Project 2 Report: Runhua Gao

API Design:

GetModel(literals): pass in a list of literals, return all possible worlds for these literals

ModelChecking(literals, KB,proved): pass in a list of literals, knowledge base(set of Sentences class), proved(The sentence we wanna see if it is entailed by knowledge base, optional), return all models that satisfy the knowledge base and print if knowledge base entails the proved sentence if proved is offered

Resolution(clause1,clause2): pass in two clause in CNF format(preprocess the sentence, convert it into CNF, hard-coded), return the result of resolved clause

Detail about Resolution:

If there are no complementary elements in these two clause, return None

If after the resolve there are no elements in union clauses, return empty(do not equal None)

If there are complementary elements in union clause after resolution, return None(A v !A must be true, discard it)

PL_Resolution(resolutionKB + proved,length): pass in the resolution knowledge base and the clause we want to proved(please negate it before pass in as required in PL_Resolution pass in KB^!a if we want to prove a)

PL_Resolution.Proved(): This method is a standard **PL_Resolution algorithm** implementation. For example, we want to see if a knowledge base could prove magical in Problem 3, then we could add **negative magical** to see if it resolves a empty in the resolution process.(return true or false)

There is a critical optimization I do here. As for big knowledge base, for example, problem 5 and problem 6, there so much clause sentences in the resolution knowledge base. In this case, each PL resolution loop could produce many resolved clauses but useless. For example, it could produce a six-length clause like(!amy, bob, cal, hal, ida, !lee) etc. But these sentences do no help to our resolution process as it could never resolve a unit clause(like amy), which is real we want the resolution algorithm to produce. So here I add a little code to the PL_Resolution algorithm:

```
    self.printResolve(s1, s2, resolvents)
if len(resolvents) <= 2:  # critical optimization
    self.new.append(resolvents)
```

This if statement is to say that the new clause set only accept those clause sentences whose length(number of symbols) is less or equal to 2. Because only these short clause sentences have hope to produce what we want!

In my opinion, **this method makes sense for all problems in this project**, as what we wanna prove is a unit clause, and the longest clause in knowledge base is less than 4(in Problem 5 it is 3). As a result, there is no need to care about those long clause because they will never resolve the unit clause we want.

Another reason I do this optimization is about the **time and space complexity**. For PL resolution algorithm, it will consume a lot time and space to check equality and store new clauses. In this case, may be for a big computer it is easy to compute whole clause without limitation of length. But for a pc, it is hard to do so.

Test Function: resolutionTest(kb,literal): pass in a resolution knowledge base and a literal(a string symbol) to see if the knowledge base could entail the literal or not ,using **PL_Resolution algorithm.** I use this test function to test all my PL resolution's correctness in all problems. The theory is to add literal and negative literal to knowledge base separately and compare the result.

Sentence(operator,*args, sentenceBase = False): pass in a operator and arguments to build a sentence,
Operators available[conjunction, disjunction, implies, equals, unit, negative]
When choose operator as unit and negative, must pass in only one argument
When choose operator as disjunction and conjunction, no limit
When choose operator as implies and equals, must pass in two argument
If a sentence A relies on another sentence B, pass B as argument of A and A.sentenceBase = True

CnfSentence(*args): pass in a list of arguments(both negative or positive),stands for a clause, that will be used in PL_Resolution

Detail:

Override __eq__ meta method of Cnfsentence, clarify if two Cnfsentences are equal relies on their args

Override __repr__ meta method of Cnfsentence, return set of its arguments' string

Arguments(symbol, isNegative): pass in a string symbol that represents an argument, if isNegative is True, the argument represents a negative literal

Details:

Override __eq__ meta method for Arguments, return if two Arguments are equals relies on their symbol string and isNegative

Override __repr__ meta method for Arguments, represent its symbol if isNegative is False, or "!"+symbol if it is Negative

Operators: and: and, or: or, not: ~, implies: >, equals: =

Files Description:

Argument.py: Stores Argument class

ModelChecking.py: Stores Modelcheck and GetModel class

PL_Resolution.py: Stores PL_Resolution and Resolution class

Parameters.py: stored all sentence operators and operator evaluation function

Sentence.py: stored Sentence and Cnfsentence class

Problem Solving Process Description:

Problem 1: Problem_1_Modus_Ponen.py

For modelchecking:

First establish two sentences, P, P implies Q. The sentence we want to prove is Q

Pass P, P implies Q as KB, Q as proved to ModelChecking class to see if knowledge base entails Q

Run **modelchecking()** to see the modelchecking result

For Resolution:

Knowledge base is established in **resolutionKB()**

Test Code:

resolutionTest(resolutionKB(), "Q")

Model checking and resolution test screen shot:

```
Now do model checking
P is True, Q is True,
P is True
 P > Q  is True
Q is True


KB entails Q
Now do PL_Resolution
test object is Q
Now add Q to knowledge base
result is:  False
Now add !Q to knowledge base
result is:  True
Q is True
--------------
```

Please directly run the Problem file to see result of both model checking and resolution test.

Problem 2: Problem_2_WumpusWorld.py

For model checking:

First establish model checking knowledge base(establsih in model checking), to see the result, directly **run modelchecking()**

For resolution:

Knowledge base is established in **resolutionKB().**

Test code:

resolutionTest(resolutionKB(), "P_1_2")

Answer: P_1_2 is False

Model checking and resolution test Screen shot:

```
B_1_1 is False, B_2_1 is True, P_1_1 is False, P_1_2 is False, P_2_1 is False, P_2_2 is True, P_3_1 is True,
  P_1_2 or P_2_1 =B_1_1  is True
 B_2_1= P_1_2 or P_2_2 or P_3_1   is True
~P_1_1 is True
~B_1_1 is True
B_2_1 is True
~P_1_2 is True


B_1_1 is False, B_2_1 is True, P_1_1 is False, P_1_2 is False, P_2_1 is False, P_2_2 is False, P_3_1 is True,
  P_1_2 or P_2_1 =B_1_1  is True
 B_2_1= P_1_2 or P_2_2 or P_3_1   is True
~P_1_1 is True
~B_1_1 is True
B_2_1 is True
~P_1_2 is True


B_1_1 is False, B_2_1 is True, P_1_1 is False, P_1_2 is False, P_2_1 is False, P_2_2 is True, P_3_1 is False,
  P_1_2 or P_2_1 =B_1_1  is True
 B_2_1= P_1_2 or P_2_2 or P_3_1   is True
~P_1_1 is True
~B_1_1 is True
B_2_1 is True
~P_1_2 is True


KB entails ~P_1_2
Now do PL_Resolution
test object is P_1_2
Now add P_1_2 to knowledge base
result is:  True
Now add !P_1_2 to knowledge base
result is:  False
P_1_2 is False
--------------
```

Please directly run the Problem file to see result of both model checking and resolution test.

Problem 3: Problem_3_HornClauses.py

For modelchecking:

The knowledge base is establish in **modelchecking()**, to see the result dirctly run it

For resolution:

Knowledge base in established in **resolutionKB()**, as we get some answer in modelchekcing,(the knowledge base could entail horned and magical), to see whole whole resolution process, **directly run Problem file**, you could see that the knowledge could resolve horn and magical as from model checking that the KB only entails magical, horned Resolution test code:

testResolution()

Answer : Could prove magical and horned, not mythical

Please directly run the Problem file to see result of both model checking and resolution test.

Problem 4: Problem_4_LiarsandTruthTellers.py

For modelchecking:

The knowledge base is return by **modelcheckKBParta(), modelcheckKBPartb()** for part a and part b, respectively.

To see the result, run **modelcheckingParta() and modelcheckingPartb()**

For resolution:

The knowledge base is return by **resolutionKBParta()**, **resolutionKBPartb().**

Test code:

testParta(); testPartb()

Answer: [False, False, True] for part a, [True, False, False] for part b

Part a model check Screen shot:

```
Amy is False, Bob is False, Cal is True,
 Amy= Cal and Amy   is True
 Bob=~Cal  is True
 Cal= Bobor~Amy   is True
```

Part b model check Screen shot:

```
Amy is True, Bob is False, Cal is False,
 Amy=~Cal  is True
 Bob= Amy and Cal   is True
 Cal= Amy and Cal   is True
```

Please directly run the Problem file to see result of both model checking and resolution test.

Problem 5: Problem_5_MoreliarsandTruthTellers.py

For modelchecking:

Establish all model checking knowledge base sentences in createKB(),

Here I use a **help function** to establish both model check and resolution knowledge base as according to the info in problem, **the format of all sentences and clauses are same**. **CreateSentence() and createCNFSentence()**

To see the result, please **run the modelchecking()** function

Here the **answer** I get is **only jay and kay are truthful**, **all others are liars**

For resolution:

The resolution knowledge base in created in **createResolutionKB()**

Directly run Problem file and you could see the test of PL_Resolution algorithm

Test code: **testResolution()**

(ie:separately add amy and negative amy to resolution knowledge base and see the result)

Please directly run the Problem file to see result of both model checking and resolution test.

Problem 6: Problem_6_DoosofEnlightenment.py

For modelchecking:

The knowledge base is returned by **modelcheckingKBParta(), modelcheckingKBPartb()** for parta and partb, respectively.

Directly run **modelcheckingParta() and modelcheckingPartb()**, you could see all models that satisfies the knowledge base.

For resolution:

The knowledge base is returned by **resolutionKBParta() and resolutionKBPartb().**

Test code:

testParta();testPartb

Answer: Part a: Choose Door **X**(Could see this from both model checking and PL resolution, in all model X is True, and in PL resolution test, only X is true.)

Part b: Choose Door **X** could be proved

Description of part b :

As there are two fragment sentences, I choose to **add another two people** to fill in these two sentences. These two people are called **T, Q**.

So the two sentences becomes

C: A and **T** are both knights.

G: If C is knight, so it is with **Q**.

Based on A and H ‘s statements and above two sentences, the result shows that knowledge base entail Door X(Both in model checking and resolution).

Appendix: CNF Format sentence for each problem:

Problem 1:

(1) P (2) !P or Q

Problem 2:

(1)!P_1_1, (2)P_1_2 or P_2_1 or !B_1_1, (3)B_1_1 or !P_1_2, (4)B_1_1 or !P_2_1, (5)!B_2_1 or P_1_1 or P_3_1 or P_2_2, (6)!P_1_1 or B_2_1, (7)!P_2_2 or B_2_1, (8)!P_3_1 or B_2_1, (9)!B_1_1,(10)B_2_1, (11)P_1_2

Problem 3:

(1)!mythical or immortal, (2)mammal or mythical, (3)!immortal or mythical, (4)horned or !immortal,(5)horned or !mammal, (6)magical or !horned

Problem 4:

Part a: (1)Cal or !Amy, (2)!Bob or !Cal, (3)Bob or Cal, (4)Bob or !Cal or !Amy, (5)Cal or !Bob, (6)Cal or Amy

Part b: (1)!Amy or !Cal, (2)Amy or Cal, (3)Amy or !Bob, (4)!Bob or Cal, (5)Bob or !Amy or !Cal, (6)Bob or !Cal, (7)!Bob or Cal

Problem 5: in clause foramt

{'!amy', 'hal'}

{'ida', '!amy'}

{'!hal', 'amy', '!ida'}

{'!bob', 'amy'}

{'!bob', 'lee'}

{'!amy', '!lee', 'bob'}

{'!cal', 'bob'}

{'gil', '!cal'}

{'cal', '!bob', '!gil'}

{'!dee', 'eli'}

{'!dee', 'lee'}

{'dee', '!eli', '!lee'}

{'cal', '!eli'}

{'!eli', 'hal'}

{'!hal', 'eli', '!cal'}

{'dee', '!fay'}

{'ida', '!fay'}

{'fay', '!ida', '!dee'}

{'!eli', '!gil'}

{'!gil', '!jay'}

{'gil', 'jay', 'eli'}

{'!hal', '!fay'}

{'!hal', '!kay'}

{'fay', 'kay', 'hal'}

{'!ida', '!gil'}

{'!ida', '!kay'}

{'ida', 'gil', 'kay'}

{'!amy', '!jay'}

{'!cal', '!jay'}

{'cal', 'jay', 'amy'}

{'!dee', '!kay'}

{'!fay', '!kay'}

{'dee', 'kay', 'fay'}

{'!lee', '!bob'}

{'!lee', '!jay'}

{'jay', 'lee', 'bob'}

Problem 6: in clause format

Part a: [Y or X or Z or W, !A or X, A or !X, Y or Z or !B, !Y or B, B or !Z, A or !C, !C or B, !A or C or !B, X or !D, Y or !D, !Y or !X or D, X or !E, Z or !E, !X or !Z or E, E or D or !F, !D or F, !E or F, !C or !G or F, C or G, G or !F, A or !H or !G, H or G, H, !A or H]

Part b: [Y or X or Z or W, !A or X, A or !X, A or !H or !G, H or G, H, !A or H, A or !C, !C or T, !A or C or !T, Q or !C or !G, C or G, !Q or G]