CS 4633/6633 Programming Assignment 1 Due: Sep 27, 2012

Part 1 The first part of this assignment is to compare several heuristics for solving the 8-puzzle using A* search. Assume the following goal state:

1	2	3
8		4
7	6	5

You don't have to implement A* (although you are welcome to). Code will be provided by Thursday that you can download from MyCourses. However, you will need to write the code for the last of the following three heuristics.

Number of tiles out of place: A simple count of the number of tiles not in their goal position. The code for this heuristic will be provided to you.

Manhattan distance: The sum of the (horizontal and vertical) distances of each tile from its goal position. The code for this heuristic will be provided to you.

Manhattan distance with linear conflicts: This is the Manhattan distance heuristic with the following enhancement: if two tiles are in the same correct row (or column) but must pass each other to get to their goal position, then add 2 to the Manhattan distance heuristic for each such occurrence. The reason you can add 2 while preserving admissibility is that the Manhattan distance heuristic does not account for the fact that one of the tiles must move out of the way to allow the other to get past.

Hand in the following for part 1:

1) A copy of the code for the Manhattan distance with linear conflicts heuristic you implemented. Indicate which version of the A* code you used to test your heuristics.
2) For at least 20 instances of the 8-puzzle, plot the *average* number of nodes expanded as a function of solution length for each heuristic.
3) At least a half-page write-up describing what you learned from these experiments.

Initial state: A tricky aspect of the sliding-tile puzzle is that not every initial state has a solution. Only half of the possible board states can be transformed into the goal state. The A* code will tell you when a particular starting state does not have a solution.

Part 2: The second part of the assignment is to implement the IDA* algorithm and use it to solve the 8 puzzle. *You must write the code for the IDA*algorithm yourself.* However, you can use code for the 8-puzzle game (the board representation, function for computing successor nodes, etc.) from the code provided for A*.

Hand in the following for part 2:

1) A documented listing of the code you wrote to implement IDA*.
2) For at least 20 instances of the 8-puzzle, plot the average number of nodes expanded *during the last iteration of IDA** as a function of solution length, using either the Manhattan distance heuristic or the linear conflicts heuristic.
3) At least a half-page write-up describing what you learned from implementing IDA*.

Extra credit:

You will get up to 20 points extra credit if your implementation of IDA* also solves the 15-puzzle. The 15-puzzle is much more difficult to solve than the 8-puzzle, but IDA* can solve it easily if implemented efficiently. (By contrast, A* runs out of memory on most examples of the 15-puzzle, especially those that require long solution paths.)

If you run your implementation of IDA* on the 15-puzzle, you should consider including the following enhancements in your code to make it more efficient.

Parent checking: You can make your implementation of IDA* more efficient by not generating a successor node that is the same as the parent of a state. Although IDA* will work correctly without this enhancement, adding this check will eliminate some duplicates from your search tree.

Node ordering: Your implementation of IDA* will run much faster if you expand the children of each node in increasing order of the heuristic estimate. This technique is called node ordering. Because IDA* stops as soon as it finds a solution, this enhancement allows it to visit many fewer nodes during the last iteration. It will not make earlier iterations any faster, but most of the running time of IDA* is spent in the last iteration.