Technical/Software Manual

Ishido Game

Sujil Maharjan

**Activities:**

1. **StartPageActivity:**

This activity is the start page of the application. It will display the user that this is the “Ishido Game” and let the player start the game.



1. **BoardActivity:**

This activity handles the main game. It displays the main board in which the user can play and reports lets the user choose the tiles and report the scores. This activity communicates with the logic of the game so that it can display accordingly.

**Model Classes:**

1. **Board:**

This class holds the model for the board. It consists of the 2 dimensional array that holds the values of the board. It provides the functions to fill the tile to the board, to check if certain location in the table is available and to calculate the total score of the move if it is placed in certain location.

1. **Deck**

This class holds the deck information of the tiles. It provides the option of having 72 tiles in the board. It also consists of a 2 dimensional array that connects between the color and symbol of the tile.

1. **TileInfo**

This class holds the general information of one tile. It holds the information such as the color and the symbol of the tile with their numeric equivalent.

1. **TileTree**

This class holds the information of a move. It is the node for the different search tree algorithms. TileTree holds the TileInfo, the coordinates of where this tile belongs to, the total score until this tile, and the parent tile of the current tile. It helps us track the entire path for the search tree algorithms.

1. **TableCoordinates**

This class holds the information of coordinates of the table. Connected with TileInfo, it becomes very useful in pointing to different locations in the table.

**AI:**

The Algorithms are used in following functions:

1. performDFS(): It uses the depth first search algorithm to calculate next tile location according to the algorithm.
2. performBFS(): It uses the breadth first search algorithm to calculate the next tile location.
3. performBestFS() : It uses the best first search algorithm to calculate the next tile location.
4. performBNB(): It uses the branch and bound algorithm and calculates the final and best path, given the branch cutoff.

**BUG REPORT:**

1. The app cannot calculate greater than 4 tiles for branch and bound search. It will go to a long loop. It is because it will have a lot of nodes in the tree to go through for more than 4 tiles.
2. Depth first search does not back track properly. It tries and fails. The error is with connecting previous tiles. It doesn’t have a good design. If it follows the design of Best First Search, it will be able to backtrack properly.

**FEATURES:**

1. **Implemented:**
2. User can select what type of search to do.
3. Row and column indexes displays aside the board for easiness in tracking.
4. The newly generated tile blinks rapidly grabbing the attention.
5. For branch and bound, if the user selected search entire tree and also sets the branch cut off, the priority is given to the branch cut off number rather than the entire tree.
6. Displays the short message if the process ends indicating that the game ends.
7. **Not implemented:**
8. I think everything is implemented.

**LOG REPORT:**

2/14/2016

* Added the FileAccess class that will handle and parse the file given to retrieve the strings of Board, stock and score.
* Added a new function in Board that will handle the board info parsed from the FileAccess. Also, since user does not have to click, I changed the board initially writing to screen to handle the filled tiles
* Made a stockQueue that is the linkedlist which handles the function of the queue of given stock. It is because we need to handle the given tiles from the first to last.
* Made the spinner and button for search options. Id is searchOK (button), searchHandle(function), searchChoice(Spinner)
* A bug found for automated system. The fillTile() checked for all of the side tiles to verify but it did not check if the tile selected is empty or taken.
* Now when we change the type of search we want to do, searchHandle () is called and it will check what type of search we are choosing. If the search type is being set for the first time, nothing happens. But if the search type is set for more than one time, then, the view of the table layout is deleted and new Board() is created in order to refresh the whole board for the new type of search.
* refreshSearchTable() is made for refreshing the search table everytime user wants a different types of search so that we can start from the beginning again. It creates new board, deck, searchModel, stock, and player.
* Finally done with DFS
* IMPORTANT CHANGE: Making a separate class for SearchModel creates a LOT of hastle and lot of annoying things to take care of for simple things. The View became very hard to update since we needed a unique condition for almost all of the variables and it was VERY VERY REDUNDANT.

2/15/2016

* BFS : lots of trouble. Tried doing the map. Got into a lot of hastle. Redid everything by replacing it with a new class TileTree that holds the TileInfo and TableCoordinates. This helps track the previous TileTree object that eventually helps us retract from the overall tree to the root node.
* Still have a lot of problems like tiles being replaced.
* POSSIBLE SOLUTION: Try to figure out when to fill the tile in the board and when not to. Only fill in the board if it is being placed in certain place like permanently. The guess of all of the tiles always being in a proper place is so wrong! Check for cell availability canFillTile() for everytime.
* Another error: check while assigning the master. It is not going through correctly. T3 is having T1 as master instead of T2.
* Assigning to the master the new tile from the queue was placed was wrong. Whenever we take a new tile from the queue, we refresh the table. It makes sense because it is actually a new search.
* Then, check for the parent tiles of the newly taken tile from the queue and place them on the board. Once placed, place the new tile. Then only check if we need to work with the new tile.
* Removed the stack of stationary files since it didn’t really make sense. We only push and pop one element. So, I changed it to a TileTree object.
* ERROR: The tiles were not being placed correctly. Once the available location was found, lets say (5,4), the column never started from 0 again even for next row. So, fixed it by changing the value of starting point of column to 0.
* TODO: Update player score accordingly.

2/19/2016

* - Added the bestFS() function.
* - Changed the name of the stack from dfsStack to searchStack hoping to
* make it usable for more search types.
* - Added an ArrayList that holds the Stack of TileTrees different for
* different scores. So, lets say we have 5 TileTrees in the board where we
* can get score of 2, the these TileTrees will be stored in a Stack in
* index 2.
* - This helped in pushing the TileTrees in open stack according to the
* order of score. (Taking first highest score).
* - Made a path stack which will hold the optimal end state path.
* CONCEPT: Take a new tile -- Find the available locations of it in the
* board -- If available, sort it according to the score and get the first
* location with highest score. -- Push it to the path -- Print it -- If
* no new location is found, then put the new tile back to stock and make
* the new tile as popped tile from open tile tree stack.

2/21/2016

* Tried a different approach for branch and bound search.
* For it we need to analyze all of the solution node (last leaf nodes in the given tree). So, I tried calculating the main algorithm first and then the printing part. It would take a long time if we have branch more than 4 since it is going to have a lot of them in the queue.
* Make performBNB() function where it will do the breadth first search until you find the solution node branch level. Once it reaches the solution node branch level, it will go through all of the solution nodes and then pick out the one that is going to produce the highest score overall.
* Added total score upto the particular node in the TileTree function. This helps us track the total score until that point. Helpful in all of the searches but just implemented in Branch and Bound. (too late to change it in the others).