**Prog 3: 8 Tiles**

|  |
| --- |
| Write a program to play the 8 tiles puzzle.  Partial output from running your program should look like:  Author: Dale Reed  Class: CS 342, Fall 2016  Program: #3, 8 Tiles.  Welcome to the 8-tiles puzzle.  Place the tiles in ascending numerical order. For each  move enter the piece to be moved into the blank square,  or 0 to exit the program.  Choose a game option:  1. Start playing  2. Set the starting configuration  Enter your choice --> 1  Initial board is:  1.  7 8  4 1 5  2 6 3  Heuristic value: 18  Piece to move: 1  2.  7 1 8  4 5  2 6 3  Heuristic value: 16  Piece to move: 4  3.  7 1 8  4 5  2 6 3  Heuristic value: 18  Piece to move: 5  \*\*\* Invalid move. Please retry.  Piece to move: 9  \*\*\* Invalid move. Please retry.  Piece to move: 7  4.  1 8  7 4 5  2 6 3  Heuristic value: 18  Piece to move: 1  5.  1 8  7 4 5  2 6 3  Heuristic value: 16  Piece to move: s  Solving puzzle automatically...........................  1.  1 8  7 4 5  2 6 3  Heuristic value: 16  2.  1 8  7 4 5  2 6 3  Heuristic value: 14  3.  1 8 5  7 4  2 6 3  Heuristic value: 14  4.  1 8 5  7 4 3  2 6  Heuristic value: 12  .  .  .  42.  1 2  4 5 3  7 8 6  Heuristic value: 4  43.  1 2 3  4 5  7 8 6  Heuristic value: 2  44.  1 2 3  4 5 6  7 8  Heuristic value: 0  Done.  The example above illustrates error checking of piece input on move three.  Piece input of 's' switches into "solve puzzle automatically" mode.  Input of 's' can be done at any point.  In the above case there *is* a solution, but in the many cases where there is no solution the program should show the best board found.  At the beginning of the program you also have the option of setting the values to be used in initializing the board.  Both of these are illustrated below:  Author: Dale Reed  Class: CS 342, Fall 2016  Program: #3, 8 Tiles.  Welcome to the 8-tiles puzzle.  Place the tiles in ascending numerical order. For each  move enter the piece to be moved into the blank square,  or 0 to exit the program.  Choose a game option:  1. Start playing  2. Set the starting configuration  Enter your choice -->  2  Some boards such as 728045163 are impossible.  Others such as 245386107 are possible.  Enter a string of 6 digits (including 0) for the board --> 728045163  Initial board is:  1.  7 2 8  4 5  1 6 3  Heuristic value: 16  Piece to move: s  Solving puzzle automatically...........................  All 181442 moves have been tried.  That puzzle is impossible to solve. Best board found was:  1 3 2  4 5 6  7 8  Heuristic value: 2  Exiting program.  **Object Oriented Approach**  You must create your program using an object-oriented approach, writing the main areas of respective functionality in the classes as shown below, using the names I have given.  I suggest you create these classes in the order shown below as well.  You may create additional classes.   1. *Board* Used to store the board pieces, initialize a board, compute the board heuristic value, find possible next moves from any board position.  If you do use a data structure that needs a hash value, you should define that as well.  Unless the board is initialized by the user, the Board class should create a board where the destination pieces are chosen at random.  Seed your random number generator using:             randomGenerator.setSeed( System.*currentTimeMillis*()); so that you get different boards each time you run the program. 2. *TilesDriver* This is the driver for the rest of the program and contains main().  Use this to test the other classes as they are created.  The loop to play the game interactively (or automatically solve the puzzle) is inside this class.  Move validation is controlled from here. 3. *Node* Nodes are what are stored in the SearchTree.  Each node should have a Board as well as elements needed to connect Nodes to each other, as needed. 4. *SearchTree* This implements the A\* state-space search algorithm which uses a *heuristic* as an approximation of the goodness of each board configuration.  If the chosen heuristic is overly optimistic, then that implies less search, but at the possible extent of an optimal solution.  A heuristic is *admissible*if it underestimates the goodness of a position.  This implies extra search, but that search then encompasses an optimal solution. The search tree should include some sort of storage for:    1. The list of all unique Nodes This can be any sort of data structure, but efficient Node lookup is important, since this is the data structure you will use to process and store only new *unique* Nodes that are not already on this list, out of the total possible ~181,442 moves.  You could maintain this as an ordered list by hash number and use binary search, or you could use a HashMap.    2. The ordered list of leaf Nodes This could be stored as an ordered linked list or as a Priority Queue.  Here the ordering is done by heuristic value, with a lower heuristic value being preferable.  The heuristic is the sum of the city-blocks distances of each piece from its desired destination. 5. *Constants* If you have constants shared with multiple classes they should be declared in the Constants class.   **Explanatory Notes**  Below is a copy of the [10-03 class notes](https://www.dropbox.com/s/n6irdec5xuu9krl/10-03%20Class%20Notes.pdf?dl=0) where I explained the process of using the A\* algorithm with the SearchTree collections of all Nodes and leaf nodes:  **Point Values**  Point values for various components are shown below.  Each step must be complete before points can be awarded for subsequent steps.   1. ( 5 points) Program prompts for and receives user input that optionally can be used to set the board 2. ( 5 points) Board is represented and allows making numbered moves to interact with the puzzle. 3. ( 5 points) End of game is detected 4. ( 20 points) Input of 's' at any point starts up solving the puzzle automatically.  Boards that have no solution display the best board. 5. ( 20 points) Boards that do have a solution display the numbered solution in order. |