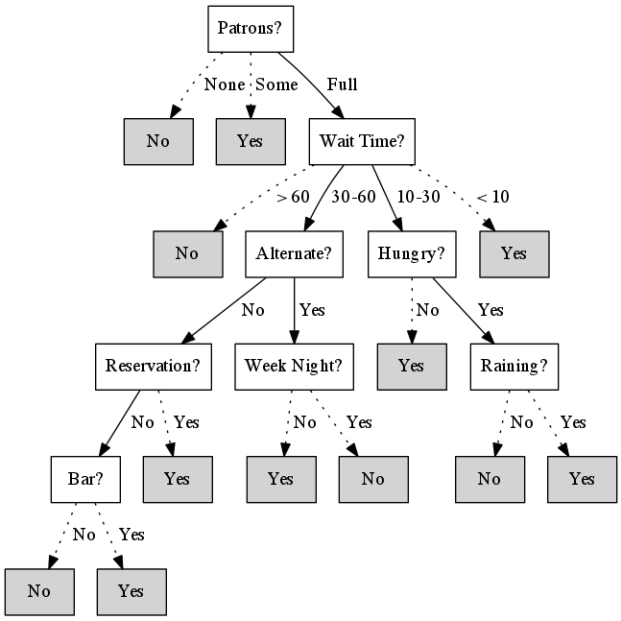
[P09] Making Decisions

Due: 12:01am, Monday November 18th.

Description

Expert systems were one of the first approaches to artificial intelligence. AI researchers would define models for decision making which could be constructed by domain experts, thereby capturing the knowledge of the expert in a program.

In a decision tree, each internal node describes a question to ask given the data, and each leaf node describes the decision made. A classic decision tree example describes determines whether a couple should wait for a table at their current restaurant or go elsewhere.



If the restaurant is not full, then the wait will not be long and the couple should wait for their table. When the restaurant is instead full, has a moderate wait, and the couple is hungry the decision to wait is dependent on the weather. If

the weather is poor then it is more important to wait then to adventure out to another establishment. There are certain social values encoded in to the tree. The stigma of an empty restaurant is sufficient for our couple to venture elsewhere regardless of the other circumstances. The structure of the tree in general depends on the data and expertise available to the person or algorithm constructing it.

While their popularity has waxed and wained, decision trees have been a staple in the subfield of expert systems.

In this assignment you will develop a program which reads a decision tree from a file in a standard format and then responds to queries on the structure. The program will support enumerations as datatypes and will respond with boolean responses.

Requirements

Your program must be in the file `DecisionTree.java`.

You are not allowed to use any imports with the exception of the `Scanner` class and the `java.io` packages and subpackages.

The decision tree specification for the restaurant example above follows. While indented for readability, whitespace is insignificant and should not be relied on.

```
? 0 NONE SOME FULL
  ! false
  ! true
  ? 1 EXCESS LONG SHORT NONE
    ! false
    ? 2 NO YES
      ? 3 NO YES
        ? 4 NO YES
          ! false
          ! true
        ! true
      ? 5 NO YES
        ! true
        ! false
    ? 6 YES
      ? 7 NO YES
        ! false
        ! true
      ! true
    ! true
```

The `?` operator, followed by an index, specifies that a decision should be made on

the entry at the specified index. A space delineated list of possible enumeration values, in the order they will be considered, are listed following the operator and index.

In this example index 0 (patrons) is associated with the enumeration containing `NONE`, `SOME`, and `FULL` in that order. The name of the enumeration should be compared to the enumeration list in the specification to find which child node to use. Index 1 (wait time) follows similarly. The remaining indicies are associated with a binary yes/no decision process, but again be wary of the order (see index 6) in which the order of the enumerated types and the order of child nodes has been flipped.

The `!` operator specifies a leaf node containing a boolean value.

The `DecisionTree` class must contain two methods. The first is a static method `construct` which takes a filename and returns a `DecisionTree` instance:

```
/**
 * Constructs a decision tree for its specification.
 *
 * @param filename a decision tree specification filename
 * @return a decision tree
 */
public static DecisionTree construct(String filename)
```

The second is an instance method `evaluate` which takes an array of `Enum` instances and returns a boolean decision.

```
/**
 * Evaluates the decision to be made given the datapoint given.
 *
 * @param datapoint a set of enumerated values
 * @return the boolean decision given by the decision tree
 */
public boolean evaluate(Enum<?>[] datapoint)
```

Evaluating a specific datapoint for the example problem:

```
enum Patrons  { NONE, SOME, FULL };
enum WaitTime { EXCESS, LONG, SHORT, NONE };
enum Affirm   { NO, YES };

DecisionTree decision = DecisionTree.construct("restaurant.dt");
decision.evaluate(new Enum<?>[] {
    Patrons.FULL,  // patrons
    WaitTime.LONG, // wait time
    Affirm.YES,    // alternate
    Affirm.NO,     // reservation
    Affirm.YES,    // bar
    Affirm.YES,    // week night
    Affirm.NO,     // hungry
    Affirm.YES     // raining
});
```

the evaluate method returns true.

Grading

The point breakdown is as follows.

20 The construct method produces a decision tree

40 The evaluate method works properly on decision trees which exclusively use the enumeration:

```
enum Affirm { NO, YES }
```

The two instances `NO` and `YES` may still be out of order in the specification.

40 The evaluate method works properly on decision trees which use arbitrary size enumerations such as:

```
enum Patrons { NONE, SOME, FULL };
```

or

```
enum WaitTime { EXCESS, LONG, SHORT, NONE };
```

Your code should conform to the courses coding style especially those concerning variable names, whitespace, and class, (main) method, and inline comments. You can lose up to 30 points for bad style or commenting.

Reminder — a program which fails to compile on the lab machines (under Java 1.6) will receive 0 points.

Submission

1. Create a zip file of your program using the utility `zip`.

```
> zip -r Program09.zip ...
```

2. Run the submit program.

```
> submit
```

   A text-based interface will appear in the terminal.

3. Use the arrow keys and enter to select a class (cs1131), section (sec1), and assignment (Program09).

4. Use the arrow keys to navigate to the file you would like to submit. Use the spacebar to select the file Program09.zip (an astrisk will appear next to the filename). If the filename is incorrect or you attempt to submit a different file, submit will not accept it.

5. Hit return to submit the selected file(s) and confirm the submission.

6. Verify that the file appears in the "Already Submitted" window pane.

You can use submit at any time to verify that the files have been turned in. The date/time of each submission is also listed.