win the game and find that a window saying we did win is shown. This window is expected to have both restart and move to menu buttons like the window appearing after losing. These buttons function the same way, so we know this is working in both the view and the controller.

Testing Level Builder View and Controller:

Similar to the level tests, we first want to make sure that the grid and buttons are in the correct place for the builder view. The color grid is expected to be above the option buttons and to the right of the color buttons, since we have added buttons to the bottom of the screen. Since this occurs when opening the level builder scene, we know that this is working properly. After this, we check to see that the exit, restart, and save are at the bottom right of the screen. Since these are in the proper place, we know they are being added to the view properly. Next, we check that the resizing text boxes and buttons are next to the other buttons, which we find that they are and the text boxes allow the user to write into them. After this, we check to see that the color choices are next to the grid and not with the other buttons. This is what appears in the view, so we know this is working properly.

The same tests for the exit, restart, and color choices were repeated for the builder, meaning that the controller was working properly for these buttons and performing the correct tasks. For the save button, we test to see if the level will save to the repository successfully and move the scene back to the menu scene. This will allow us to know if the controller for the save

button is properly working and performing the right tasks. This turns out to be true, with the level saving properly and moving the scene back to the menu. For the resizing part of the view, we tested this by changing the size of the grid to both smaller and larger. When clicking the resize button after typing in the proper sizes, we expect the number of rows and columns to be the equal to the correlating inputs. After doing the tests, we found that the grid is properly resizing and resetting the colors back to the default red.

Lastly, to test and make sure the grid is not flood filling as it would in a level, we first select a color and click on the screen. Even if an adjacent cell is the same color, we expect only the cell that was clicked to change colors. After doing this test, we found that there was no flood filling of the color occurring, meaning that this was functioning properly.

Testing Hint View and Controller

For basic form, all buttons are in the correct position. We expect the restart button to be located on the bottom left corner with the exit button directly above it. There will always be four colors along the bottom row to the right of these buttons. From left to right: red, green, blue, and light blue. Above these buttons is the Kami board which is in accordance with both the dimensions of the LevelBuilder, and the coloring.

Flood fill works as expected, and respects inefficient colorings. We can change the current color as many times as we'd like, the last click will be the color that is filled on the board. The reset button resets the board to its starting state. The exit button transfers the user to the menu.

If we exit the hint view, having found a solution or not, we can re-enter the build view from the menu and will be faced with an entirely red five-by-five board. This is the starting state of the build view. From there, we can build an entirely new level that is too complex to solve and proceed to the hint view with this new level. This step-by-step summary is to illustrate that the backend state of the underlying graphs is resetting. Moreover, the stored commands reset.

We find that any sequence of moves that solves the board is exactly the same as the hints that are presented when the level is loaded. If we exit the hint view and return to it with a new level, there will be no record of past hints. If we reset the hint view, there will be no record of past hints.