CS540 – Assignment 2

CS540 – Fall 2015 – Sections 2 & 4
University of Wisconsin, Madison

Submission and Deadlines:

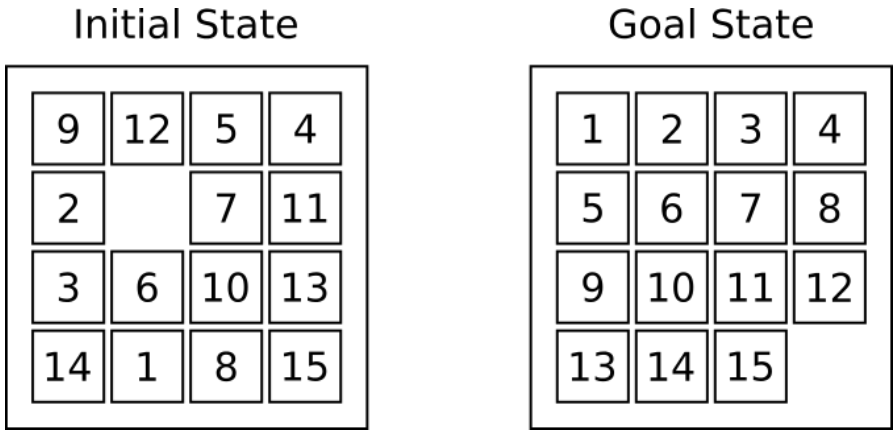
You will submit two separate files through Moodle for this assignment: (1) a .pdf file with your written answers to the theory questions, and (2) a .java file with your code implementation for the practical problems. Both of these files shall be submitted no later than **Noon on Tuesday, October 20th** to receive full credit. Late submissions will be penalized 10% per 24 hours that they are received late. After five days (120 hours), late submissions will no longer be accepted for credit.

Collaboration Policy:

This entire assignment shall be completed individually. While you are encouraged to discuss relevant concepts and algorithms with classmates and TAs, you shall only use different examples and problems from those assigned below. Academic misconduct on this assignment includes, but is not limited to: sharing, copying, and failing to protect the privacy of your answers and code fragments with respect to the assignment described below.

Heuristic Search (Theory)

0. For this portion of the assignment, consider the following sliding tile puzzle:



This fifteen-puzzle is a larger version of the eight-puzzle described in Chapter 4. For each of the following problems, assume that the available sliding actions are searched in the following order: slide-left (moving the 7 in the initial state), slide-up, slide-right, and then slide-down.

1. List the first ten actions explored for each of the following search strategies. List each action by the number of the tile that is being moved to explore a new state. As an example, a slide-up action followed by a slide-right action might be listed as either 6, 3 (if the first action leads to the state that the second action is explored from), or 6, 2 (if both actions are taken from the initial state). For the first three search strategies (1.1-1.3) assume that the all previously explored states are being tracked, and that all actions leading to any previously explored states are skipped. For the final search strategy (1.4) include all explored actions, even those that return to a previously explored state.

1.1. Breadth-First Search
1.2. Uniform-Cost Search: Where the cost of moving each tile is the number on that tile. And ties are broken using the action priorities listed above with respect to the earliest different action between two paths with equal costs.
1.3. Depth-First Search
1.4. Iterative Deepening Depth-First Search

2. Assume that the cost of sliding each tile is one. Based on this assumption, evaluate whether each of the following heuristics are admissible or not admissible.

   A(state) = the number of tiles in the wrong column + the number of tiles in the wrong row
   B(state) = the number on the largest tile that is out of place
   C(state) = the total number of tiles that are in their correct position
   D(state) = the sum of the Manhattan distances between each tile's position and goal position
   E(state) = the number of tiles that are in the wrong column
   F(state) = the total number of tiles that are not in their correct position
   G(state) = the sum of the straight-line distances between each tile's position and goal position

3. Independent of whether they are admissible, which of the following statements about the heuristics listed in the previous problem are true?

   3.1. B dominates A
   3.2. D dominates G
   3.3. A dominates E
   3.4. F dominates A

4. Calculate the heuristic value for the initial state depicted above according to each of the following heuristics:

   4.1. A(InitialState) = ?
   4.2. B(InitialState) = ?
   4.3. D(InitialState) = ?

5. Now perform the first steps in the A* Search algorithm using as the heuristic function: the total number of tiles that are not in their correct position. The actual cost of sliding each tile is one. List the first seven actions explored in this search. For each of these actions, include the number of the tile being slid along with the following values for the resulting state: the heuristic estimate h(), the incurred cost g(), and the resulting search priority f(). Ties are broken using the action priorities listed above with respect to the earliest different action between two paths with equal costs.

   Note regarding "the first seven actions ***explored*** in this search", this refers specifically to actions leading to states that are added to the explored queue (as discussed in class), and not actions leading to states that have only been added to the frontier queue.

Decision Trees (Practical)

0. For this portion of the assignment, you will be implementing a class to learn and test a decision tree. You are being provided with code for a processing driver application (Main.java), and a

DrawableTree class (DrawableTree.java) that your decision tree class will inherit from to help you visualize the resulting tree. Your implementation will be exclusively written into the provided class skeleton (FIRSTNAME_LASTNAME_DecisionTree.java). After replacing the FIRSTNAME_LASTNAME portion of this filename with your own name, you will need to update the classname within this file, and the single reference used to instantiate this class in Main.java. Your final solution should run without any other changes being made to Main.java or DrawableTree.java. DO NOT include any package statements, and DO NOT call exit from any of the methods that you are implementing.

Processing
Since the latest stable release of processing was relased a few days prior to this assignment, we will continue to use version 2.2.1 of the processing library. This is the same version that was used for Assignment01, and the core.jar file for this version is available on moodle.

Data
For this assignment, you will need some data to learn from, and then to test your decision tree against. You are being provided with two main xml data files for this purpose: trainData.xml and testData.xml. Each of these files contain the voting records for select members of the 1984 US house of representatives over 16 key pieces of legislation, along with each representative's political party affiliation (based on this UC-Irvine ML Dataset). In the provided xml files, you will find that each example contains a party attribute, along with 16 vote## attributes. The party is either DEMOCRAT or REPUBLICAN, and this is the attribute that your decision tree will learn to predict. Each vote## attribute has a value of YEA or NAY, and all examples include the same number of vote attributes. Although it will be nice for your program gracefully report any anomalies it might find in the format of this data, all of the grading tests will utilize data sets that conform to this format. You are highly encouraged to create some additional smaller data sets to test, and validate your programs correctness.

1. There are nine methods for you to implement in the decision tree class: seven that contribute to learning the decision tree (see step 2), and two that contribute to testing the tree that you have learned against new examples (see step 4). You are welcome to add helper methods and fields to this class, as long as calling each of the assigned methods is sufficient for them to complete their task.

2. The learnFromTrainingData() method is responsible for loading an xml file of training examples, and then creating a decision tree based on those examples. This resulting decision tree must ultimately be stored in the XML field *tree* that is part of the DrawableTree super class. Another helpful field in this class is the PApplet field *p*. This can be used to access processing functionality like the loadXML(...) method: via a call to p.loadXML(). Don't forget to remove empty whitespace nodes from any xml data that is loaded in this way. Finally, the *dirtyTree* field in the DrawableTree class should be set to **true** before learnFromTrainingData() returns. This flag will notify the DrawableTree class that it should re-draw the new tree that you have just built and stored in the field *tree*. The large task of building a decision tree has been decomposed into six methods that are described in the next step. You must fully implement each of these methods for full credit on this assignment.

3. As discussed in class, you will use a recursive tree building algorithm to build your decision tree. Each call to the recursiveBuildTree() method will utilize a unique set of examples (children of *dataset*) to decide whether the current tree node (*tree*) should be labeled with a

party classification, or an attribute to further branch on and then recursively fill the children of. Party classifications can be decided using the plurality() method. This method should return a string indicating the mode or most common party among the examples passed to it (or "TIE" in rare case that you have an exact tie). The chooseSplitAttribute() method calculates and returns the name of the attribute that will best divide the provided dataset into the lowest entropy subgroups. The entropy that results from splitting any dataset by a given attribute will be calculate by the calculatePostSplitEntropy() method, which in turn may call the method calculateEntropy() which calculates the entropy of any single group of examples. Remember that this entropy is with respect to the party affiliation that we are trying to learn. The final method that you will implement and utilize in building your tree is the entropy formula for Boolean random variables, based on the probability of being in either of the two possible categories. The formula for this calculation is on page 704 in your textbook, but don't forget to use the limit when the input probability makes this formula unstable.

4. The runTests() method loads a file full of examples to test any decision tree that you have built. There is only one helper method assigned for you use to this end: predict(). The predict() method steps through a decision tree based on the attributes of a single example, and returns the party affiliation reached in a leaf of this tree. This predict() method should ignore any party attributes that are passed to it. After comparing each of the testing examples' actual parties to your decision tree's predicted parties, runTests() should return the percentage of examples that your tree was able to correctly predict.

5. Congratulations. Now that you have implemented each of the assigned methods to learn and test a decision tree, you are now left with the task of testing and validating the code that you have written. You are encouraged to develop some smaller sample training and testing data sets that you can work through by hand, and then compare your results to those created by your code. The provided tree visualization code may help you in this endeavor. There is no specific credit or deliverable tied to this testing, but ensuring that your code is correct should help your code avoid failing any of the tests that we design for grading the nine methods that you have been assigned to implement.