**Program 3 Fall 2014**

**20 Points**

**Rotation Puzzle Revisited AVL Trees**

PART 1

You have been given the AVL tree code from your author as a starting point. Reading code is a valuable experience. On the job, you are often asked to modify/maintain existing code. You can’t start over and do it your way. You must incorporate your changes into the exisiting code. **You are expected to understand the code that has been given to you.** Make the following changes:

1. Change all variable names to be meaningful names.
2. Make any changes needed to conform to our style guidelines.
3. Write a toString function which creates an indented version of the tree (as in program 2).
4. Implement the removeMin function. Remove the smallest node in the tree and rebalance. This is to be your own work, not copied from ANY other source.

The data to be stored in the AVL tree is to be generic and work for either an integer or a GameState (of the rotation puzzle variety).

Illustrate that the AVL tree is working properly by printing the tree (using toString) after each of the following groups of operations:

* Add: 1 3 5 7 9 11 2 4 8 (now print tree)
* Remove 7 9(now print tree)
* Add 50 30 15 18 (now print tree)
* Remove Min (now print tree)
* Remove Min (now print tree)
* Remove Min (now print tree)
* Add 17(now print tree)

PART 2

**Use the AVL tree as a priority queue to improve your solution to the Rotation puzzle problem (of program 1).**

A priority queue is a queue such that insertions are made in any order, but when you remove a node from the queue, it is the one with the best priority. Our priority will be an estimation of the total amount of work needed to solve the problem – so a lower score is preferred. Thus, we will first consider boards that “look better” to us.

In our previous solution, (program 1) , we considered all one-move solutions before considering all two move solutions, and so on. If we are smarter in which solutions we consider, we can improve the functioning of the solution. In this assignment, you will compare your previous solution to this new technique.

We define a *state* of the game to contain at least:

* board
* cost so far: number of moves taken from initial state to reach current board
* expected cost to reach a solution
* String of moves indicating how the current board was achieved.

**Best-first search.** We describe a solution to the problem that illustrates a general artificial intelligence methodology known as the [A\* search algorithm](http://en.wikipedia.org/wiki/A*_search_algorithm).

First, insert the initial state into a priority queue. Then, delete from the priority queue the state with the smallest expected cost, and insert onto the priority queue all neighboring states (those that can be reached in one more move) using an appropriate estimated cost for each. Repeat this procedure until the state removed from the priority queue is the goal state. The success of this approach hinges on the choice of estimated cost or *priority function*.

**You can compute your “expected work function” any way you want as long it is reasonable and underestimates the real cost.** **Be creative.**

Here is one approach. You can use any method you want. This is just one possibility. For each number, give each entry a zero (if it is in the right location), a 1 if it is in the correct row, but the wrong column, a 1 if it is in the correct column but the wrong row, and a 2 if it is in the wrong row and wrong column.

5 8 6

4 3 1

9 7 2

*So for example, for the board above, the distances each cell is from the goal location is shown in red. Note, 5 has a distance of two because it is in the wrong row and wrong column.*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 5 | 8 | 6 |  | 2 | 1 | 1 |
| 4 | 3 | 1 |  | 0 | 2 | 2 |
| 9 | 7 | 2 |  | 1 | 1 | 2 |

If I sum these distances, I get 12. However, since one rotation can affect the distance by 3, we divide the total distance by 3 so that we have an underestimate of the actual number of moves required. (A\* search does not work if you don’t use an underestimate of the work.)

Here is another example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 6 |  | 1 | 1 | 1 |
| 4 | 5 | 9 |  | 0 | 0 | 1 |
| 7 | 8 | 1 |  | 0 | 0 | 2 |

This board has a total distance of 6. However, since one rotation can affect the distance by 3, we divide the total distance by 3 so that we have an underestimate of the actual number of moves required. Thus, we estimate this board requires at least two moves to solve. (Note, that is indeed true.)

So (for this example) if we had already taken 5 moves to get to the current state, the expected cost for the state would be 7 (5 moves to get to the state and an estimated 2 more moves to finally win). Notice we are making a crude estimate of how good the state is. We prefer a state which took few moves to reach and is close to the goal state.

We make a key observation: to solve the puzzle from a given state on the priority queue, the total number of moves required (including those already made) is at least its priority, using our distance function.

Consequently, as soon as we dequeue a board which has the goal state, we have not only discovered a sequence of moves from the initial board to the board associated with the state, but one that makes the fewest number of moves. Notice that when we enqueue the state, we do not know we won’t find another way which is better. When we dequeue, we know everything else we will enqueue in the future will take at least as many moves to reach the goal. This is true because our distance is an UNDER-estimate of the number of moves required.

**Output:**

Since we want to compare this version with our brute force solution in program 1, you will need to have both methods working. Create a class “RotationGame” which allows you to call either “bruteForceSolve” or “aStarSolve”. Each method should keep track of the total number of states that were pulled off the queue.

Print the series of states that are dequeued from the priority queue, ending with the winning state. Then re-print the original board.

Show the output for the following sample inputs:

Input 1

5 6 4

8 2 9

1 7 3

Input 2

2 1 8

9 4 7

3 6 5

Input 3: Find an example for which your “more intelligent” method saves significant time over the brute force method of program 1. Hard code this example.

Input 4: Random board (using makeboard)

### Sample Output

When using the A\* search, show each state as you pull it off the queue. Be sure to print all the data in the state.

When using the brute force search, just show us the final sequence of moves and the total number of states dequeued.

**Hints**

Print output both to the console and to a file. Then, you can easily comment out the printing to the console (to make it run faster and easier to read), while leaving the complete output in the file for the grader to look at.

**What to Turn in:**

Submit a zip file containing your software project . The zip file should contain a readme file which tells how to run the program (what IDE and compiler you used).