# Problem #1

**Consider the partially explored Wumpus World:**



**Figure 1 – Original Wumpus World Board**

**Here “X” indicates explored squares with nothing detected. “B” indicates a breeze was felt on that square while an “S” indicates a stench from the wumpus. For each square on the frontier, determine the probability the square is safe. Which square should a rational agent who must find the gold search next?**

Given this Wumpus World, there are two possible frontiers. They are shown in and .



**Figure 2 – Possible Wumpus World Board #1**



**Figure 3 – Possible Wumpus World Board #2**

When cell 1,4 is excluded, there is only a single logical board. Hence that board has probability 1. This means the probability of a pit in square 1,4 is defined as below:

since is equal to 1 as the sum of the probabilities is 1. Note for squares 1,4 and 2,3, the probability of wumpus 0 since neither had a stench detected.

Given the known information, square 2,3 has to have a pit. This can be shown using the probabilistic method as well:

is equal to making the resulting probability vector:

The probability that square 3,2 is safe is defined as:

The presence of a pit and the presence of a wumpus are independent. Hence, this equation simplifies to:

The probability of a pit in square 3,2 is defined as:

The normalization constant becomes making the probability vector:

The probability of a Wumpus in cell 3,2 is defined as:

Note the probability of any unexplored square having a Wumpus is 0.1 since there is only one wumpus and 10 unexplored squares. The normalization constant is making the probability vector:

Hence the probability square 3,2 is safe is:

The last frontier square is cell dependent solely on the presence of a wumpus since it does not have a breeze. Hence, its probability is:

The normalization constant is 10 making the probability vector:

Hence, the agent should explore square 3,2 should be explored next as it is guaranteed to be safe (i.e. . In contrast, squares 2,3 and 4,1 are guaranteed to lead to the agent’s death . Square 1,4 is reasonable safe (i.e. ), but it is still less safe than square 3,2.

Problem #2

**Consider the following table of data concerning whether or not a person was a movie star:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Example ID** | **Plastic Surgery** | **Teeth Color** | **Manicure** | **Pedicure** | **Is Movie Star?** |
| *e1* | FaceLift | Yellow | No | No | No |
| *e2* | None | White | Yes | Yes | Yes |
| *e3* | NoseJob | White | No | Yes | Yes |

Table – Decision Tree Algorithm Learning Data Set

**Work out the decision tree that our decision tree learning algorithm would compute from the above training set.**

The decision tree algorithm is a greedy divide-and-conquer algorithm that always tests the most important attribute first. It then divides the problem into smaller subproblems that each can be solved recursively. The “most important attribute” is the one that makes the most difference in the classification of the example.

**def** **DECISION\_TREE\_LEARNING**(*examples*, *attributes*, *parent\_examples*):

# examples – Set of all cases from the learning set.

# attributes – Unclassified features in our decision problem.

# parent\_examples – Examples in the tree’s parent node

**if** len(*examples*) == 0:

**return** **PLURALITY\_VALUES**(*parent\_examples*)

**elif** all *examples* have the same *classification*:

**return** *classification*

**elif** len(attributes) == 0:

**return** **PLURALITY\_VALUES**(*examples*)

# Select the most important argument

*A* = **IMPORTANCE**(*a*, *examples*)

# Iterate through the domain values of that argument and build a subtree for it

**for each** *v\_k* **in** *A*:

# Select the subset of examples that have the value v\_k

*examples\_subset* = { *e* : *e* *examples* and *e.A* = *v\_k* }

*subtree* = **DECISION\_TREE\_LEARNING**(*examples\_subset*, *attributes* - *A*, *examples*)

add a branch to tree with label (*A* = *v\_k*) and subtree *subtree*

# Return the tree/subtree

**return** tree

Figure – Decision Tree Algorithm Pseudocode

The two important subfunctions in this algorithm are:

**PLURALITY\_VALUES** – Returns the most common output value from a set of examples. If two or more output values are maximally likely, the algorithm randomly breaks the tie.

**IMPORTANCE** – Greedily selects the attribute that is most likely to minimize the decision tree. A “good” attribute is one that divides the *examples* into sets each of which is either all positive examples or all negative examples, which would result in leaves on the decision tree.

The flow of this algorithm are:

**Step #1: Initial Function Call**

In , I added example identification numbers to each training data set. This was to simplify the notation in the step by step explanation. Using this notation, the initial function call would be:

**Step #2: Check the Termination Conditions**

From the pseudocode in , you can see that none of three initial cases are satisfied:

*examples* – Not empty as it has all three initial examples: .

Identical classification for *examples* – Two examples are positive while one example is negative.

*attributes* – Not empty. It has

Since the termination conditions are not met, run the IMPORTANCE function.

**Step #3: Run the Importance Function**

The quality of an argument in a Boolean decision tree is defined by the **information gain** equation:

The entropy of a Boolean random variable, , is defined as:

The information gain for the four variables in this tree are:

**Note:** I use approximately in some of the previous equations since is undefined even when multiplied by 0.

Hence, variables , , and all have equivalent information gain. The algorithm can choose between them randomly. In this case, I will choose . This results in two new recursive calls. They are:

For :

For :

**Step #4: Check Termination Conditions of Recursive Call for** :

There is only one example () in ­*examples* for this function call. Hence the function returns the classification “No”. Immediately after this step, the edge would be added, but for simplicity of explanation, I delayed it until step #6.

**Step #5: Check Termination Conditions of Recursive Call for** :

Both examples ( and ) have classification “Yes” so “Yes” is returned. Immediately after this step, the edge would be added, but for simplicity of explanation, I delayed it until step #6.

**Step #6: Connect Subtrees**

Subtrees are appended. Both subtrees are leaf nodes as shown in .



Figure – Decision Tree for Problem #2