Write-Up for Project 2

To begin with, I mostly emulated pseudo-codes in the textbook for this project as it already provides solid grounds for both part I and part II. For the part I, I have implemented truth-table enumeration method using recursion, whose detail was aptly delineated by Peter Norvig in the textbook. As for part II, I have decided to implement WalkSAT algorithm because it seemed like I could reuse pl_true() function in the implementation of WalkSAT.

That said, as much as my codes resemble pseudo-codes in the textbook, I believe what makes my program idiosyncratic is its implementations of 'propositional_symbols()' and 'pl_true()' functions.

```
def propositional_symbols(kb_and_alpha):
    '''breaking up kb and alpha into symbols'''
    split_symbols = re.split(r"\s|-|,|v|w+\d|[()]|;", kb_and_alpha)  # parsing symbols
    only_symbols = {sym for sym in split_symbols if sym.isalnum()}
    return sorted(only_symbols)
```

Figure 1. propositional_symbols() function

As shown in figure 1 and figure 2, I have chosen to parse propositional logic sentences / clauses using regular expression. The reason for the decision is threefold: I am familiar with using regular expressions as I have used it previously on other projects for other courses. However, more importantly, using regression expressions can make the program easily adopt to new propositional symbols that might get added in the future. For instance, should the need arise in the future to add a new symbol, say $, I can just add |$ to the end of the argument of re.split(). Additionally, I think it lead to much shorter code in general since I do not need to create a new class for propositional symbols.

```
def pl_true(kb, model):
    '''testing whether propositional logic is true in the model'''
    rules = kb.strip(';')  # model is dict
    rules = re.split(r", ", rules)  # string
    # find value of each key that matches with rule and replace them
    rule_number = 1
    rule_dictionary = {}
    for rule in rules:
        # looping over each rule
        rule = re.sub('[\^]', 'and', rule)
        rule = re.sub('(<=>)', 'iff', rule)
        rule = re.sub('(=>)', 'then', rule)
        split_symbols = re.split(r"\s+|(\W+)", rule)
        erase_empty = [sym for sym in split_symbols if sym]
        # print(erase_empty)
        new_list = []
        for each in erase_empty:
            # looping over each character in a rule
            if model.get(each) is True:
                new_list.append(str(True))
            elif model.get(each) is False:
                new_list.append(str(False))
            elif model.get(each) is None:
                if each == '-':
                    new_list.append('not')
                elif each == 'v':
                    new_list.append('|')
                elif each == 'and':
                    new_list.append('&')
                elif each == 'then':
                    new_list.append('then')
                elif each == 'iff':
```

Figure 2. pl_true() function

In that vein, my ‘pl_true()’ function first splits clauses into a list of individual clauses. And then, it instantiates a dictionary who would store true/false value of each rule (or clause) at the end of evaluation. Next, it finds propositional symbols and substitutes the symbol with a human-readable language that would again later be converted into symbols that compliers can understand.

Of all the propositional symbols, I found implementing ‘implication’ the most challenging because I had to convert p -> q into –p v q. The workaround I used was to split a clause whenever there is an occurrence of ‘->’ and pad parentheses to every element in the list created by splitting the clause, except the last element. Finally, I would add negation symbols in front of every element but the last one and join the list again using ‘or’. At first, I was a fairly hesitant to leverage this workaround because it seemed like a greedy algorithm that would later prove to have a fetal error. However, so far, it seems to work well for solving given problems. Therefore, I would say more test is needed.

As for WalkSAT algorithm, I think it was quite straightforward since much of its most difficult parts were already solved in part I. For part II, I have included print statements for each step so that the reviewer can see how the programming is solving problems.

```
if p > probability:
    symbol_to_be_flipped = random.choice(
        propositional_symbols(random_false_clause))
    print('\nThe value of ' + str(symbol_to_be_flipped) + ' will be flipped.')
else:
    num_of_satisfied_clauses = {}
    for flipped in propositional_symbols(random_false_clause):
        counter = 0
        # for each symbol in a randomly picked clause that was false
        # flip it and change the value of the symbol that gives
        # the maximum number of satisfied clauses
        model[flipped] = not model[flipped]
        # loop over each clause again with the changed value
        for cl in clause:
            if pl_true(cl, model):
                # add to counter
                counter += 1
        # resetting the value after simulation
        model[flipped] = not model[flipped]
        num_of_satisfied_clauses[flipped] = counter
    symbol_to_be_flipped = max(
        num_of_satisfied_clauses, key=num_of_satisfied_clauses.get)
    print('\nFlipping ' + str(symbol_to_be_flipped) +
    ' gives us ' + str(num_of_satisfied_clauses[symbol_to_be_flipped])
    + ' satisifed clauses')

# change the value of a symbol in the model
model[symbol_to_be_flipped] = not model[symbol_to_be_flipped]
```

Figure 3. Random assignment part of WalkSAT algorithm

Figure 3 shows where I have spent most time in implementing WalkSAT. The most interesting part of WalkSAT algorithm is how it chooses which symbol to flip. For probability I used 0.5 as recommended by the textbook. I chose max_flip to be 50 because in testing the algorithm, I have found that it usually takes about at most 10 iterations before the program terminates. I wanted the program to have a somewhat loose upper bound in case of a complex propositional sentences so I chose 50.

That said, first if statement is for situations when probability is over 0.5, in which case the program randomly picks a symbol from a randomly chosen false clause. Then, it flips the value of the symbol at the end of if statement (at the end of else).

Now, else statement is for cases when the probability is less than 0.5. Should the probability chosen to be less than 0.5, the program first instantiates a new dictionary which stores the number of clauses that would be satisfied under a new condition in which a symbol of a false clause is flipped. After 'simulating' each clause under a new condition, the program then chooses the symbol that gives the maximum number of satisfied clauses. Finally, it flips the value of the said symbol.

I have included instructions as to how to run the program in the README.txt file. Overall, it was a great project which furthered my understanding of propositional logic and how the a computer program might go about processing natural language.

The followings are the screenshots of how my program performs given problem #1 – 3.

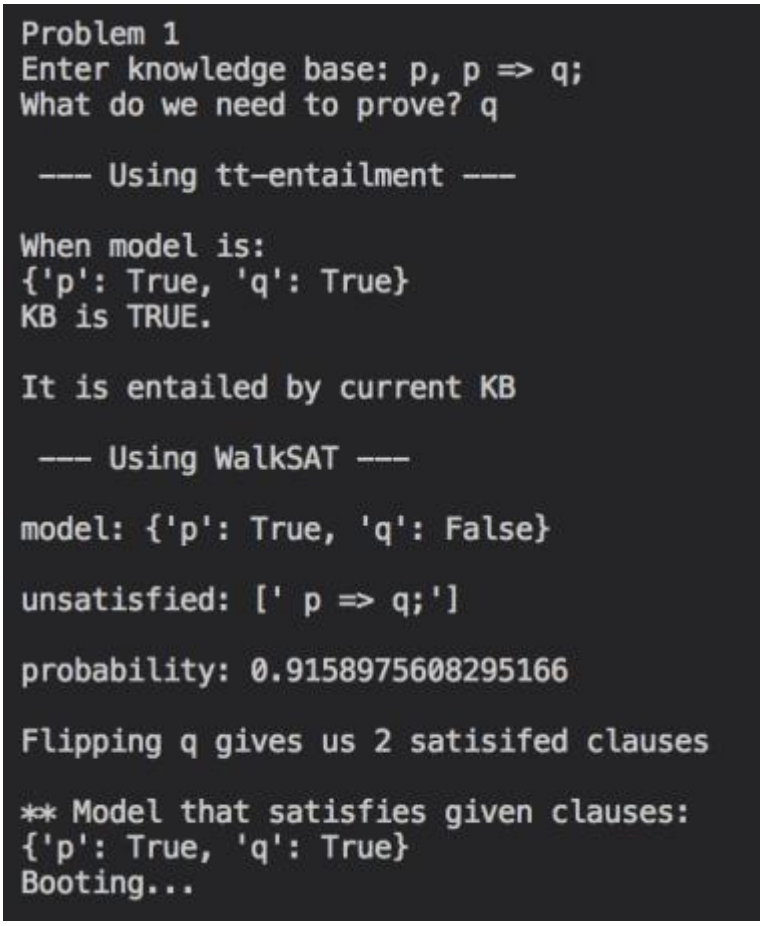

Figure 4. Problem #1

```
Problem 2
Enter knowledge base: -p11, b11 <=> (p12 v p21), b21 <=> (p11 v p22 v p31), -b11, b21;
What do we need to prove? p12

 --- Using tt-entailment ---

When model is:
{'b11': False, 'b21': True, 'p11': False, 'p12': False, 'p21': False, 'p22': True, 'p31': True}
KB is TRUE.


When model is:
{'b11': False, 'b21': True, 'p11': False, 'p12': False, 'p21': False, 'p22': True, 'p31': False}
KB is TRUE.


When model is:
{'b11': False, 'b21': True, 'p11': False, 'p12': False, 'p21': False, 'p22': False, 'p31': True}
KB is TRUE.


 --- Using WalkSAT ---

model: {'b11': False, 'b21': True, 'p11': False, 'p12': True, 'p21': False, 'p22': True, 'p31': False}

unsatisfied: [' b11 <=> (p12 v p21)']

probability: 0.3550311227402577

The value of p12 will be flipped.

** Model that satisfies given clauses:
{'b11': False, 'b21': True, 'p11': False, 'p12': False, 'p21': False, 'p22': True, 'p31': False}
```

Figure 5. Problem # 2

```
Problem 3
Enter knowledge base: myth => -mort, -myth => (mort ^ mamm), (-mort v mamm) => horn, horn => magic;
What do we need to prove? myth

 --- Using tt-entailment ---

When model is:
{'horn': True, 'magic': True, 'mamm': True, 'mort': True, 'myth': False}
KB is TRUE.


When model is:
{'horn': True, 'magic': True, 'mamm': True, 'mort': False, 'myth': True}
KB is TRUE.

It is entailed by current KB

 --- Using WalkSAT ---

model: {'horn': False, 'magic': False, 'mamm': True, 'mort': False, 'myth': True}

unsatisfied: [' (-mort v mamm) => horn']

probability: 0.8476704454458824

Flipping mamm gives us 4 satisifed clauses

** Model that satisfies given clauses:
{'horn': False, 'magic': False, 'mamm': False, 'mort': False, 'myth': True}
```

Figure 6. Problem #3 part (a)

```
Problem 2
Enter knowledge base: myth => -mort, -myth => (mort ^ mamm), (-mort v mamm) => horn, horn => magic;
What do we need to prove? magic

 --- Using tt-entailment ---

When model is:
{'horn': True, 'magic': True, 'mamm': True, 'mort': True, 'myth': False}
KB is TRUE.

It is entailed by current KB

 --- Using WalkSAT ---

model: {'horn': True, 'magic': False, 'mamm': False, 'mort': False, 'myth': True}

unsatisfied: [' horn => magic;']

probability: 0.06358622040273298

The value of horn will be flipped.

** Model that satisfies given clauses:
{'horn': False, 'magic': False, 'mamm': False, 'mort': False, 'myth': True}
```

Figure 7. Problem #3 part(b)

```
Problem 3
Enter knowledge base: myth => -mort, -myth => (mort ^ mamm), (-mort v mamm) => horn, horn => magic;
What do we need to prove? horn

 --- Using tt-entailment ---

When model is:
{'horn': True, 'magic': True, 'mamm': True, 'mort': True, 'myth': False}
KB is TRUE.

It is entailed by current KB

 --- Using WalkSAT ---

model: {'horn': True, 'magic': False, 'mamm': False, 'mort': False, 'myth': False}

unsatisfied: [' -myth => (mort ^ mamm)', ' horn => magic;']

probability: 0.7195031726885011

Flipping myth gives us 3 satisifed clauses
unsatisfied: [' horn => magic;']

probability: 0.3308295118674722

The value of horn will be flipped.

** Model that satisfies given clauses:
{'horn': False, 'magic': False, 'mamm': False, 'mort': False, 'myth': True}
Booting...
```

Figure 8. Problem #3 part (c)