

PROJECT ASSIGNMENT II

UNIVERSITY KNOWLEDGE GRAPH & UNIVERSITY Chatbot

A Report
Presented to
The Department of Electrical & Computer Engineering
Concordia University

In Partial Fulfillment
of the Requirements
of COMP 474/6741

by

Filip Jodoin	ID: 27879105
Ling Zhi Mo	ID: 40024810
Sean O'Connor	ID: 40016165

Concordia University
April 24th, 2020

ABSTRACT:

The first phase of our project was to gather information on all Concordia University courses. This information was used to generate a Knowledge Graph. The second phase of our project revolves around creating a Chatbot which uses the Knowledge Graph information from the first phase as its dataset for answering questions. This report will discuss the details regarding the three steps (obtaining, shaping and testing data) of phase one, and the logic behind the steps of designing our Chatbot in phase two.

OBJECTIVES:

- ❖ **Phase I:** Creating a Knowledge Graph with a dataset complete with all available courses (along with their respective details) at Concordia University. A Querying section must be implemented in order to test the validity of our Knowledge Graph.
- ❖ **Phase II:** Creating a Chatbot using the Knowledge Graph generated in Phase I. The Chatbot is capable of answering a variation (similarity of 75%) of 4 typical questions.

TASKS:

PHASE I

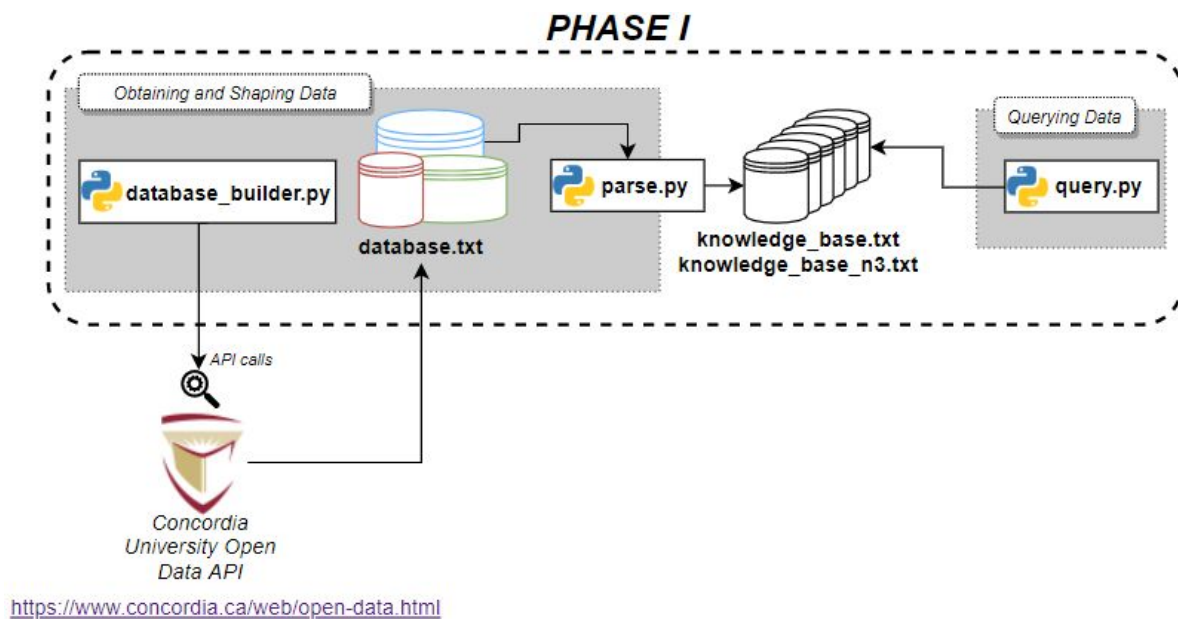


Figure 1: Phase I diagram

❖ **Vocabulary:**

Description how you modeled the schema for your knowledge base, including the vocabularies you reused, any vocabulary extensions you developed, etc. Give brief justifications where appropriate (e.g., choice of existing vocabulary).

Fixes from assignment #1:

- rdfs:seeAlso has been documented in Table 1
- ex:Concordia instance has been added to this report and the generated knowledge base
- focu:offeredAt was defined in relation to the ex:Concordia instance

Table 1: Vocabulary

RDF	type	For defining classes and properties
	Property	
RDFS	Class	For defining classes
	subClassOf	For extending class definitions
	domain	For defining properties
	range	
	seeAlso	To associate Concordia University with its dbpedia entry.
FOAF	firstName	For the personal information of students in our knowledge base
	lastName	
	mbox	
	Person	A student is a person
	topic	Property used to relate a topic to a university course
DBO	document	For defining our transcripts document seemed a fitting concept.
	type	For use of the Public_university type
DBR	Course_(education)	Suitable for broadly defining the courses offered at universities
	Public_university	Suitable definition for Universities
XSD	string	Literal strings serve as the object of several properties we have generated.

We re-used the focu schema <<http://focu.io/schema#>> from the lab exercises contain our classes and properties and the ex schema <<http://example.org/>> to contain instances of these classes and other existing classes. Their definitions and details concerning their usage are given below. As per the assignment specifications each class and property created for our vocabulary possesses an `rdfs:label` and `rdfs:comment` which are omitted for brevity.

focu:Student (class):

`rdf:type` *rdfs:Class*
`rdfs:subClassOf` *foaf:Person*

An entity defined by us to represent students in our knowledge base. Based off of similar ones created during the lab exercises and in-class worksheets.

ex:Student (instance):

<code>rdf:type</code> <i>focu:Student</i>	<code>foaf:firstName</code>
<code>focu:hasRecord</code>	<code>foaf:lastName</code>
	<code>foaf:mbox</code>

Students possess URIs of the format <<http://example.org/X>>, where X is an integer value generated sequentially during knowledge base creation. Students possess first names, last names and email addresses as defined in the FOAF vocabulary. We define an additional property *hasRecord* (described below). **A student may have 0 or more hasRecord predicates**, one per *Record*.

focu:hasRecord (property):

<code>rdf:type</code> <i>rdf:Property</i>	<code>rdfs:domain</code> <i>focu:Student</i>
	<code>rdfs:range</code> <i>focu:Record</i>

A property defined by us to associate a single record to a single student. Refer to *Record* below for more information.

focu:subject	focu:catalog	focu:subject_catalog	(properties)
---------------------	---------------------	-----------------------------	---------------------

The Concordia opendata API from which we retrieve course information returns the discipline of the course (i.e. COMP) and the number of the course (i.e. 474) as subject and catalog, respectively. Each of these three terms are properties possessed by other classes in our knowledge base. All three are kept separate as they are needed or convenient when querying or navigating our graph.

focu:Record (class):

rdf:type *dbo:document*

An entity representing a single transcript for a particular class. Refer to the **instance** definition below for more information.

ex:Record (instance):

rdf:type *focu:Record*

focu:semester

focu:subject_catalog

focu:grade

Records possess URIs of the format <http://example.org/recordX>, where X is an integer value generated sequentially during knowledge base creation. Record instances possess actual data for a given record associated grades (A, B, ... , F) the subject_catalog (i.e. COMP474) and a semester (annual i.e. 2001, 2002, ... , 2019). Each of these data points are defined in focu as properties possessed by Records.

ex:Course (instance):

rdf:type *dbo:Course_(education)*

focu:subject

foaf:topic

focu:catalog

focu:offeredAt

Our course instances are identified by the combination of their subject (i.e. COMP) and catalog (i.e. 474) to create their URI <http://example.org/COMP474>. The terms subject and catalog were detailed previously in this report. **A course may have 0 or more foaf:topic predicates**, one for each topic. Topics are discovered by dbpedia Spotlight using the course description and title to offer more information about a course. The focu:OfferedAt property associates a course instance to the Educational institution at which it is offered.

ex:Concordia (instance):

dbo:type *dbr:Public_university*

foaf:name

rdfs:seeAlso *dbr:Concordia_University*

A singular instance of Concordia is defined in schema.ttl and carried forward into knowledgebase.txt by parsey.py

The instance is associated with the dbpedia entry for Concordia University and primarily serves as the object of each Course instance's focu:offeredAt property.

❖ **Knowledge Base Construction:**

*Describe (a) your dataset and (b) your process and developed tools for populating the knowledge base from the dataset.
Describe how to run the tools to create the knowledge base. Explain your process for linking entities to DBpedia.*

(a) Describe dataset

- ❖ The dataset is stored in a local file “database.txt”.
- ❖ The dataset contains 7050 data points, where each datapoint contains information on a course from Concordia University.
- ❖ The datapoints are in a string dictionary format, as per our request (during API call process; consult (b), Database_builder.py section below)
- ❖ Each datapoint has the following dictionary structure:

i.e.

```
{  
  "000026": {  
    "title": "Financial and Managerial Accounting",  
    "subject": "ACCO",  
    "catalog": "220",  
    "career": "UGRD",  
    "classUnit": "3.00",  
    "prerequisites": "Prerequisites of this course",  
    "crosslisted": "null",  
    "description": "Description of this course",  
    "sub_catalog": "ACCO 220"  
  }  
}
```

(b) Describe developed tools

- ❖ *database_builder.py* (Building dataset)
 - **Step 1 (line 23):** CONCORDIA’S OPEN DATA api call (api_call())
 - In this step, we are using Concordia’s OpenData api to gather all available course information. Specifically, we are requesting “Course Description” which contains a course unique id with the course description, and “Course Catalog” which contains the same course unique id with its catalog information (i.e course title, course subject, course number, etc.). To accomplish this http request, we used python module “requests”. Once the api call returns the results string, we will build the dictionary objects (the datapoint), and dump the dictionary to the local file “database.txt”.
 - **Step 2 (line 125):** Crawling Concordia’s undergraduate calendar

(undergrad_scheme())

- In this step, we are updating our dataset for the datapoints that are undergraduate courses and are missing a specific “description”. To accomplish this goal, we first used “requests” to get Concordia’s undergraduate program’s main page, which contains all subjects’ undergraduate calendar webpage that has course description. Next, we will request each url by using python module “requests_futures”, an extension module of “request”, and parse the return web content to retrieve the missing undergraduate course description. Finally, we will update our database file with the new dataset.

➤ **Step 3 (line 194):** Crawling Concordia’s graduate calendar

(graduate_scheme())

- In this step, we are updating our dataset for the datapoints that are graduate courses and are missing a specific “description”. To accomplish this goal, we first used “requests” to get Concordia’s graduate program summary page, which contains every graduate program main page and corresponding graduate program course calendar page that has a graduate course description. Similar to the above step, we used “requests_futures” to make a http request to each program calendar, retrieve required information and update our database.

❖ *parse.py* (Generating Knowledge Graph)

➤ **Step 1 (line 61):** Extract relevant entries from each database.txt datapoint

- Our project’s scope only requires certain fields for each course, therefore it is important to not include unnecessary information into our knowledge graph.

➤ **Step 2 (line 89):** Obtain Topics related to each course

- In order to obtain all topics related to a course, the title and description of said course must be passed to DBpedia Spotlight. DBpedia Spotlight is hosted locally through docker and will return all tagged topics which were generated. If a course DOES NOT return a topic, it is deemed irrelevant and is discarded.

➤ **Step 3 (line 135):** Add all valid courses to Knowledge Graph

- Now that all the valid courses have been parsed and separated from the invalid and unstructured courses, we must populate the Knowledge Graph.

➤ **Step 4 (line 160):** Generating Students and adding them to Knowledge Graph

- In this step, we used a custom function (create_students() line 23) that will generate 100 students where each one has a unique id, first name, last name, email address and a randomized record list. Each record in the record list contains the following information: year of complete, course (subject and catalog), and corresponding grade.

❖ **Queries:**

Describe your translation of the knowledge base queries below into SPARQL. Provide (brief) example output for each query.

❖ *query.py* (Questioning the Knowledge Graph)

The following image is the menu in the query.py module;

```
Enter 1 for number of triple in Knowledge Base
Enter 2 for total number of students, courses and topics
Enter 3 for all topics related to a given course
Enter 4 for all completed courses for a given student
Enter 5 for all students familiar with a given topic
Enter 6 for all topics familiar to a given student
> █
```

Figure 1: query.py menu

1. Total number of triples in the KB

```
> 1
Number of triple(s) in Knowledge Base: 49619
```

Figure 2: Query 1 output; Number of Triples in Knowledge Graph

- Line 33; simply prints the length of our graph g. No SPARQL query required.

2. Total number of students, courses, and topics

```
> 2
100 Students
4595 Courses
6482 Topics
```

Figure 3: Query 2 output; number of Students, Courses and Topics in Knowledge Graph

- Line 38; Performs 3 different SPARQL queries. Essentially, the Students and Courses use the COUNT(*) AS ?[variable_name] keyword in the SPARQL query to obtain their values. The Topics count query first populates a list with all topics obtained from its SPARQL query; leverages DISTINCT and GROUP BY to ensure the topics list is populated with unique entries. The length of this list is the number of Topics in our graph.

3. For a course c, list all covered topics using their (English) labels and their link to DBpedia

```
> 3
Please enter the Subject Catalog (i.e. ACCO435) of a Course
> CCMP474

- Topic Label:
College of Osteopathic Medicine of the Pacific
- Dbpedia URL:
http://dbpedia.org/resource/College_of_Osteopathic_Medicine_of_the_Pacific

- Topic Label:
Artificial intelligence
- Dbpedia URL:
http://dbpedia.org/resource/Artificial_intelligence

- Topic Label:
Computer simulation
- Dbpedia URL:
http://dbpedia.org/resource/Computer_simulation

- Topic Label:
Expert system
- Dbpedia URL:
http://dbpedia.org/resource/Expert_system

- Topic Label:
Blackboard
- Dbpedia URL:
http://dbpedia.org/resource/Blackboard

- Topic Label:
Coen River
- Dbpedia URL:
http://dbpedia.org/resource/Coen_River

- Topic Label:
Conflict resolution
- Dbpedia URL:
http://dbpedia.org/resource/Conflict_resolution

- Topic Label:
Knowledge acquisition
- Dbpedia URL:
http://dbpedia.org/resource/Knowledge_acquisition
```

Figure 4: Query 3 output; Topics and DBpedia URLs associate with a given course (subject + catalog)

- Line 77; With the course (subject + catalog) given as an input, the SPARQL query is done very fast. For the Topics, it only needs to search for all rdf:topic associated with the course. The Labels are obtained by parsing the rdf:topic obtained previously. Essentially, we would attach a Label to each Topic in the Knowledge Graph by calling the DBpedia spotlight candidate API option... unfortunately it is not included

in the docker image we have hosting, therefore we have not found an efficient way to fetch the labels via API without getting blocked.

4. For a given student, list all courses this student completed, together with the grade

```
> 4
Please enter the ID of a given Student
> 78
Complete Courses for [Student 78]
2010 http://example.org/MAST898C D
2016 http://example.org/ENGL252 B
2009 http://example.org/DART262 A
2003 http://example.org/CEPS1024E B
2017 http://example.org/ELEC6391 B
1998 http://example.org/HISW242 B
1991 http://example.org/EXCI357 A
2011 http://example.org/DRAW410 C
2008 http://example.org/MANA298 A
1995 http://example.org/FBRS260 C
```

Figure 5: Query 4 output; courses (semester, subject+catalog and grade) completed by given student id

- Line 100; By giving the query a student id, the SPARQL query will first query for the records of the given student. Each record contains a semester, a course (subject + catalog) and a grade; said information is retrieved and only printed if the grade obtained is NOT "F".

5. For a given topic, list all students that are familiar with the topic (i.e., took, and did not fail, a course that covered the topic)

```
> 5
Please enter the topic of interest
> Psychology
http://example.org/64
http://example.org/35
http://example.org/100
http://example.org/67
http://example.org/98
http://example.org/22
http://example.org/23
http://example.org/35
http://example.org/42
http://example.org/22
http://example.org/90
http://example.org/6
http://example.org/35
http://example.org/25
http://example.org/32
http://example.org/11
```

Figure 6: Query 5 output; return student id of students who are familiar with a given Topic

- Line 124; The SPARQL query starts by obtaining the course (subject + catalog) associated with the given Topic (input). All of the records connected to said course are then fetched. Records with grades which are NOT “F” (by using FILTER) will be kept for a final query; return student id of belonging to records kept.

6. For a student, list all topics (no duplicates) that this student is familiar with (based on the completed courses for this student that are better than an “F” grade)

```
> 6
Please enter the ID of a given student
> 45
http://dbpedia.org/resource/Hebrew_language
http://dbpedia.org/resource/Classical_music
http://dbpedia.org/resource/Syntax
http://dbpedia.org/resource/Academic_grading_in_the_United_States
http://dbpedia.org/resource/Economic_efficiency
http://dbpedia.org/resource/Monopoly
http://dbpedia.org/resource/Microeconomics
http://dbpedia.org/resource/Satire
http://dbpedia.org/resource/Maria_Edgeworth
http://dbpedia.org/resource/Japanese_novel
http://dbpedia.org/resource/Ann_Radcliffe
http://dbpedia.org/resource/Charles_Burney
http://dbpedia.org/resource/Charles_Dickens
http://dbpedia.org/resource/Literary_genre
http://dbpedia.org/resource/Gothic_fiction
http://dbpedia.org/resource/Literary_realism
http://dbpedia.org/resource/Christian
http://dbpedia.org/resource/Jesus
http://dbpedia.org/resource/Bible
```

Figure 7: Query 6 output; Display all Topics familiar to given student id

- Line 148; SPARQL query obtains all records associated with the student id given. If a record’s grade is not “F”, then it proceeds to the next query. From here, SPARQL will extract the course (subject + catalog) of said record. Finally, all topics linked to the course will be considered as familiar topics.

TASKS:

PHASE II

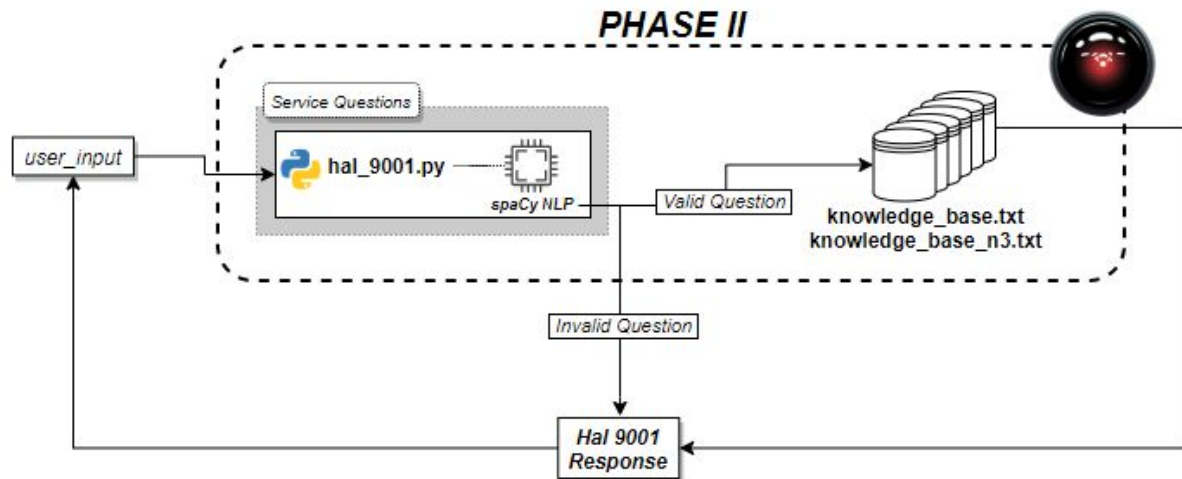


Figure 8: Phase II diagram

❖ University Chatbot:

You have to develop a new Python-based Chatbot interface to your knowledge base from A1 (i.e., a grounding-based bot). It has to be able to answer (at least) the following questions:

1. "What is the <course> about?" E.g., "What is COMP 474 about?": provides the course description as an answer.
2. "Which courses did <Student> take?" Lists course names/subject/numbers, like "COMP 474 Intelligent Systems" and the achieved grade/term.
3. "Which courses cover <Topic>?" E.g., "Which courses cover Expert Systems?": lists all courses that include this topic.
4. "Who is familiar with <Topic>?" E.g., "Who is familiar with Natural Language Processing?": maps to Query #5 from A1.

An explanation of the method of design for the Chatbot is necessary.

- ### ❖ Chatbot Design Method:
- As we were tasked to design a Chatbot capable of answering four major questions, our Chatbot leverages Natural Language Processing (spaCy libraries). A spaCy feature allows the Chatbot to weigh any user-input to one of the four major questions. If the user-input is greater or equal to 75% similar to one of the questions, the Chatbot will accept the input and query the Knowledge Graph.

❖ *hal_9001.py* (University Chatbot)

- **4 major questions (line 12 - 15):** When referring to major questions, Figure 9 are those hard coded into *hal_9001.py*

```
12 question_1 = nlp("What is about?")
13 question_2 = nlp("Which courses did take")
14 question_3 = nlp("Which courses cover")
15 question_4 = nlp("Who is familiar with")
```

Figure 9: Major Chatbot Questions

➤ **Step 1 (line 202):** Obtain input from the user, determine the type of question being asked and parse the important information for Step 2.

- Line 207 of `__main__` will wait for the user to input a question through the command line. Once entered, the `determine_question` function will process this user-input. Figure 10 below shows that on line 27, the entire question is treated as a token using the `nlp = spacy.load("en_core_web_md")` variable declared on line 11. This token is weighed against the major questions by using the spacy similarity feature. If the user-input is 75% (or higher) similar to a major question, the input is processed on line 31, 41, 51 or 61. Regex is then used to parse the question in order to return the type of question and the question details necessary for querying the Knowledge Graph; line 38, 48, 59, or 69. If no match is found, Hal 9001 notifies the user.

```
20 ##### FUNCTIONS #####
21 def determine_question(input_text):
22     """
23     Uses NLP to determine with 75% certainty which question is being asked
24     :param input_text: string
25     :return: string, string
26     """
27     text_doc = nlp(input_text)
28     for token in text_doc:
29         if token.lower_ in ["about"]:
30             question_similarity = text_doc.similarity(question_1)
31             if question_similarity > 0.75:
32                 # print("[", text_doc.text, "|", question_1.text, "|", question_similarity)
33                 course = re.match(r'.* is (.*) about', input_text)
34                 if course==None:
35                     print("Hal_9001 > Your question does not make sense... Try again, human.")
36                     match=True
37                 else:
38                     return("question_1", course.group(1))
39         elif token.lower_ in ["take"]:
40             question_similarity = text_doc.similarity(question_2)
41             if question_similarity > 0.75:
42                 # print("[", text_doc.text, "|", question_2.text, "|", question_similarity)
43                 student = re.match(r'.* did (.*) take', input_text)
44                 if student==None:
45                     print("Hal_9001 > Your question does not make sense... Try again, human.")
46                     match=True
47                 else:
48                     return("question_2", student.group(1))
49         elif token.lower_ in ["cover"]:
50             question_similarity = text_doc.similarity(question_3)
51             if question_similarity > 0.75:
52                 # print("[", text_doc.text, "|", question_3.text, "|", question_similarity)
53                 topic = re.match(r'.* cover (.*)', input_text)
54                 #print(topic)
55                 if topic==None:
56                     print("Hal_9001 > Your question does not make sense... Try again, human.")
57                     match=True
58                 else:
59                     return("question_3", topic.group(1))
60         elif token.lower_ in ["familiar"]:
61             question_similarity = text_doc.similarity(question_4)
62             if question_similarity > 0.75:
63                 # print("[", text_doc.text, "|", question_4.text, "|", question_similarity)
64                 familiar_topic = re.match(r'.* with (.*)', input_text)
65                 if familiar_topic==None:
66                     print("Hal_9001 > Your question does not make sense... Try again, human.")
67                     match=True
68                 else:
69                     return("question_4", familiar_topic.group(1))
70     return("null", "null")
```

Figure 10: `determine_questions(input_text)` function

➤ **Step 2 (line 211):** If a valid question was given by the user in Step 1, Step 2 proceeds with servicing the question by querying the Knowledge Base, similar to the Query section of Phase I.

- Line 212 of `__main__` calls `query_knowledge_graph(question_type, question_details)` with the results from `determine_question`. As Figure 11 depicts, `question_type` will determine which major question is being processed; which if-statement will be selected.

```

73 def query_knowledge_graph(question_type, question_details):
74     """
75     Query knowledge graph by create an appropriate SPARQL-query format depending on the question_type
76     :param question_type: string
77     :param question_details: string
78     :return:
79     """
80
81     # QUESTION 1
82     if question_type is "question_1":
83
84     # QUESTION 2
85     elif question_type is "question_2":
86
87     # QUESTION 3
88     elif question_type is "question_3":
89
90     # QUESTION 4
91     elif question_type is "question_4":

```

Figure 11: `query_knowledge_graph(question_type, question_details)` function

- **Question 1 “What is <course> about?”**, (line 81): Figure 12 is almost identical to the `query.py` option 3 (refer to Figure 1). Figure 13 shows a typical interaction with Hal 9001 of a Question 1 scenario.

```

81 # QUESTION 1
82 if question_type is "question_1":
83     # question_details holds course name; return all topics related
84     c_sub_cata = question_details
85     target = URIRef("http://example.org/" + c_sub_cata)
86     q_topic = prepareQuery(
87         """SELECT ?topic WHERE {
88             ?c_sub_cata foaf:topic ?topic
89         }""",
90         initNs = {
91             "c_sub_cata": target, "foaf": FOAF, "rdf": RDF, "focu": "http://focu.io/schema#"
92         }
93     )
94     print("Hal_9001 > " + question_details + " is (potentially) about:")
95     for row in g.query(q_topic, initBindings={"c_sub_cata": target}):
96         topic_label = re.match(r'.*/resource/(.*)', str(row[0]))
97         print("    ", topic_label.group(1).replace("_", " "))

```

Figure 12: `query_knowledge_graph` function, Question 1


```

User > What is COMP474 about?
Hal_9001 > COMP474 is (potentially) about:
College of Osteopathic Medicine of the Pacific
Coen River
Expert system
Knowledge acquisition
Conflict resolution
Blackboard
Artificial intelligence
Computer simulation

```

Figure 13: Question 1 output

- **Question 2** “Which courses did <Student> take?”, (line 99): Figure 14 is almost identical to the query.py option 4 (refer to Figure 1), except that Hal 9001 will return all courses, completed and failed. Figure 15 shows a Question 2 interaction with Hal 9001

```

99  # QUESTION 2
100 elif question_type is "question_2":
101     # question_details holds Student ID; return courses Student has completed
102     student_id = question_details
103     target = URIRef("http://example.org/" + student_id)
104     q = prepareQuery(
105         """SELECT ?sem ?sc ?g WHERE {
106             ?s_id focu:hasRecord ?rec .
107             ?rec focu:semester ?sem .
108             ?rec focu:subject_catalog ?sc .
109             ?rec focu:grade ?g .
110         }""",
111         initNs = {
112             "s_id": target, "foaf": FOAF, "rdf": RDF, "focu": "http://focu.io/schema#"
113         }
114     )
115     print("Hal_9001 > Student " + student_id + " has taken the following courses:")
116     for row in g.query(q, initBindings={"s_id": target}):
117         pretty_row = (" %s %s %s" % row).replace("http://example.org/", "")
118         print(pretty_row)

```

Figure 14: *query_knowledge_graph* function, Question 2

```

User > which course did 20 take?
Hal_9001 > Student 20 has taken the following courses:
2019 IMCA400 F
1993 SPAN664 A
2004 ENGR6411 A
2017 SOEN6861 C
2003 PERC384 A
2018 HIST223 D
1996 CEFR702 F
2002 AERO201 D
2003 MECH7511 D
2012 ENVS668 B
2013 SOCI441 C
2016 ELEC6431 B
2018 HIST387B A
1991 FMST200 F
2008 IRST315 B
2017 ARTT398 C
1990 BLDG6861 A
1993 RELI419 C
2002 INDI802 D
1998 THEO204 F

```

Figure 15: Question 2 output

- **Question 3 “Which courses cover <Topic>?”**, (line 120): Figure 16 shows the first part of handling question 3. The first part of code will return if and only if the user's input “Topic” has a match to the knowledge graph. In case that the user's input is not perfect, Figure 17 shows the second part to handle this case. We will first tokenize the user's input, capitalize some of the tokens and parse to the graph again. Some cases are very special, and we assume the perfect input from the user.

```
# QUESTION 3
elif question_type is "question_3":
    # question_details holds a Topic; return courses covering said Topic
    enter = False
    target = URIRef("http://dbpedia.org/resource/" + question_details.replace(" ", "_"))
    q_topic = prepareQuery(
        """SELECT ?c_sub_cata WHERE {
            ?c_sub_cata foaf:topic ?topic
        }""",
        initNs = {
            "topic": target, "foaf": FOAF, "rdf": RDF, "focu": "http://focu.io/schema#"
        }
    )
    if len(g.query(q_topic, initBindings={"topic": target}))>0:
        enter=True
        print("Hal_9001 > The following courses has",question_details+":")
    for row in g.query(q_topic, initBindings={"topic": target}):
        print("      ",row[0].replace("http://example.org/", ""))

    if enter:
        return
```

Figure 16: *query_knowledge_graph* function part 1, Question 3

```
potential_target_list=[]
target_split = question_details.split(" ")
for indexI,i in enumerate(target_split):
    target_string = ""
    for indexJ,j in enumerate(target_split):
        if indexI>=indexJ:
            target_string+=target_split[indexJ].capitalize()
            if indexJ<len(target_split)-1:
                target_string+="_"
        else:
            target_string+=target_split[indexJ].lower()
            if indexJ<len(target_split)-1:
                target_string+="_"
    potential_target_list.append(target_string)

for i in potential_target_list:
    target = URIRef("http://dbpedia.org/resource/" + i)
    q_topic = prepareQuery(
        """SELECT ?c_sub_cata WHERE {
            ?c_sub_cata foaf:topic ?topic
        }""",
        initNs = {
            "topic": target, "foaf": FOAF, "rdf": RDF, "focu": "http://focu.io/schema#"
        }
    )
    enter=False
    if len(g.query(q_topic, initBindings={"topic": target}))>0:
        enter=True
        print("Hal_9001 > The following courses has",question_details+":")
    for row in g.query(q_topic, initBindings={"topic": target}):
        print("      ",row[0].replace("http://example.org/", ""))
    if enter:
        break
```

Figure 17: *query_knowledge_graph* function part 2, Question 3

```

LingMachine:project1 lingzhimo$ python3 hal_9001.py
Hal_9001 > Greetings human, my name is Hal 9001. How may I assist you?
User > What is COMP474 about
Hal_9001 > COMP474 is (potentially) about:
    Artificial intelligence
    Coen River
    College of Osteopathic Medicine of the Pacific
    Knowledge acquisition
    Computer simulation
    Expert system
    Blackboard
    Conflict resolution
User > Which courses cover Expert system
Hal_9001 > The following courses has Expert system:
    COMP474
User > Which courses cover Knowledge acquisition
Hal_9001 > The following courses has Knowledge acquisition:
    COMP474
User > Which courses cover conflict resolution
Hal_9001 > The following courses has conflict resolution:
    COMP6741
    INSE6270
    AHSC680
    POLI215
    HENV670
    SOEN341
    COMP474
    AHSC330
    PSYC374
User >

```

Figure 18: Question 3 output

- **Question 4 “Who is familiar with <Topic>?”**, (line 176): Figure 19 shows the code which handles question 4. The task is very similar to query.py option 5 (Refer to Figure 1). A SPARQL query (line 182) returns all students who have taken a course which features the topic of interest with a non-failing grade. Hal 9001 will reply to the user with the list of students. Figure 20 shows a typical interaction with Hal 9001

```

176 # QUESTION 4
177 elif question_type is "question_4":
178     # question_details holds a Topic; return students familiar with said Topic
179     topic_of_interest = question_details.replace(" ", "_")
180     topic_of_interest = topic_of_interest.replace("?", "")
181     target = URIRef("http://dbpedia.org/resource/" + topic_of_interest)
182     q = prepareQuery(
183         """SELECT ?student WHERE {
184             ?sub_cata foaf:topic ?tar .
185             ?record focu:subject_catalog ?sub_cata .
186             ?record focu:grade ?grade .
187             FILTER(?grade != "F") .
188             ?student focu:hasRecord ?record
189         }""",
190         initNs = {
191             "tar": target, "dbp": "http://dbpedia.org/resource/" ,
192             "foaf": FOAF, "rdf":RDF, "focu":"http://focu.io/schema#" }
193     )
194     print("Hal_9001 > The following Students are familiar with " + question_details.strip("?") + ":")
195     for row in g.query(q, initBindings={"tar": target}):
196         pretty_row = row[0].replace("http://example.org/", "")
197         print("    ", pretty_row)

```

Figure 19: *query_knowledge_graph* function, Question 4

```
User > Who is familiar with Artificial intelligence?  
Hal_9001 > The following Students are familiar with Artificial intelligence:  
          99  
          100  
          74  
          88  
          64  
          54  
User >
```

Figure 20: Question 4 output

APPENDIX:

Assumptions:

- The level certainty when obtaining Topics from DBSpotlight was not modified from Phase I to Phase II. Therefore our Chatbot will still return things such as “Coen River” being a topic of any COEN (Computer Engineering) course.
 - Concerning the 5th element of the Phase I deliverables;
 - 5. *A link (rdfs:seeAlso) to a web page with the course information, if available*
- Our data was sourced using the Concordia *opendata.concordia.ca/API* which does not provide an additional link for courses, therefore this task is not covered in our report.