جامعة الطائف
المنظومة الجامعية

LOGO.ADAM96.COM

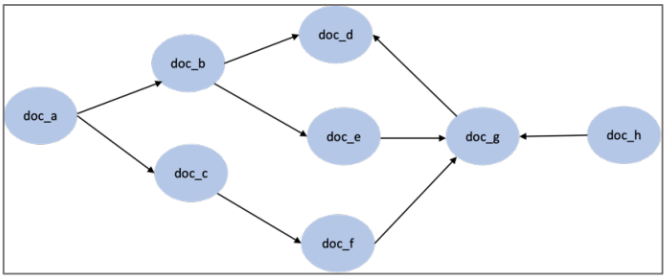
Assignment 2

Artificial Intelligence 501481-3

Knowledge Representation and Reasoning with Prolog

Task1: Citation Graph [5 marks]

Consider the following directed acyclic graph that represents citations between publications. In information science and bibliometrics, a citation graph is a directed graph that describes the citations within a collection of documents. Each vertex (or node) in the graph represents a document in the collection, and each edge (arc) is directed from one document toward another that it cites. For instance, in the given citation graph below, document b cites documents d and e, and document b is cited by document a.



1) Considering nodes (document names) as constants, represent the facts in the given citation graph in Prolog.

```
node(doc_a, doc_c).
node(doc_a, doc_b).
node(doc_c, doc_f).
node(doc_b, doc_e).
node(doc_b, doc_d).
node(doc_e, doc_g).
node(doc_f, doc_g).
node(doc_g, doc_d).
node(doc_h, doc_g).
```

Based on the given Citation Graph, the relationships between documents have been represented using facts in Prolog.

In this model:

Each document is represented as a **node**.

2) Represent the following scenario in your Prolog program:
   a. Two documents are connected if there is a directed path from the first document to the second document in the directed acyclic graph. Is there any specific property that you need to use to define this sentence in your program? What is it? Explain it briefly.

```
connected(A, B) :- node(A, B).
connected(A, B) :- node(A, C), connected(C, B).
```

Recursion is the process of a function or rule calling itself, either directly or indirectly. It's a powerful tool often used to solve problems that can be broken down into smaller, similar sub-problems. In logic programming languages like Prolog, recursion is a key concept, especially for handling tasks that involve hierarchical or interconnected data, such as graphs, family relationships, or hierarchical databases

Example of Recursion in Graphs:
To check if there's a path between two nodes A and B, we use the connected(A, B) relation.

Direct Path: If A is directly connected to B.

This checks if there's a direct connection. If so, it returns true.

$$connected(A, B) :- node(A, B).$$

Indirect Path: If A is connected to an intermediate node C, and C is connected to B.

This rule keeps checking for connections through intermediate nodes (C) until it finds a direct link or exhausts all options.

$$connected(A, B) :- node(A, C), connected(C, B).$$

Works:
If a direct connection is found, recursion stops.

Otherwise, it keeps trying through intermediate nodes

b. Query the SWI-Prolog to show whether document a and document g are connected.

```
% c:/users/rana/onedrive/
?- connected(doc_a,doc_g).
true
```

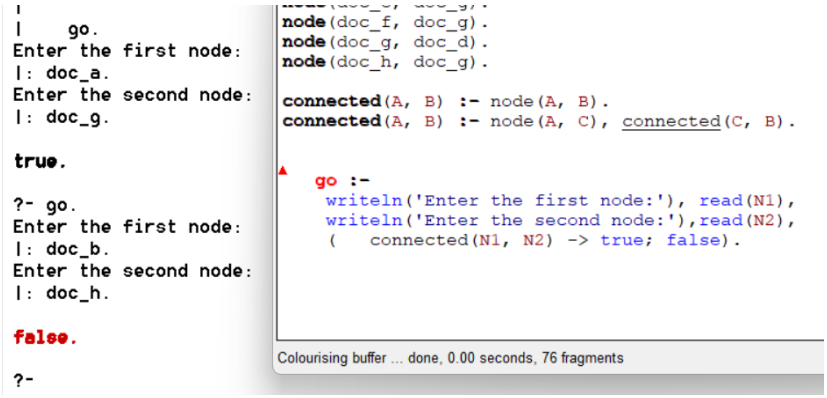
```
connected(A, B) :- node(A, B).
connected(A, B) :- node(A, C), connected(C, B).
```

c. Query the SWI-Prolog to show whether document b and document h are connected.

```
|     connected(doc_b,doc_h).
false.

?-
```

```
connected(A, B) :- node(A, B).
connected(A, B) :- node(A, C), connected(C, B).
```

d. Receive the first and the second documents to check the existence of a path between them **from the user's keyboard**. The output is as follows:



- The line $writeln('Enter\ the\ first\ node:'), read(N1)$ displays a message on the screen asking the user to input the first node using the $writeln$ command.
After that, the read command reads the input entered by the user and stores it in the variable N1.

- The next line does the same thing but asks the user to input the second node, and the entered value gets stored in N2.

- The line $(connected(N1, N2) -> \ true;\ false)$ checks if there's a connection between the two nodes N1 and N2:

If there is a connection (meaning the $connected(N1, N2)$ condition is true), it returns true.

If there's no connection, it returns false.

- The $->$ symbol here means "if the condition is met," and it's used as part of the conditional check to determine the correct outcome based on whether the nodes are connected It's similar to the if conditional statement you learned in C++.

e. Using forward chain reasoning, properly explain the answer of (b) or (c).

(c)

1. Direct Connection

$$connected(A, B) :- node(A, B).$$

This rule means that two nodes A and B are directly connected if a fact $node(A, B)$ exists in the database.

To check if doc_b and doc_h are directly connected, we look for the fact:

$$node(doc\_b, doc\_h).$$

Based on the provided facts, this relationship does not exist.

Result: There is no direct connection between doc_b and doc_h.

2. Indirect Connection

$$connected(A, B) :- node(A, C), connected(C, B).$$

This rule means A and B are indirectly connected if. There's an intermediate node C such that, A is directly connected to C ($node(A, C)$,C is connected to B ($connected(C, B)$).

We look for all nodes C such that:

$\_\ node(doc\_b, C)$.

Based on the facts, the nodes directly connected to doc_b are:

1-$node(doc\_b, doc\_d)$.

2-$node(doc\_b, doc\_e)$.

Check connection between C and doc_h:

$\_connected(doc\_d, doc\_h)$.

There's no $node(doc\_d, doc\_h)$ in the facts.

There's no indirect connection either (no intermediate path from $doc\_d\ to\ doc\_h$).

Result: False.

(b)

1-Direct Connection

$$connected(A, B) :- node(A, B).$$

This rule states that two nodes A and B are directly connected if a fact $node(A, B)$ exists in the database.

To check if doc_a and doc_g are directly connected, we look for the fact:

$$node(doc\_a, doc\_g).$$

Based on the provided facts, this relationship **does not exist**.

Result: doc_a and doc_g are **not directly connected**.

2-Indirect Connection

$$connected(A, B) :- node(A, C), connected(C, B).$$

This rule states that A and B are indirectly connected if there is an intermediate node C such that:

A is directly connected to C ($node(A, C)$).

C is connected to B ($connected(C, B)$).

We look for all nodes C such that:

$\_\, node(doc\_a, C)$.

From the facts, the nodes directly connected to doc_a are:

$node(doc\_a, doc\_b)$
$node(doc\_a, doc\_c)$

Verify connection between C and doc_g:

$connected(doc\_b, doc\_g)$.

$node(doc\_b, doc\_g)$ exists in the facts.

Result: doc_b is **connected**(**true**) to doc_g.

Since doc_a is connected to doc_b, and doc_b is connected to doc_g, there is an indirect connection between doc_a and doc_g.

3) Find a list of documents that are cited by document a (using the **findall** built-in predicate).

```
|
|     find_list(doc_a,List
List = [doc_c, doc_b].

?- find_list(doc_b,List).
List = [doc_e, doc_d].

?-
```

```
find_list(Doc, List) :- findall(Target, node(Doc, Target), List).
```

Doc: The input item for which we want to retrieve all related values.

List: The output collection that will contain all the retrieved values.

Purpose of the node(Doc, Target) relation: This defines how values are connected to Doc, allowing us to gather all items linked to it.

The findall command: It is used to gather all possible values (Target) associated with Doc through the node relation.

Finally, all the gathered values are stored in List as the result.

4) If two documents are connected, we need the program to show the path from the first to the last node as a list. For example, the **path from document a to document g are [doc_a, doc_b, doc_e, doc_g]** and **[doc_a, doc_c, doc_f doc_g]**. If the two nodes are not connected, we should get false as an answer from SWI-Prolog. One way to represent this is by introducing a new three-arguments predicate where the first will match the initial node, the second will match the final node, and the third will match a list consisting of the nodes in a path. Query to SWI-Prolog to show the path between different pairs of nodes as shown in the figure below. **Note**: you need to use the connected predicate that you defined in 2.a to define the path rule.

```
|     path(doc_a,doc_g,X).
X = [doc_a, doc_b, doc_e, doc_g] ;
X = [doc_a, doc_c, doc_f, doc_g] ;
false.

?-  path(doc_a,doc_b,X).
X = [doc_a, doc_b] ;
false.

?-  path(doc_a,doc_h,X).
false.

?-  path(doc_b,doc_g,X).
X = [doc_b, doc_e, doc_g] ;
false.

?-
```

```
.pl [modified]
File  Edit  Browse  Compile  Prolog  Pce  Help
.pl [modified]

path(Start, Goal,[Start, Goal]) :- node(Start, Goal).
path(Start, Goal, Path) :-  node(Start, Node2), path(Node2, Goal, Path2),append([Start], Path2, Path).
```

Rule: $path(Start, Goal, [Start, Goal]) :- node(Start, Goal).$

Purpose: Find a direct path between Start and Goal.

Result: A $list$ $[Start, Goal]$ that shows the direct connection.

Brackets [ ]: Used to store the direct path.

Rule: $path(Start, Goal, Path) :- node(Start, Node2), path(Node2, Goal, Path2), append([Start], Path2, Path).$

Purpose: Find an indirect path between Start and Goal by going through intermediate nodes.

$node(Start, Node2)$: Find an intermediate node $Node2$ directly connected to $Start$.

$path(Node2, Goal, Path2)$: Call the same rule recursively to find a path from $Node2$ to $Goal$.

$append([Start], Path2, Path)$: Add Start at the beginning of the resulting path $Path2$ to form the full $path$

$append$ is used to merge two lists into one list.

As shown in the graph, a solid directed arrow from CS15 to CS16 indicates that CS15 is a prerequisite to CS16. A dotted arrow means that ONE of the these courses is a prerequisite of the course. For example, to take CS33, you have to take *either* CS16 *or* CS18. Course names are constants, so you have to use lowercase letters (cs22,cs33, etc) in your code. More facts about the transcript are:

- CS15, CS17, CS33, CS141 and CS126 are only offered in the fall.
- CS16, CS18, CS22, CS32, CS166 are only offered in the spring.

prerequisite :

These define the courses that a student must complete before they can take a specific course.

optionP :

These define alternative courses that a student can take as a requirement for another course.

offered:

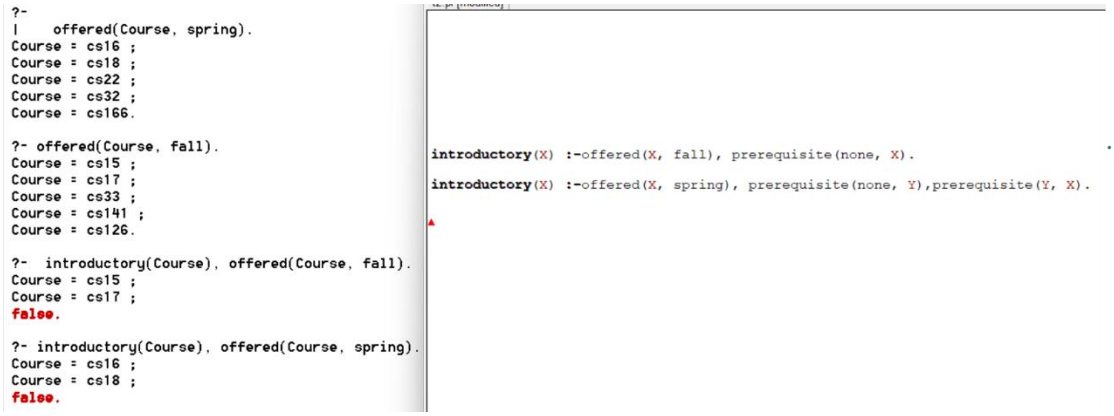
These indicate the semesters in which courses are offered, whether in the fall or spring.

introductory:

These define introductory courses that do not depend on any other courses.

```
prerequisite(cs33, cs166).
prerequisite(cs32, cs126).
prerequisite(cs15, cs16).
prerequisite(cs17, cs18).
prerequisite(cs22, cs126).
prerequisite(cs22, cs141).
prerequisite(cs16_or_cs18, cs32).
prerequisite(cs16_or_cs18, cs33).

optionP(cs16, cs32).
optionP(cs16, cs33).
optionP(cs16, cs141).
optionP(cs18, cs32).
optionP(cs18, cs33).
optionP(cs18, cs141).

offered(cs15, fall).
offered(cs17, fall).
offered(cs33, fall).
offered(cs141, fall).


offered(cs16, spring).
offered(cs18, spring).
offered(cs22, spring).
offered(cs32, spring).
offered(cs166, spring).


introductory(cs15).
introductory(cs17).
introductory(cs22).
```

- A course is **introductory** if it's offered in the fall and has no prerequisites, or if it's offered in the spring and its prerequisite that has no prerequisites.



```
?-
|    offered(Course, spring).
Course = cs16 ;
Course = cs18 ;
Course = cs22 ;
Course = cs32 ;
Course = cs166.

?- offered(Course, fall).
Course = cs15 ;
Course = cs17 ;
Course = cs33 ;
Course = cs141 ;
Course = cs126.

?-  introductory(Course), offered(Course, fall).
Course = cs15 ;
Course = cs17 ;
false.

?- introductory(Course), offered(Course, spring).
Course = cs16 ;
Course = cs18 ;
false.
```

```
introductory(X) :-offered(X, fall), prerequisite(none, X).
introductory(X) :-offered(X, spring), prerequisite(none, Y),prerequisite(Y, X).
```

$$introductory(X) :- offered(X, fall), prerequisite(none, X).$$

Purpose: To identify introductory courses that are offered in the fall.
Condition:

The course X is offered in the fall ($offered(X, fall)$).

It has no prerequisites ($prerequisite(none, X)$).
Result: The course X is considered introductory if these conditions are met.

$$introductory(X) :- offered(X, spring), prerequisite(none, Y), prerequisite(Y, X).$$

Purpose: To identify introductory courses that are offered in the spring.
Condition:

The course X is offered in the spring ($offered(X, spring)$).

It has a prerequisite Y ($prerequisite(Y, X)$).

The prerequisite Y has no prerequisites ($prerequisite(none, Y)$).
Result: The course X is considered introductory if it has only one prerequisite, which is itself introductory.

- A course is **intermediate** if it is not introductory, but its prerequisites are all introductory courses.

```
?- intermediate(X).
X = cs16 ;
X = cs141 ;
X = cs126 ;
X = cs18 ;
false.

?-
```

t2.pl [modified]

```
intermediate(X) :- prerequisite(Y, X), introductory(Y).
```

This rule is used to identify intermediate courses. Any course that depends on an introductory course is considered an intermediate course. The idea is that an intermediate course must have an introductory course as a prerequisite.

- A course is **upper level** if its prerequisites are not introductory courses, or if its prerequisites are CS22 **and** introductions.

```
?-
% c:/users/ran
% c:/users/ran
?- upper_level
X = cs166 ;
X = cs33 ;
X = cs32 ;
X = cs126 ;
false.

?-
```

```
upper_level(X) :-
    prerequisite(Y, X),
    not(introductory(Y)),
    not(intermediate(X)).


upper_level(X) :-
    X = cs126,
    prerequisite(cs22, X),
    prerequisite(Z, X),
    introductory(Z).
```

The first rule defines advanced courses that do not depend on introductory or intermediate courses. However, when testing the code using only this rule, all advanced courses appeared except for **cs126**, which should be considered an advanced course. Therefore, the second rule was added to define **cs126** as an advanced course, provided it depends on **cs22** and **Z**, where **Z** is an introductory course.

If **cs126** is not considered an advanced course because it depends on **cs22**, then the first rule is correct and contains no issues, as it accurately defines advanced courses that do not depend on introductory courses.

- Course may only be of one level; i.e., introductory courses may not also be intermediate courses, etc.

```
|
|
|    courseLevel(Course, Level).
Course = cs166,
Level = upper_level ;
Course = cs33,
Level = upper_level ;
Course = cs32,
Level = upper_level ;
Course = cs16,
Level = intermediate ;
Course = cs141,
Level = intermediate ;
Course = cs18,
Level = intermediate ;
Course = cs15,
Level = introductory ;
Course = cs17,
Level = introductory ;
Course = cs22,
Level = introductory ;
false.

?-
```

```
courseLevel(X, upper_level) :- upper_level(X),
  not( intermediate(X)),
    not( introductory(X)).
courseLevel(X, intermediate) :- intermediate(X),
  not(upper_level(X)),
    not( introductory(X)).
courseLevel(X, introductory) :- introductory(X),
  not( upper_level(X)),
    not(intermediate(X)).
```

These rules are used to ensure that a course can only belong to one level (introductory, intermediate, or upper-level). It prevents any course from being classified under more than one level.

You can now add a set of facts that students have taken some courses using the `has_taken` predicate (i.e., `has_taken`(sarah, cs15)). Then, you need to add one more rule to test the eligibility of a student to take a course:

- `can_take`(Student, Course), true when Student is eligible to take Course (i.e., they have taken the necessary prerequisites).

```
?- can_take(sarah, Course).
Course = cs16 ;
Course = cs126 ;
Course = cs141 ;
Course = basic_course ;
Course = advanced_course ;
false.

?- can_take(sarah, cs15).
false.

?- can_take(sarah, cs16).
true .

?-  can_take(Student, cs16).
Student = sarah
```

```
has_taken(sarah, cs15).
has_taken(sarah, cs22).
has_taken(rana, cs16).
has_taken(rana, cs17).

can_take(Student, Course) :-prerequisite(Prerequisite, Course), has_taken(Student, Prerequisite).
can_take(Student, Course) :-optionP(Prerequisite, Course),has_taken(Student, Prerequisite).
can_take(none, Course) :- prerequisite(none, Course).
```

The code is used to determine if a student is eligible to take a specific course based on the courses they have completed. **can_taken 1**is used to track the courses the students have completed, and **can_take 2** checks if the student has finished the necessary prerequisites or alternative courses to be eligible to take a particular course.

can_take 3 This rule is used to check for courses that don't have any prerequisites (courses that students can take directly without needing to complete other courses first). The term **none** means that there are no conditions or other courses that need to be completed before the student can take the course.

Now that you have implemented the knowledge base (facts and rules), you can test your implementation by querying the knowledge base:

- Add a fact that a student (e.g., sarah) has only taken one intermediate course and the prerequisites for this intermediate course. Then test whether sarah can take any other courses from the curriculum.

```
% c:/users/rana/onedrive/desktop/t2 compiled 0.00 sec, 0 clauses
?- has_taken(sarah, intermediate_course).
true.

?- can_take(sarah, advanced_course).
true .

?-  can_take(sarah, Course).
Course = cs16 ;
Course = cs126 ;
Course = cs141 ;
Course = basic_course ;
Course = advanced_course ;
false.

?-
```

t2.pl [modified]

File Edit Browse Compile Prolog Pce Help

t2.pl [modified]

```
has_taken(sarah, cs15).
has_taken(sarah, cs22).
has_taken(rana, cs16).
has_taken(rana, cs17).

 has_taken(sarah, basic_course1).
has_taken(sarah, intermediate_course).

prerequisite(intermediate_course, basic_course).
prerequisite(intermediate_course, advanced_course).

can_take(Student, Course) :-prerequisite(Prerequisite, Course), has_taken(Student, Prerequisite).
can_take(none, Course) :- prerequisite(none, Course).
```

Facts Addition: The code includes facts that specify the courses each student has completed. These facts are used to check if a student is eligible to take a course based on the prerequisites they have already completed.

- The only course that manal can take is cs32 (thus insert the required facts to make the answer to this query true).

```
?- has_taken(manal, cs32).
true.

?-
```

```
has_taken(manal, cs32).

can_take(manal, Course) :-
    Course \= cs32, false.
```

The code defines that Manal can only take **cs32**. If we try to check if Manal can take any other course besides **cs32**, the result will be **false**. However, if the course is **cs32**, the result will be **true**.

\= is a comparison operator that ensures the value of **Course** is not equal to **cs32**. If **Course** equals **cs32**, the query will return **true**.

- Retrieve the introduction, intermediate and upper-level courses in the curriculum.

```
|
|     introductory(Course).
Course = cs15 ;
Course = cs17 ;
Course = cs22 ;
false.

?-  intermediate(Course).
Course = cs16 ;
Course = cs18 ;
Course = cs126 ;
Course = cs141 ;
false.

?- upper_level(Course).
Course = cs166 ;
Course = cs32 ;
Course = cs33 ;
Course = basic_course ;
Course = advanced_course ;
Course = cs126 ;
false.
```

```
introductory(cs15).
introductory(cs17).
introductory(cs22).

introductory(X) :-offered(X, fall), prerequisite(none, X).

introductory(X) :-offered(X, spring), prerequisite(none, Y),prerequisite(Y, X).


intermediate(X) :- prerequisite(Y, X), introductory(Y).

upper_level(X) :-
    prerequisite(Y, X),
    not(introductory(Y)),
    not(intermediate(X)).

upper_level(X) :-
    X = cs126,
    prerequisite(cs22, X),
    prerequisite(Z, X),
    introductory(Z).
```

I have referred to a variety of resources to understand how to solve the codes. These include watching several educational videos on YouTube, exploring relevant books, and consulting the primary reference material, which is the chapters of the course.

Logic Programming with Prolog

Lecture Notes An Introduction to Prolog Programming

youtube