CPCS-331 Artificial Intelligence (I)

Exploring Semantic Nets for Knowledge Representation

Report

Prepared by Group AE:

Name	ID
Ahmad Ajedaani	2136071
Mahmued Alardawi	2135209
Abdullah Emad Almashharawi	2136141
Abdullah Abed Alharbi	2135999

Introduction

Semantic networks are another form of knowledge representation, where knowledge is represented as a graph. The nodes of the graph represent objects in the world while the arrows represent the relationship between nodes.

In this report, we will be implementing them in practice, with a premade example, and testing it by applying operations and queries on the semantic net.

Choice of Domain

The semantic network that we've implemented can work on any domain. However, for the sake of simplicity, we've chosen a premade example which is the university system and its relations, due to how complex the system is so we can demonstrate the capability of semantic networks.

Design of the Semantic Net

The design will be using a graphing module for creating the semantic net, representing nodes using vertices and arrows using edges. To display the semantic net, we'll be using a plotting module to help with that task.

There will also be queries to look for a specific node or its relations, and a query to get the links from one specific node to another. Not to mention operations for manipulating the graph such as deleting or adding nodes/relations.

Implementation

We'll be implementing the problem and the semantic net using Python, due to having a plethora of libraries fit for this task. We will also be using two main modules, Networkx, and Matplotlib; both of which we'll go on an in-depth explanation later.

As mentioned earlier, we've decided to put a premade graph inside the code rather than the code scanning from, for example, a text file/console. The same goes for the queries, graph editing, plotting, etc.

Some of the operations will already be available in the imported modules, while others will be custom made for this program.

Basic Operations & Modules

- Modules:

1. Networkx: This Python library is instrumental for creating and visualizing complex graphs and networks. It provides a suite of powerful functions that facilitate the analysis of relationships between nodes within our case study. Noteworthy functions include:

   'add_edge': Allows for the insertion of edges between nodes, accepting a 'relationship' parameter to define the connection.
   'remove_nodes_from': Removes a specified list of nodes from the graph.
   'remove_edge_from': Eliminates an edge when provided with a list of node tuples.
   'spring_layout': Computes node positions in a 2D space, outputting a dictionary with x and y coordinates for each node.
   'draw': Executes the graphical representation of the network, requiring the graph and position dictionary, with an optional 'with_labels' boolean parameter to display node labels.
   'get_edge_attribute': Retrieves specific attributes associated with an edge.
   'draw_network_edge_label': Renders edge labels that denote relationships.
   'Graph': Constructs a new networkx Graph object instance.

2. Matplotlib: is a famous library for plots. In our use case, we only used 'show' method to show the graphs drawn by networkx's methods

- Basic Operations:

1. *draw_graph*: visualizes a graph object using the networkx and matplotlib libraries. It computes the positions of the nodes using a spring layout, draws the graph with node labels, retrieves and adds edge labels based on a 'relationship' attribute, and finally displays the visualization on screen.
2. *draw_graph_with_target*: visualize a graph, highlighting a specific target node with the red color.
3. *check_node*: checks if a given node exists in a given graph or not.
4. *get_edges_of_node*: print out the edges connected to a specific node in a graph, along with the 'relationship' attribute associated with each edge.
5. *Create_subgraph_from_path*: checks if the given pattern (path) existed in the graph or not, if yes, then it draws out a sub graph with the given path.

Testing and Results

This section will show a demonstration of a subset of the network with input & output examples with code snippets. For simplicity, we will only look at the relation between University and Faculty along with their relations.

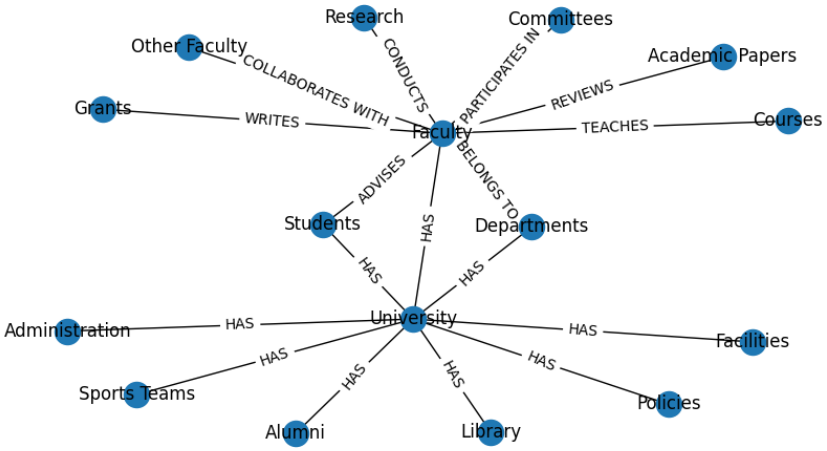
Adding nodes:

```
# Adding Entities (Nodes):
G.add_nodes_from([
    "University", "Departments", "Faculty", "Students", "Administration", "Library",
    "Sports Teams", "Alumni", "Facilities", "Policies", "Courses", "Research",
    "Committees", "Grants", "Academic Papers"
])
```

Adding edges for both:

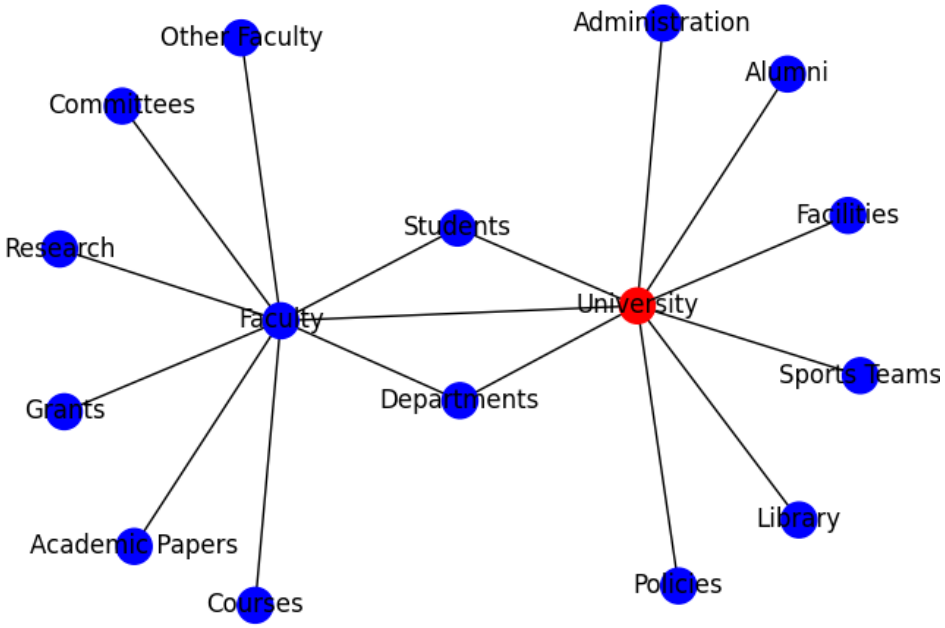
```
# Adding Relationships (Edges):
G.add_edge("University", "Departments", relationship="HAS")
G.add_edge("University", "Faculty", relationship="HAS")
G.add_edge("University", "Students", relationship="HAS")
G.add_edge("University", "Administration", relationship="HAS")
G.add_edge("University", "Library", relationship="HAS")
G.add_edge("University", "Sports Teams", relationship="HAS")
G.add_edge("University", "Alumni", relationship="HAS")
G.add_edge("University", "Facilities", relationship="HAS")
G.add_edge("University", "Policies", relationship="HAS")

G.add_edge("Faculty", "Courses", relationship="TEACHES")
G.add_edge("Faculty", "Research", relationship="CONDUCTS")
G.add_edge("Faculty", "Students", relationship="ADVISES")
G.add_edge("Faculty", "Committees", relationship="PARTICIPATES IN")
G.add_edge("Faculty", "Other Faculty", relationship="COLLABORATES WITH")
G.add_edge("Faculty", "Grants", relationship="WRITES")
G.add_edge("Faculty", "Academic Papers", relationship="REVIEWS")
G.add_edge("Faculty", "Departments", relationship="BELONGS TO")
```

The result after calling the draw_graph() method would be:



- Operations on the graph:

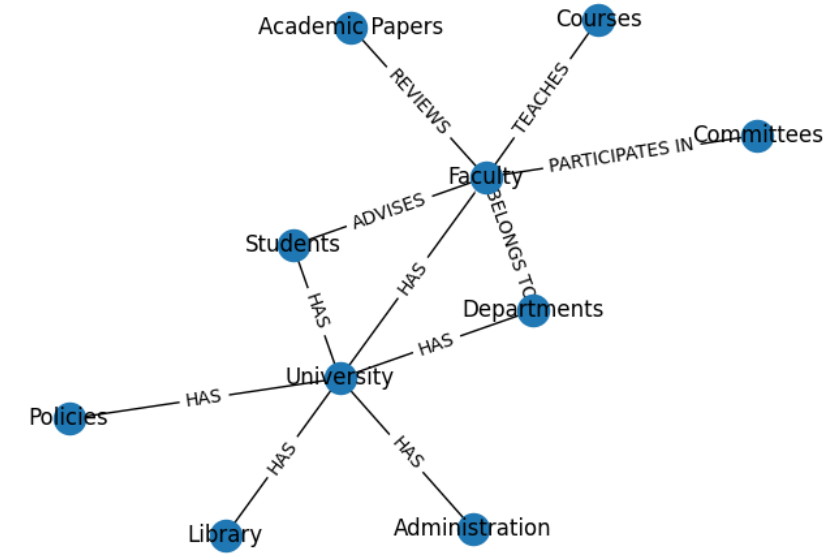
- Draw graph with targeted node, by setting the target to ‘University’ we get the following graph:



- Remove the following nodes

```
# Removing nodes
G.remove_nodes_from(["Grants", "Other Faculty", "Research", "Adminstration", "Sports Teams", "Alumni", "Facilities",])
draw_graph(G)
```

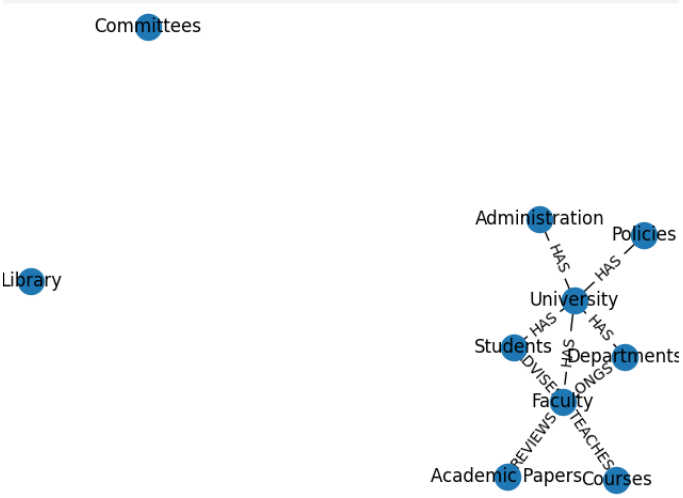
Output:



- removing edges:

```
# Removing edges
G.remove_edges_from([("Faculty", "Committees"), ("University", "Library")])
draw_graph(G)
```

Output:

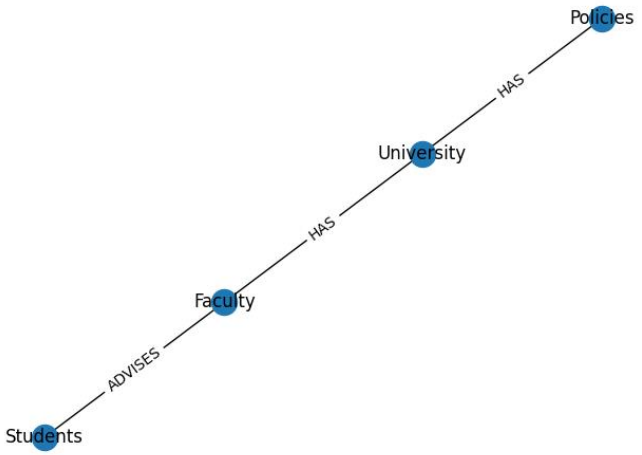


- Finding If a relation exists and draw it

Example:

```
# Finding a specific pattern
pattern = ["Policies", "University", "Faculty", "Students"]
g1 = create_subgraph_from_path(G, pattern)
draw_graph(g1)
```

Output:



- Some queries:

- Checking if a node exists

```
# Check if node exists
check_node(G, "University")
print()

check_node(G, "Kharbosh")
print()
```

Output:

```
The node 'University' is in the graph.

The node 'Kharbosh' is not in the graph.
```

- Get edges of a node:

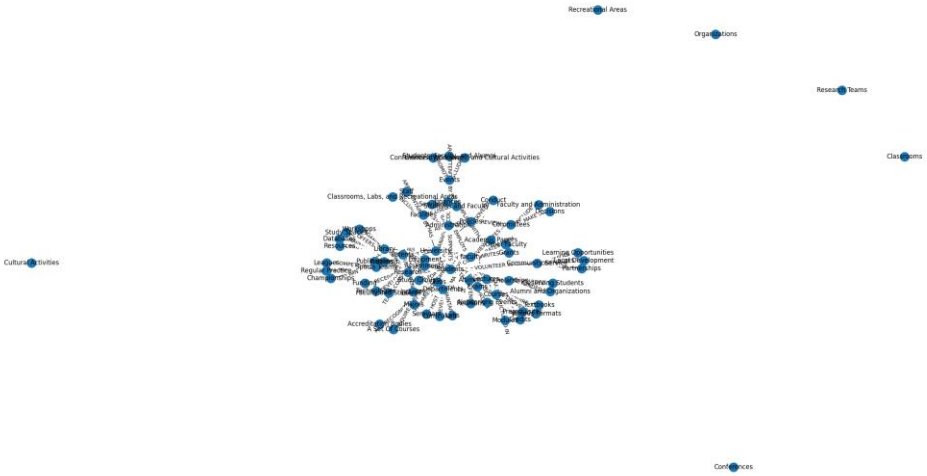
```
# Finding edges of node
get_edges_of_node(G, "University")
print()
```

Output:

```
Edge: University, Edge: Departments, Relationship: HAS
Edge: University, Edge: Faculty, Relationship: HAS
Edge: University, Edge: Students, Relationship: HAS
Edge: University, Edge: Administration, Relationship: HAS
Edge: University, Edge: Policies, Relationship: HAS
```

Conclusion

In conclusion, after adding all nodes and edges we will get this very complex network:



Please run the code to see better insights into the graph details and provided images to understand the relation better.