Kurtis Hausman

Provide a concise answer to each of the following questions. Your answers should **NOT** include code. It should be an English explanation, and each question can be answered in a few sentences. If your answer to one of the questions includes the use of a set, map, or symbol table, include an explanation of why/how you are using it, but do **NOT** explain what the data structure itself does or how it works. For example, if you use a BinarySearchST, do not explain to me how binary search works, how insertion works, etc. Instead, explain, what are the keys, what are the values, and why/how you are using the symbol table in your solution. Each question is worth 5 points.

* Breadth-first search is done on a graph. What is the graph being searched in your solution? What are the vertices of the graph and when is there edge between two vertices?

The graph being searched are all of the possible board configurations generated from the various valid moves beginning from the solved board.

Each possible board configuration is treated as a vertex, and an edge exists between two vertices if you can move directly from one board configuration to the other by applying a single valid puzzle move. The BFS algorithm explores this graph of configurations until it finds the solved state.

* During breadth first search, you need to loop over all neighbors of a given vertex. Explain how your code does this. How does your code know who the neighbors of a given vertex are?

To find neighbors of a board configuration, the code applies each and every valid puzzle move to produce a new configuration. Every configuration you can reach in exactly one move is considered a neighbor. This ensures the BFS systematically explores all directly reachable states from the current board.

* To be able to recover the solution path, breadth-first search requires that for each vertex, you keep track of what vertex you came from when you first encountered it (edgeTo array). How do you keep track of that in your solution? Again, if you use a data structure, make sure to explain what is stored in it and how it is updated/used.

My code is able to discover the solution because I used an object (named ParentInfo) that stores two things for each newly discovered board:

* (1) the board that led to it (its “parent”) represented by predecessors
* (2) the move that transformed the parent board into the new one.

This record is kept in a HashMap where the key is the newly discovered board, and the value is the ParentInfo object. Each time the BFS algorithm encounters a board for the first time, it creates and stores one of these records. Later, once the search finds the solved board, the code can follow the chain of ParentInfo records backwards (from child to parent) to reconstruct the entire path of moves.

* Assuming your breadth-first search has already been performed, how is the solution itself recovered? Again, avoiding explaining line by line. Instead, explain what has been stored in the various variables by the BFS, and how that information is used to construct a solution.

The solution is recovered by working backwards from the starting board to the solved board. The variable ‘current’ holds the randomly generated starting board and we iterate back through the predeccesors (parent-child links) of the starting board until we get to the solved board.

Each link tells me which board came before the current board and which move took you from that previous board to the current one. We reset ‘current’ each time to become its parent after we collect its move, invert it to account for the change of direction and store it in the list of moves (reversedMoves). Inverting is necessary because we are going in the reverse order of what we built the BFS in, so row slides (a,b,c,d moves) will be the opposite of what they originally were.

This way the loop finishes once current equals the start board, indicating that we have built to path from start to solve. We construct the character array solution by iterating through the reversedMoves list.

* At some point you probably used a map/symbol table. Did you try using both a balanced tree (TreeMap) and a hash table (HashMap)? What was the total runtime for P2Test.java when you used a hash table and what was the total runtime when you used a balanced tree? (Simply fill in the table below.) What did you need to change/add to your code to switch from one kind of symbol table to the other? (To get full points, your code must contain both versions with one version commented out as mentioned in the writeup for P2.)

|  |  |
| --- | --- |
| **Symbol table type** | **Total time taken by P2Test.java** |
| Hash table | 27.946 seconds |
| Balanced tree | 41.597 seconds |

The first necessary change in my code came in the constructor for my predecessors (edgeTo) map in the Solver class where I needed to construct a new TreeMap rather than a HashMap.

The second necessary change in my code came in the Board class, where I needed to implement comparable in the class heading as well as implement a compareTo() method for Board objects so that the TreeMap could sort as it is supposed to.

* If your solution does only a single breadth-first search at the beginning, regardless of the number of times solve is called, how did you do it? Focus on how your search and solution recovery differ from the more straightforward BFS described in the book and in class.

I perform a single BFS in the constructor starting from the solved board to precompute the entire all reachable Board configurations along with their predecessor links. Instead of running a new search for every call to solve(), our solver uses this precomputed mapping to quickly backtrack from any given starting configuration to the solved state. This approach differs from the standard BFS we discussed in class, which would typically take a graph for each starting board configuration and do the solving on that graph, figuring out the shortest distance to solved.

Instead we start our BNS from the solved board, creating all configurations only once from the input state and explore until a solution is found. I shifted the heavy computation to the initialization phase which turns each solve() call into a simple lookup and backtracking operation to construct the character array of moves that reach the solved state.