Dashboard / My courses / COSC367-2020S2 / Weekly quizzes / 3. Propositional inference

Started on	Thursday, 30 July 2020, 5:47 PM
State	Finished
Completed on	Friday, 7 August 2020, 5:06 PM
Time taken	7 days 23 hours
Marks	2.14/2.20
Grade	**97.47** out of 100.00

Information

The syntax of propositional definite clauses

We follow the following syntactic rules to describe knowledge bases containing propositional definite clauses.

1. A knowledge base is a collection of zero or more propositional definite clauses (PDCs).
2. Atoms must start with a lower case letter.
3. Each PDC must have a head. The head is an atom.
4. A PDC can optionally have a body. If a body is present, the `:-` symbol (which means 'if') follows the head. The body comes after this symbol. The body is one or more atoms conjuncted by `,` symbol. For example `p :- q, r.` means "p if (q and r)" or equivalently "(q and r) implies p".
5. Every PDC must end with a period.

Question **1**

Correct

Mark 0.14 out of 0.20

Consider the following knowledge base.

```
p :- q.
a :- b.
b.
```

How many distinct interpretation functions are there for this knowledge base? 16 ✓

How many models does this knowledge base have? 3 ✓

In how many models is `a` true? 3 ✓

In how many models is `b` true? 3 ✓

In how many models is `p` true? 2 ✓

In how many models is `q` true? 1 ✓

Your answer is correct.

You have correctly answered 6 part(s) of this question.

Information

Reading knowledge bases

In the following programming questions, your program will need to read in a knowledge base in the form of a string and perform automatic inference. If you wish you can use the following generator function to read the knowledge base.

```
import re

def clauses(knowledge_base):
    """Takes the string of a knowledge base; returns an iterator for pairs
    of (head, body) for propositional definite clauses in the
    knowledge base. Atoms are returned as strings. The head is an atom
    and the body is a (possibly empty) list of atoms.

    -- Kourosh Neshatian - 31 Jul 2019

    """
    ATOM   = r"[a-z][a-zA-z\d_]*"
    HEAD   = rf"\s*(?P<HEAD>{ATOM})\s*"
    BODY   = rf"\s*(?P<BODY>{ATOM}\s*(,\s*{ATOM}\s*)*)\s*"
    CLAUSE = rf"{HEAD}(:-{BODY})?\."
    KB     = rf"^({CLAUSE})*\s*$"

    assert re.match(KB, knowledge_base)

    for mo in re.finditer(CLAUSE, knowledge_base):
        yield mo.group('HEAD'), re.findall(ATOM, mo.group('BODY') or "")
```

You can use `list(clauses(a_knowledge_base_str))` to see what it returns and whether it is useful to you.

Please note that the function is *not* provided on the server. If you decide to use this function, include it in your code.

▶ [Note: If you are working with older versions of Python at home you can click here to use the following which works for Python 3.5 and below.

Question **2**

Correct

Mark 1.00 out of 1.00

Write a function `forward_deduce` that takes the string of a knowledge base containing propositional definite clauses and returns a (complete) set of atoms (strings) that can be derived (to be true) from the knowledge base.

Note: remember to include the `clauses` generator function (and `import re`) if necessary.

For example:

Test	Result

`from student_answer import forward_deduce`
 	
`kb = """`
`a :- b.`
`b.`
`"""`
 	
`print(", ".join(sorted(forward_deduce(kb))))`
	`a, b`

`from student_answer import forward_deduce`
 	
`kb = """`
`good_programmer :- correct_code.`
`correct_code :- good_programmer.`
`"""`
 	
`print(", ".join(sorted(forward_deduce(kb))))`	

`from student_answer import forward_deduce`
 	
`kb = """`
`a :- b, c.`
`b :- d, e.`
`b :- g, e.`
`c :- e.`
`d.`
`e.`
`f :- a,`
`     g.`
`"""`
 	
`print(", ".join(sorted(forward_deduce(kb))))`
	`a, b, c, d, e`

Answer: (penalty regime: 0, 15, ... %)

```
import re

def clauses(knowledge_base):
    """Takes the string of a knowledge base; returns an iterator for pairs
    of (head, body) for propositional definite clauses in the
    knowledge base. Atoms are returned as strings. The head is an atom
    and the body is a (possibly empty) list of atoms.

    -- Kourosh Neshatian - 31 Jul 2019

    """
    ATOM   = r"[a-z][a-zA-z\d_]*"
    HEAD   = rf"\s*(?P<HEAD>{ATOM})\s*"
    BODY   = rf"\s*(?P<BODY>{ATOM}\s*(,\s*{ATOM}\s*)*)\s*"
    CLAUSE = rf"{HEAD}(:-{BODY})?\."
    KB     = rf"^({CLAUSE})*\s*$"

    assert re.match(KB, knowledge_base)

    for mo in re.finditer(CLAUSE, knowledge_base):
        yield mo.group('HEAD'), re.findall(ATOM, mo.group('BODY') or "")


def forward_deduce(knowledge_base):
    """string of a knowledge base containing propositional definite clauses and returns a (complete)
    list1 = list(clauses(knowledge_base))
    #list1 is a list of clauses
    true_set = set()
    finished = False
    while (finished == False):
        current_true_set = true_set.copy()
        for i in list1:
            #for every clause if the items in the list to the right are in the set, then i can add i
            #if one full for loop and no change, then terminate
            #select clause in kb such that b in set c for all i and h not in c
            #C := C U {h}
            if i[1] == [] and (i[0] not in current_true_set):
                current_true_set.add(i[0])
                break
            else:
                if (set(i[1]).issubset(true_set)):
                    current_true_set.add(i[0])
                else:
                    continue
        if (current_true_set == true_set):
            finished = True
        else:
            true_set = current_true_set
    return true_set


    #return list1



```

	Test	Expected	Got	

✓	`from student_answer import forward_deduce`
	
	`kb = """`
	`a :- b.`
	`b.`
	`"""`
	
	`print(", ".join(sorted(forward_deduce(kb))))`
		a, b	a, b	✓

✓	`from student_answer import forward_deduce`
	
	`kb = """`
	`good_programmer :- correct_code.`
	`correct_code :- good_programmer.`
	`"""`
	
	`print(", ".join(sorted(forward_deduce(kb))))`
				✓

✓	`from student_answer import forward_deduce`
	
	`kb = """`
	`a :- b, c.`
	`b :- d, e.`
	`b :- g, e.`
	`c :- e.`
	`d.`
	`e.`
	`f :- a,`
	`     g.`
	`"""`
	
	`print(", ".join(sorted(forward_deduce(kb))))`
		a, b, c, d, e	a, b, c, d, e	✓

✓	`from student_answer import forward_deduce`
	
	`kb = ""`
	`print(", ".join(sorted(forward_deduce(kb))))`
				✓

✓	`from student_answer import forward_deduce`
	
	`kb = """`
	`a.`
	`z.`
	`"""`
	`print(", ".join(sorted(forward_deduce(kb))))`
		a, z	a, z	✓

	Test	Expected	Got	

✓	`from student_answer import forward_deduce`
	
	`kb = """`
	`wet :- is_raining.`
	`wet :- sprinkler_is_going.`
	`wet.`
	`"""`
	
	`print(len(forward_deduce(kb)))`
		1	1	✓

✓	`from student_answer import forward_deduce`
	
	`kb = """`
	`this_is_true :- this_is_true.`
	`"""`
				✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

Question **3**

Correct

Mark 1.00 out of 1.00

Write a class `KBGraph` that poses a knowledge base and a query as a graph. The query will be a set of atoms (strings). If you wish you can use the template provided in the answer box. You must also provide an implementation of `DFSFrontier`. You can simply copy this across from the graph search quiz if you have answered that question.

The graph class will not be tested on the order of edges; you can generate edges in whatever order you wish. The input knowledge base is guaranteed to not have cyclic clauses. For example the following is NOT an example input:

```
a :- b.
b :- c.
c :- a.
d.
```

Notes

1. See the top-down proof procedure, answer clauses, and the example search tree for an SLD resolution in the lecture notes.
2. You need to think about the right representation for nodes. Since all *answer clauses* are of the form `yes :- a_body`, you can factor out the 'yes' part and only represent the body.
3. After answering this question, as an additional exercise for yourself, improve your program such that knowledge bases with cycles can be handled.
4. While the question is not asking for this, note that the proof can produced by printing the path (if there is one). Therefore it is useful to have meaningful labels for the edges of the graph. Also the length of the proof will depend on the search strategy (type of frontier).

For example:

Test	Result

```
from search import *
from student_answer import KBGraph, DFSFrontier

kb = """
a :- b, c.
b :- d, e.
b :- g, e.
c :- e.
d.
e.
f :- a,
     g.
"""

query = {'a'}
if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
    print("The query is true.")
else:
    print("The query is not provable.")
```
	The query is true.

```
from search import *
from student_answer import KBGraph, DFSFrontier

kb = """
a :- b, c.
b :- d, e.
b :- g, e.
c :- e.
d.
e.
f :- a,
     g.
"""

query = {'a', 'b', 'd'}
if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
    print("The query is true.")
else:
    print("The query is not provable.")
```
	The query is true.

Test	Result

```
from search import *
from student_answer import KBGraph, DFSFrontier

kb = """
all_tests_passed :- program_is_correct.
all_tests_passed.
"""

query = {'program_is_correct'}
if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
    print("The query is true.")
else:
    print("The query is not provable.")
```
	The query is not provable.

```
from search import *
from student_answer import KBGraph, DFSFrontier

kb = """
a :- b.
"""

query = {'c'}
if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
    print("The query is true.")
else:
    print("The query is not provable.")
```
	The query is not provable.

Answer: (penalty regime: 0, 15, ... %)

Reset answer

```
import re
from search import *


def clauses(knowledge_base):
    """Takes the string of a knowledge base; returns an iterator for pairs
    of (head, body) for propositional definite clauses in the
    knowledge base. Atoms are returned as strings. The head is an atom
    and the body is a (possibly empty) list of atoms.

    -- Kourosh Neshatian - 31 Jul 2019

    """
    ATOM   = r"[a-z][a-zA-z\d_]*"
    HEAD   = rf"\s*(?P<HEAD>{ATOM})\s*"
    BODY   = rf"\s*(?P<BODY>{ATOM}\s*(,\s*{ATOM}\s*)*)\s*"
    CLAUSE = rf"{HEAD}(:-{BODY})?\."
    KB     = rf"^({CLAUSE})*\s*$"

    assert re.match(KB, knowledge_base)

    for mo in re.finditer(CLAUSE, knowledge_base):
        yield mo.group('HEAD'), re.findall(ATOM, mo.group('BODY') or "")


class KBGraph(Graph):
    def __init__(self, kb, query):
        self.clauses = list(clauses(kb))
        self.query = query


    def starting_nodes(self):
        return[(self.query)]
        #node_list = []
        #for items in self.query:
            #node_list.append(items)


        #return node_list

    def is_goal(self, node):
        " ** i am checking that the node is a goal state e.g it propagates to a [] or is true**"
        return (len(node) == 0)


    def outgoing_arcs(self, tail_node):
        " ** COMPLETE ** "
        #print("tail node is")
        #print(tail_node)
        new_nodes = []
        new_arcs = []

        for node in tail_node:
            #for every node in tail nodes
            for clause in self.clauses:
                #check if it has a corresponding clause
                if node == clause[0]:
                    #if the node matches the clause in the kb
                    #append possible node branch to new_nodes
                    new_nodes.append(clause[1])

        for new_node in new_nodes:
            #for every new possible path added by kb
            new_arcs.append(Arc(tail_node, new_node, str(tail_node) + "->" + str(new_node), 0))

        #print("new arcs are:")
        #print(new_arcs)
        return new_arcs


class DFSFrontier(Frontier):
    """Implements a frontier container appropriate for depth-first
    search."""

    def __init__(self):
        """The constructor takes no argument. It initialises the
        container to an empty stack."""
        self.container = []

    def add(self, path):
        #the container from a DFS should be LIFO so i pop from the right append to the right
        self.container.append(path)
        #raise NotImplementedError # FIX THIS

    def __iter__(self):
        """The object returns itself because it is implementing a __next__
        method and does not need any additional state for iteration."""
        return self

    def __next__(self):
        if len(self.container) > 0:
            return self.container.pop()
            #raise NotImplementedError # FIX THIS return something instead
        else:
            raise StopIteration   # don't change this one

```

	Test	Expected	Got	

✓	`from search import *`
	`from student_answer import KBGraph, DFSFrontier`
	
	`kb = """`
	`a :- b, c.`
	`b :- d, e.`
	`b :- g, e.`
	`c :- e.`
	`d.`
	`e.`
	`f :- a,`
	`     g.`
	`"""`
	
	`query = {'a'}`
	`if next(generic_search(KBGraph(kb, query),`
	`DFSFrontier()), None):`
	`    print("The query is true.")`
	`else:`
	`    print("The query is not provable.")`
		The query is true.	The query is true.	✓

	Test	Expected	Got	

✓	from search import *
	from student_answer import KBGraph, DFSFrontier
	
	kb = """
	a :- b, c.
	b :- d, e.
	b :- g, e.
	c :- e.
	d.
	e.
	f :- a,
	     g.
	"""
	
	query = {'a', 'b', 'd'}
	if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
	    print("The query is true.")
	else:
	    print("The query is not provable.")
		The query is true.	The query is true.	✓

✓	from search import *
	from student_answer import KBGraph, DFSFrontier
	
	kb = """
	all_tests_passed :- program_is_correct.
	all_tests_passed.
	"""
	
	query = {'program_is_correct'}
	if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
	    print("The query is true.")
	else:
	    print("The query is not provable.")
		The query is not provable.	The query is not provable.	✓

✓	from search import *
	from student_answer import KBGraph, DFSFrontier
	
	kb = """
	a :- b.
	"""
	
	query = {'c'}
	if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
	    print("The query is true.")
	else:
	    print("The query is not provable.")
		The query is not provable.	The query is not provable.	✓

✓	from search import *
	from student_answer import KBGraph, DFSFrontier
	
	kb = ""
	
	query = {'proposition'}
	if next(generic_search(KBGraph(kb, query), DFSFrontier()), None):
	    print("The query is true.")
	else:
	    print("The query is not provable.")
		The query is not provable.	The query is not provable.	✓

Passed all tests! ✓

Correct

Marks for this submission: 1.00/1.00.

◄ 2. Cost-sensitive search, pruning, and heuristics

Jump to...

4. Declarative programming with Prolog (i) ►