CS-3114 Spring 2014 – Project 3 – Puzzle Solver

Updated 3/29/14 – updates in yellow

Due: Friday April 11, 9pm.
Late Policy as in the syllabus: 10% per day, maximum of 3 days.

Assignment:

For this project you design a solver for the 15-puzzle. The 15-puzzle is a 4x4 grid of tiles, numbered 1-15, and the 16th slot is empty. You begin the puzzle with the tiles arbitrarily ordered, then move tiles by successively sliding an adjacent tile into the empty slot, with the goal of arranging all the tiles into row-major sorted order. Note that half of all possible initial states are solvable. Swapping any two tiles of an unsolvable state will produce a solvable state.

Artificial intelligence and data mining methods for solving such puzzles frequently rely on brute force search of all possible moves. Conceptually, think of the problem as a state transition graph, where a state is a particular configuration of the puzzle tiles and is represented as a graph node, and a transition between states is one tile move and is represented as a graph edge between the neighboring nodes. You search the graph, starting at the initial scrambled puzzle state, and follow edges searching for the goal state in which the tiles are all correctly ordered.

To do this, you will design and implement a Breadth-First Search algorithm and two accompanying data structures: a Hash Table for the node store, and a Priority Queue for search sequencing.

Sample scrambled state:

1	10	15	4
13	6	3	8
2	9	12	7
14	5	-	11

15-puzzle goal state:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	-

Implementation and Design Requirements:

For the purpose of this assignment, we define a state as "**visited**" in the context of BFS to mean that the state "**has been en-queued**" (regardless of whether or not it has been de-queued). This is important to ensure your output matches webcat's.

Puzzle States: Create a representation for a puzzle state. The state should capture the positions of the 15 tiles and the empty slot. It should be efficient in size (e.g. note that you only need 4 bits to represent tile values), because you will create and store many puzzle states during the search process. States should be able to print their grid configuration, and compute neighbor states. A state should be able to compare equality with another state, and compute a hash value for the node store. Thus, you should override the **equals(obj)** and **hashCode()** methods of Java's Object base class for your State. Equals should identify whether 2 states have the same puzzle configuration (and thus are duplicates of the same state). Your computed hash values should take into account the entire configuration of the puzzle as best as possible so as to avoid hash

collisions as much as possible.

Breadth-First Search: Starting at the initial puzzle state, use BFS (OpenDSA 14.3.2) to systematically explore the graph of puzzle states until the goal state is found. Use the rules of the puzzle to dynamically compute the nodes and edges on the fly as you search. That is, you grow the graph as you explore it. Begin with a single node, the initial state of the puzzle. At each state there are at most 4 possible moves that can be made, each leading to a neighbor state that can be computed. Two data structures support this process: the node store and the node sequence queue. As you compute each new state, first search for that state in the node store to see if you have already visited that same state before (to avoid cycles). If not, then add the new state to the node store **and** en-queue it on the search sequence queue for later traversal.

You should compute neighbors of a state in the following order based on the direction the hole moves in: **Up, Right, Down, Left.** This is important to ensure consistency with the graders evaluation. You need to efficiently compare each visited state to the goal state. Once the goal state is visited, you will also need a way to output the discovered solution sequence of moves to go from the initial state to the goal state. To maintain consistency with graders output: for each state transition **A->B** in your solution sequence, **A** must be the first node that pushed **B** onto the queue.

Node Store: Efficiently store the set of puzzle states that have been already visited so far by the search algorithm. You should **not** explicitly store the edges, because neighbor states can be computed dynamically. You should **not** pre-compute the entire state-transition graph as a data structure, because the graph is huge (think combinatorics) and only a small portion is needed to find the way from start to goal.

For the node store, implement a **Hash Table** (OpenDSA 10) with closed hashing and linear probing of step size 1 (OpenDSA 10.7.1). The table size should be dynamic, using the size doubling strategy and appropriate load factor to maintain efficient amortized time. You are required to implement the hash table as a **generic** so that it can store any data type. It should use the hashCode() and equals() methods of the generic type to compute hash keys and test for exact matches. Remember that it is possible for different states to map to the same hash key.

Search Sequence Queue: Efficiently manage the sequence of states to be explored by BFS, determining which of the nodes on the graph frontier to explore next. You will implement the sequence queue using two different Queue variations: **FIFO** and **Priority** You will use **Java's** built in queue classes for these, and you will compare performance of the two variations.

- **FIFO Queue:** A standard queue (OpenDSA 5.11) that simply visits nodes in the order that they were inserted, with amortized O(1) time for en-queue and de-queue operations. This Queue should enable your BFS to find the shortest path. Use Java's **java.util.ArrayDeque<T>.**
- **Priority Queue:** A min-heap (OpenDSA 7.11) that orders nodes according to a priority metric (see distance metric below), with O(log n) time for en-queue and de-queue operations. This Queue should enable your BFS to find solutions more quickly. You must use **java.util.PriorityQueue<T>** with an initial capacity of **1000**. This is important to maintain consistency with the grader's output. You will need to supply a Comparator that compares states according to the distance metric.

Priority Queue distance metric: For node priority ordering, implement a distance heuristic that measures how close a state is to the goal state. Compute the sum of the "**Manhattan distance**"

each tile has to move to get to its sorted position. That is, add the number of rows and the number of columns that each tile has to traverse to get to its final spot. Then sum these distances for all tiles in the state (do not include the empty slot). For the de-queue operation, the priority queue should return the state with the least distance as the next state to visit.

Experiments:

Now, collect some data about the relative performance of the two Queue structures as follows.

Implement another algorithm that generates random solvable initial scrambled states. Start with the goal state, and iteratively apply **N** random moves. Each random move must **not** simply undo the previous random move. Thus, a random move is one of 3 possible moves from the current state. Use Java's **Math.random(),** and be careful that you give equal probability to each of the possible moves. You can control the approximate difficulty of solving the random scrambled state by varying N.

Generate a series of random scrambled initial states by varying **N** from 1 up to some very large number. Run your solver using each Queue on each random initial state. For each Queue on each input, collect data about (1) the length of the solution path found, (2) the number of visited nodes, and (3) the solver run time. Make a line graph comparing these data for the two Queues, with N on the x-axis. You will submit this analysis as part of your report in the Deliverables.

Input and Output:

WebCAT will test your solver by invoking it as follows (this implies a class Solver containing "static void main()" in the default package). Note that you will likely also have other ways to invoke or test your solver for the Experiments etc.

```
$ java Solver <O> <Q> <V> <initial configuration of tiles>
```

where the command line arguments, in main(String[] args), are:

<O> = "V" or "S", for one of the following System.out output formats:

- "V" = Output the numbered sequence of states **visited (en-queued)** by BFS, terminating as soon as the goal node is visited (en-queued), with each state identified by the (row, column) position of the hole.
- "S" = Output the numbered **solution** sequence of states to proceed from initial state to goal state, in forward order, with each state identified by the (row, column) position of the hole, and the move to get to the next state identified by the direction the hole moved and the number on the tile that moved.

<Q> = "F" or "P", for which type of queue to use. F = FIFO queue, P = Priority queue.

<V> = "V" or "Q", for verbosity setting. V= Verbose mode, should also output each human-readable puzzle state grid within the <O> outputs. Q = Quiet mode, only output is as described in <O>.

<initial configuration of tiles> = The row-major listing of the initial configuration of the tiles, with tile numbers separated by spaces, and "0" representing the hole.

Detailed input and output samples will be posted.

For example:

```
$ java Solver S P Q 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 0
```

would use an initial state identical to the goal state.

Deliverables:

1. Report

Submit a PDF file containing a short report that presents:

(1) Your analysis of comparing FIFO Queue to Priority Queue, including the graph you generated in the Experiments section.
(2) Your overall object-oriented design.

You should submit your report PDF along with your code on WebCAT as below, by including it in your Eclipse project or your submitted zip file.

2. Code:

You will submit your project code through the automated Web-CAT server. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will not be given a copy of grader's test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix zip command) or else a tar'ed and gzip'ed archive. Either way, your archive should contain the source code for the project only (no .class files or binary resources).

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. Programs that do not contain this pledge will not be graded.

```
// On my honor:
//
// - I have not used source code obtained from another student,
//    or any other unauthorized source, either modified or
```

```
//    unmodified.
//
// - All source code and documentation used in my program is
//    either my original work, or was derived by me from the
//    source code published in the textbook for this course.
//
// - I have not discussed coding details about this project with
//    anyone other than the instructor, ACM/UPE tutors or the TAs assigned
//    to this course. I understand that I may discuss the concepts
//    of this program with other students, and that another student
//    may help me debug my program so long as neither of us writes
//    anything during the discussion or modifies any computer file
//    during the discussion. I have violated neither the spirit nor
//    letter of this restriction.
```