Keith Allen

Capstone Paper

Fall 2023

1 Background

The past few decades have seen an explosion in Ontology research, much of which now is concerned with the collaboration between the field of Ontology and Computer Science. With any research there is a need to measure results to understand if changes are bringing the desired progress, and in a field that routinely focuses on improving the handling of large data sets, the results themselves can become large and unruly, leading to the need to employ special computational methods to measure results. This paper will lay out a background about Ontology and ontologies, provide the motivation for why this project needs to repurpose a specific similarity metric, and then offer a possible solution and implementation of the similarity metric proposed.

1.1 What is an Ontology?

The story of Ontology is the story of sorting, cataloging, and classifying the world around us. It is a subfield of Metaphysics, which in turn is a branch of Philosophy. In its most general sense, Ontology is concerned with creating a taxonomy of entities and the relations between the entities, referred to as ontologies. These classifications and relations are universal and aim to represent reality in a consistent repository[1][2]. This is a monumental task, and in the process of tackling it many types of ontologies have been developed. Two are of concern to this project.

The first are top level ontologies (TLO). These ontologies classify terms in high level, general categories. For example, there are the Basic Formal Ontology (BFO) superclasses

[1]This is at least the perspective conveyed by Barry Allen in [2].

of **object** and **process**[2] [2]. Because these classifications are so general, and there are so few native BFO terms, that at first glance terms assigned to a superclass may not appear to have much in common get grouped together. For example, in the domain ontology Food Ontology (FoodOn), *Liquid* and the *consumer-ready food packaging* both belong in the BFO **material entity** superclass. To get into the specifics, domain ontologies are needed.

Domain ontologies are more granular, and as their name implies, are built to classify terms in a specific domain. Some examples of domain ontologies include the Ontology of Electronics (OOE) and the Epilepsy Ontology (EPIO). Instead of large overarching classifications, domain classes can become incredibly specific, such as the Condiment class within the (FoodOn) [6]. When using best practices to create ontologies, all domain terms are children of TLO terms, and together they can create highly consistent and detailed ontologies.

1.2 Importance of Ontologies

Now that the goal of Ontology, and the types of ontologies, have been outlined an important question arises. Why put in the time and effort to study and create ontologies? The answer to this question lies within data analysis and the ever-increasing utilization of large data sets. The use of ontologies as a whole allows for consistent semantic labeling of large data sets. By labeling the elements of these massive data sets, they become easier to compute over, and better insights can be drawn from them [2]. Not only can more be learned from consistently labeled data, but this practice also allows data to be reused and shared both within, and outside, of the original organization or domain. These benefits drawn from using ontologies have been noticed and in response, ontologies are being developed both by companies and the government in an attempt to help standardize ever-expanding information systems.

However, the pairing of top level and domain ontologies was not always the norm. The

[2]Throughout this paper, terms assigned to ontologies are presented using *italics*, and ontological classes using **bold text**.

modern field of Applied Ontology traces its origins within Bioinformatics [2]. Decades ago experts in different domains of Bioinformatics started building ontologies, but when trying to share their ontologies they ran into the issue of *siloing*. All of their respective ontologies were well structured, and thoughtfully built, but there was nothing connecting them. Instead each ontology existed on its own, separated into silos by this lack of centralized agreement. On their own, each ontology was an outstanding resource to its organizations, but they lost out on the benefits of sharing information and ontology terms amongst themselves. In an effort to combat this, TLOs were introduced. While there are multiple common TLOs, this project focuses on (BFO).

BFO is the first TLO recognized by the International Organization for Standardization (ISO). It was recognized as ISO/IEC 21838-2:2021 in November of 2021 [1], but even before becoming an ISO standard BFO had been widely adopted throughout the world, heavily so within Bioinformatics. This is, of course, because of its close work and development alongside Bioinformatics since the early 2000's. According to the official BFO website, BFO is currently utilized by over 400 ontologies by over 100 organizations [6]. Its widespread implementation makes it a great candidate to explore the possibilities of assisting users with automation. By using a TLO that is already standardized the products of this research can be utilized by an established community.

1.3 Project Motivation

It has been established that ontologies can offer great improvements when computing over large data sets and the benefits are increased by tying together domain ontologies via TLOs. So why are there not domain ontologies for every domain and all of their relevant terms already? The reality of creating domain ontologies is that it is an arduous process that takes a great length of time, along with close collaboration between domain experts and ontologists. In addition to the amount of labor required, the process is also prone to human error and if not created correctly, an ontology might have circular relations between

terms. Sets of two terms that form these circular relations can be easy to spot with the human eye, but cycles containing three terms or more can be hard to spot [8]. Thus, in an attempt to alleviate these burdens the parent project of this smaller project has worked to create a system that a domain expert can use to efficiently build accurate domain ontologies.

The Dialog Based Ontology Learner (DBOL) uses current trends in Artificial Intelligence, such as dialog systems and analogical reasoning, to do just that. Over the past two years the DBOL team has worked to create and implement a set of questions, known as the beginner rules, that are intended to assist the user in classifying a term to its correct BFO superclass[3]. Before the system can assist the user in creating domain ontologies, it must first be able to correctly classify terms within the BFO ontology. The end goal is an automated system that a domain expert can use to create BFO-compliant domain ontologies. After initial creation of the beginner rules the system was opened to testers. Using the results of this initial test edits were made and a second round of user testing was administered. The testing consisted of a short training on how to use the system, followed by having the user classify five terms using the DBOL, and while attempting to analyze the results, a new question arose. What does it mean to classify a term incorrectly? There is the simple "correct" or "incorrect", but there is more meaning to be found than just "yes" or "no". To better understand, consider the following example.

As previously stated, the term *laptop* with the definition "a portable computer small enough to use in your lap" has a BFO superclass of **object**. A visible representation of this correct classification can be found in figure 1. On the contrary, figure 2 contains two incorrect classifications. The left side of figure 2 assigns *laptop* as an **object aggregate**, while the right assigns it as an **immaterial entity**. The classification of **object aggregate**, while wrong, is at least more correct than **immaterial entity**.[4]

[3]A superclass is a parent classification. For example, a *laptop* has a superclass of **object**.

[4]All relations in figures 1 and 2 are the BFO "is_a" relation. For example, *laptop* "is_a" **object** and in turn **object** "is_a" **material entity**.

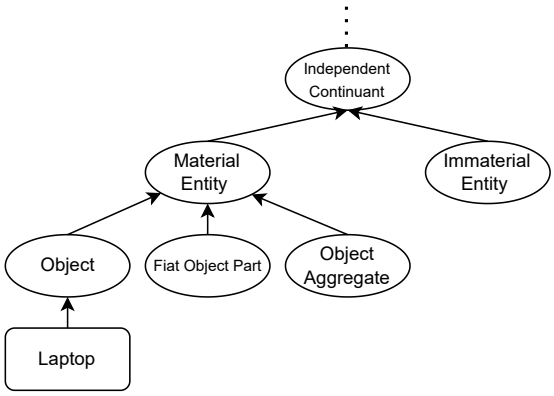

Figure 1: Correct assignment of *laptop* as an **object**.

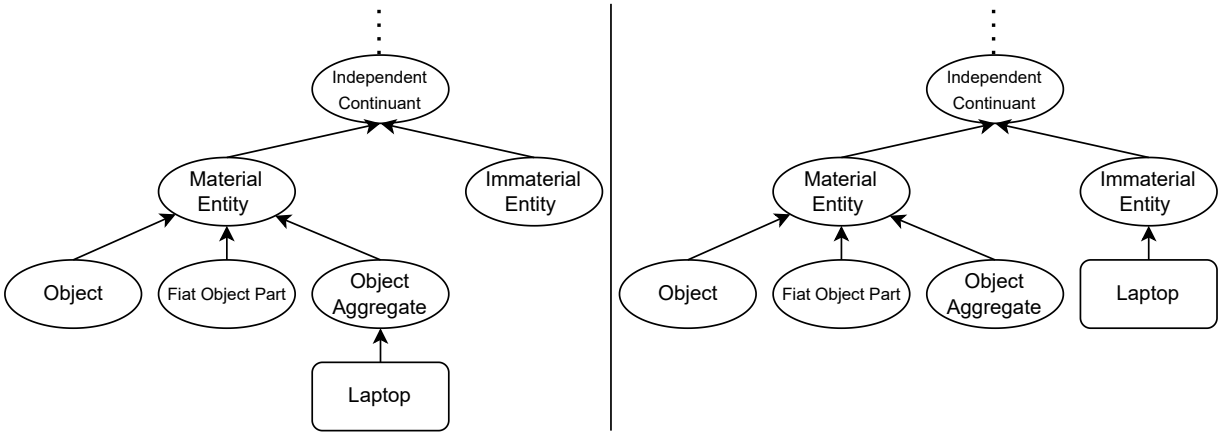

Figure 2: Incorrect assignment of *laptop* as an **object aggregate** (left) and an **immaterial entity** (right).

This is because in the first case the user, with the help of the system, was able to navigate to the portion of the DBOL concerned with material entities. Therefore even when the term is classified incorrectly some of its core attributes may still be correct. If a laptop is being considered an **object aggregate** then there was confusion as to whether a *laptop* is a standalone entity, or if it is somehow in a group, or a group itself of smaller components. However, an **immaterial entity**, as its name implies, is not made of matter. If the DBOL is unable to find out whether the term is physically made of matter, this points to a different problem than just whether a term is a standalone entity. To make these judgements consistently a standard measurement is required but to measure, first the problem must be formalized and abstracted.

1.4 Background Summary

In this section the field of Ontology, some examples of ontologies, and their uses have been defined and explained. Additionally, the goal of automating ontology creation has been established and the current goal of the parent project has been shared. Finally, the concern with measuring within ontologies has been posed and will motivate the remainder of this paper.

2 From Ontology to DAGs

2.1 Defining the Problem

Now that the motivation of this topic study has been established and the key question has been asked, now the problem can be abstracted. To begin, the constraints of the real world use case need to be explored. From the figures used in the example in the previous section some patterns can be found. The first key takeaway is that the terms used in each figure remain the same. Not only that, but all of the relations between terms are the BFO "is_a" relation, used to assign superclasses and organize terms in pairs of parent and child. Looking at the figure it is not a stretch to pick up on the graph structure that the nodes and their relations seem to create.

These three features of the BFO hierarchy can then be mapped to features of a graph. The sets of terms being identical can instead be understood as the vertices of two graphs being the same. Since all the relations are the same, this corresponds to all weights of a graph's edges being the same. Finally, since the ontologies differ in what nodes are connected to each other, the graphs would have two different sets of edges. With these three aspects of the abstracted graphs, it can be seen that the BFO hierarchy and any terms assigned to their BFO superclass form a directed acyclic graph, also known as a DAG.

2.2 Defining DAGs

Before finalizing ontologies' status as DAGs, first DAGs must be defined. The following definitions were adapted from *Digraphs Theory, Algorithms and Applications* [3].

Definition 1. A directed graph D is a non-empty finite set $V(D)$ of **vertices** and a finite set $E(D)$ of ordered pairs of distinct vertices called **edges**. Edges have a tail and a head, meaning that the edge (u, v) is different from the edge (v, u).

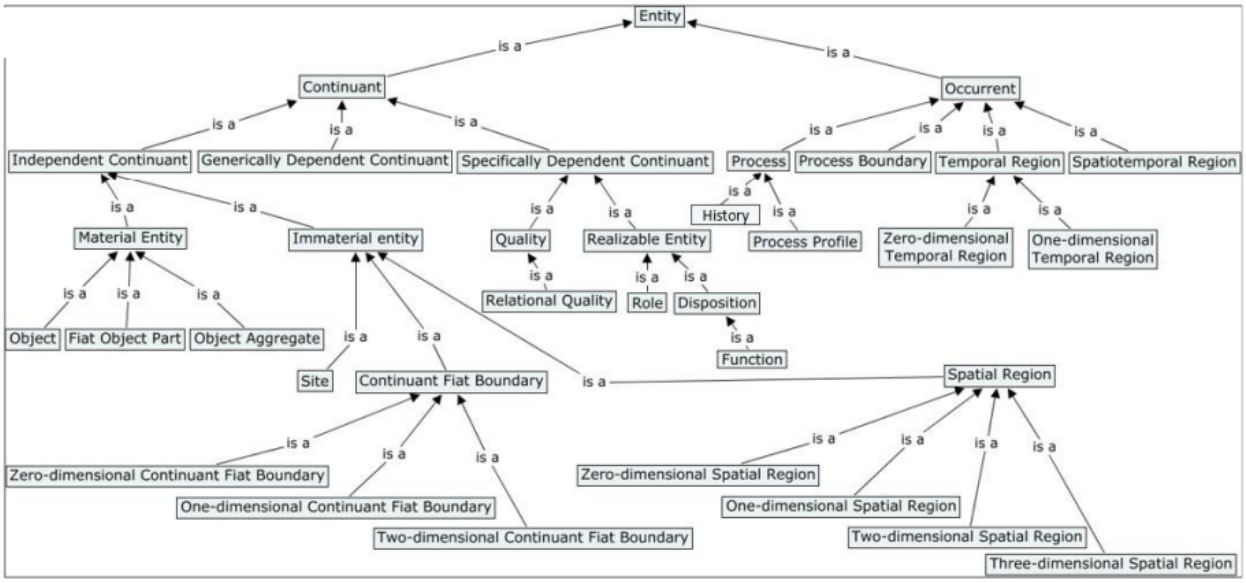

Figure 3: BFO 2.0 Hierarchy from the BFO Github [7]

Definition 2. Directed Acyclic Graph: A directed graph without a cycle.

Definition 3. A cycle is a non-trivial path from vertex v to v.

With these definitions, it is possible to organize the BFO ontology as a DAG. The initial step is straightforward. If the terms are the nodes and the relations between them the edges, and since the BFO "is_a" relation is a direction relation, there is clearly a directed graph. The difficulty comes from ensuring that there are no cycles within the directed graph, and this is where it becomes crucial to limit the current project to measuring over only the BFO ontology. Due to BFO being a single finite, and small ontology, proving that it does not contain cycles is trivial. Figure 3 displays the total BFO hierarchy. Since, all new terms are assigned to a preexisting BFO classification, they would be leaf nodes, or nodes with no children.[5] No cycles are found within the BFO-defined classes, and by just adding leaf nodes, without relations between them, the resulting ontology, and its graph representation, will also not contain cycles.

[5]At the top of BFO everything is a subclass of **entity**. In a mirroring sense there is a BFO term, the BFO **nothing**, which every term and superclass is a parent of. This is largely overlooked in this paper since it does not change BFO's status as a DAG and in fact could possibly be used to narrow the ontology's structure down further into a lattice.

2.3 DAG Guarantees in Domain Ontologies

If the TLO BFO is a DAG, then what can be guaranteed of BFO-compliant domain ontologies? Since the DBOL will ideally be used to create full domain ontologies, being able to measure over a finite well-established set of terms is useful, but not rather interesting. Instead it is desirable to extend any measurement from BFO into the domain ontologies.

It is the generally held belief within Applied Ontology that domain ontologies are DAGs [2]. If an ontology is built with the BFO best practices, it will be a DAG and ideally might even be a tree. However, multiple inheritance is common within domain ontologies, and some may even be afflicted with sets of cyclic terms. As mentioned in subsection 1.3, cycles of two are easy to find, however cycles of three can be harder to detect and often require computers to find[8]. As far as research can find, no one has proven that domain ontologies are DAGs, and for this topic study this is not a concern since the current measurements are solely over the BFO hierarchy. This proof of whether domain ontologies are DAGs is outside the scope of this topic study and left for another day.

3 Similarity Metrics

3.1 Defining Katz Similarity

Now that the BFO ontology can be abstracted into a DAG, it is possible to start measuring the difference in term assignments. For this project Katz Similarity (KS) has been chosen for the task. KS was first coined by Leo Katz in 1953 and was originally known as Katz Centrality and was used to measure the "influence of an actor within a graph" [4].

Because, the specific case being measured, where both graphs share a set of nodes and have a different edge set, KS is a perfect candidate. Additionally, KS accounts for the direction of edges within an ontology as well as helps to measure the importance of the parent child relation and both the length and number of all paths between two nodes in the graph. All of which help incorporate the concept of the least common ancestor (LCA). The LCA is why in figure 2 the left assignment is closer. The LCA of **object** and **object aggregate** is **material entity**, while the LCA of **object** and **immaterial entity** is **independent continuant**. Since the least common ancestor of the first is closer, terms classified within their superclass are more similar.

This project's implementation of KS is based heavily off of the work of Nayak et. al, who used KS to measure differences of knowledge hierarchies and their evolution. The knowledge hierarchies examined in that project are closely related to ontologies, and were mostly focused on large language repositories such as WordNet [5]. Together this all adds up to KS earning a place as the front runner of similarity metrics for this project.

The final comparison of two graphs is known as the Katz Graph Similarity (KGS) and requires three steps to find. The KGS is a calculation requiring two vectors known as the Katz Similarity Vector (KSV) of each graph, and to find the KSV of each graph you must calculate the KS of each node.

When working with KS it is important to label the two graphs distinctly, say Graph

One (G_1) and Graph Two (G_2), note that both are DAGs that have the same nodes or vertices. Throughout the remaining explanation of KS these graphs will be consistently referenced.

The first step in finding the KGS is finding the KS between the nodes of the graph.

Definition 4. [5] The Katz Similarity between nodes u and v of the graph G; is defined as

$$KS(u, v) = \sum_{l} t\alpha^l$$

where t is the number of paths length l from u to v and $0 < \alpha < 1$. Note: α is a tunable variable typically set to 0.05.

To help illustrate the KS between two nodes consider the following. There are two vertices u and v of the graph G. They have one path length 1 and three paths length 2. Therefore, the Katz similarity is equal to $\alpha + 3\alpha^2$.

All of the KS values can then be stored in the KSV.

Definition 5. [5] The Katz Similarity Vector (KSV): A graph can be represented by its KSV where the n^{th} element of the vector is the Katz Similarity of the n^{th} pair of vertices in $V \times V$.

To maintain consistency between graphs, the KSV should be ordered by breadth-first traversal[6]. Consider a set of nodes in graph G, with m nodes, in order of a breadth-first search so each node is numbered $1, 2, \ldots, m$. The vector can then take form:

$$KSV(G) = \Big[KS(1,1), \quad KS(1,2), \quad \cdots \quad KS(1,m), \quad KS(2,1), \quad \cdots \quad KS(m,m)\Big].$$

The final step is to calculate the KGS using the newly formed KSVs.

[6]The KSVs being in the same order is all that matters, a breadth-first traversal was chosen for ordering to maintain consistency throughout the project, but when implemented the KSVs were stored as hashmaps making order irrelevant.

Definition 6. [5] Katz Graph Similarity: Given two Katz similarity vectors KSV_1 and KSV_2 derived from graphs G_1 and G_2, with the same vertex set. Then,

$$KGS(G_1, G_2) = \frac{2}{1 + \exp(\gamma||KSV_1(:, i) - KSV_2(:, i)||_p)}$$

where $||.||_p$ is the L_p-norm of the i^{th} vector differences and $\gamma > 0$ is tunable parameter.

For this project implementation, an L_p norm of 1 was used, measuring the vectors in Manhattan distance. The KGS(G_1,G_2) is the final result and is bounded as $0 < KGS \leq 1$, where a KGS of 1 is returned for identical graphs and as the graphs become more different the similarity value decreases.

3.2 Speeding up Katz

However calculating the KS within a graph can start to cause issues. This part of the procedure is computationally expensive since every node must be compared to every other node in the graph. The time complexity caused by this is the main downside to using KS and in large ontologies that can contain tens of thousands of terms, KS seems unusable. However, there is a method to speed up the calculation of the KS. The properties of DAGs can be harnessed to calculate the KS without comparing every node to each other.

Instead of computing the KS in any order, the KS must be evaluated in order of depth first traversal. Before comparing any nodes first the graph must be sorted in "topological order", by depth, with the root being place at level 0, and every other node being placed in lowest possible level when compared to the root. For example, the root and a node v can be connected by two paths, the first of length two, the second of length 2, the second of length 4, then v belongs in in level 2. Calculating KS is now done from the lowest topological ordering to the highest, using the following formula.

Definition 7. [5] For a graph G; in topological order with vertices u,v such that,

$$KS'(u,v) = \alpha \times \left(\sum_{p \in parents(v)} KS'(u,p) \right) + \alpha \times \Upsilon(u \to v)$$

where $\Upsilon(u \to v) = 1$ when there is an edge from u to v and 0 otherwise.

Consider the following reasoning as to why this is possible. Let G be a DAG with root r. Also consider a subset of G's nodes, p_1, p_2, $\ldots p_n$, and v of which all p_i are the parents of v. Since the only way to traverse to v is through one of its parents, and since KS is being calculated by topological order, when calculating the similarity between any node called u belonging to a lower order than v then $KS(u,p)$ is already known. Therefore, the $KS(u,v) = KS(u,p) + \alpha$, if u and v are connect and $KS(u,v) = KS(u,p)$ otherwise.

After the topological level is computed the nodes are "removed" from the graph. Removed being a lose term that means to stop calculating over them. Whether the terms are being physically removed from a database is a decision made depending on the language KS is implemented in.

3.3 Implementing Katz Similarity in Java

Previous implementations of KS have been programmed in C. However, the DBOL is a Java project and thus for this paper KS will be implemented in Java. Choosing to write this code in Java leads to similar code as the C implementation, but calculating the individual KSs differs. Instead of finding a topological ordering and then computing in that order, the calculation of KS is done starting at the root and then by performing a breadth-first traversal throughout the graph. By moving through the graph breadth-first, the program moves through in ascending topological order. As KS values are calculated, instead of removing nodes from the graph, the values are stored in hashmaps that are then later used as the KSV to compute the final KGS. The KS function itself is recursive to allow for fewer calculations, instead using the quick look-ups of the hashmaps to cut the

	object aggregate*	**immaterial entity**
$\alpha = 0.05, \gamma = 0.05$	1.0	0.9999999921875001
$\alpha = 0.10, \gamma = 0.05$	1.0	0.9999997500000001
$\alpha = 0.05, \gamma = 0.10$	1.0	0.9999999843749998

Figure 4: KGS of *laptop* as both an **object aggregate** and **immaterial entity** with various α and γ

computation time. The code itself can be found in the appendix section 5.

3.4 Performance on Ontologies

Now that Katz Similarity has been defined, sped up, and, most importantly, implemented in Java, does it work? To test this, remember the previous example of assigning the term **laptop**. The BFO ontology was imported as the required edge list format, and **laptop** was added to three separate version of the file with its correct, close, and far classifications. Then the KGS was found for both the close and far classification examples, with varying α and γ. All of which can be seen in figure 4.

While the **immaterial entity** column is performing as expected, the **object aggregate** is not. Since a KGS equal to 1 is only supposed to be possible when the graphs are identical, this appears to be a major issue. However, the reasoning lies in precision. The definition of KGS assumes that however the value is being computed, there is an infinite level of precision, or in other words that rounding errors will not occur. The difference in the correct assignment of *laptop* and its assignment of **object aggregate** are simply too close for Java doubles to hold the difference. This was an issue that was expected on large ontologies, but was surprising to see with BFO since it is so small (only 38 relations). However, this issue can be avoided, while also speeding up the calculation of the KGS. Instead of attempting to measure the entirety of both graphs, instead the KGS can be approximated.

While this project was not able to implement the approximation of KGS, it is the next

logical step. The process is straightforward. Instead of calculating the KS of each node to every other node, instead small sections of the graphs are extracted, and the KGSs of these subgraphs are calculated [5]. This reduces the number of computations needed to calculate the KGS, which beats out even the sped up version of KGS using both the KS' and breadth-first traversal scheme. Additionally, by only examining small portions of the larger graph at a time, precision can be kept while avoiding the use of specialized types, such as BigDecimal, that retain precision, but add overhead.

As mentioned, the code written to implement this project can be found in the appendix, section 5.

4 Conclusion

Future work on this project would have a few next steps. First, the code base needs to be reformatted to address some choices made in early stages of development that no longer make sense. The Vertex class and the AuxUtils class can be either removed, or their functionality moved into the CalcUtils class, which can be more meaningfully renamed to Graph, since it represents and allows for calculation over the ontology as a graph. Additionally, work would need to be done to compile and then import the user classifications from the parent projects results. A pipeline could be set up to transform the results from their current format into usable edge files, along with another edge set of agreed classifications. Finally, the most important next step would be to implement the KGS approximation. With BFO alone being too large to detect small changes, having the BFO hierarchy, along with hundreds of user terms, would likely also cause loss of precision and rounding issues.

However, it is clear that the use of KGS similarity is a possible path to improve the evaluation of new tools being built to help ontologists, not only for the DBOL project team, but in any case where two ontologies of the same term need to be compared.

5 Reflection

This project is an offshoot of the larger project that I worked on as an independent study, and then eventually as my Honors Thesis. I enjoyed being able to approach Ontology with a more mathematical lens, and since the goal of the parent project is aimed at bringing more structure to creating ontologies, this fit in great. When I started as a Math major I had never programmed before, and only wandered into the field due to the cognate requirements of the degree. I am glad I did since the intersection between the two fields is where I find the work I enjoy the most.

Overall, I thought the capstone project was a great opportunity to bring in parts of my degree from Math and Computer Science. This was especially apparent when I was presenting for this project. When presenting at both the MAA Seaway Meeting, and at capstone presentations, I had to make sure to take care explaining topics from Computer Science. On the other hand, when sharing this work with the DBOL team, I found myself explaining the Math behind the similarity metric and why Katz Similarity was the right choice. Having spent so much time at the crossing of Math and CS I found myself blending which topics came from each, and having to slow down and remind myself so that I could make the work accessible to others.

By far my favorite part of this capstone, and the honors thesis, was presenting about the work and sharing it with others. Both in formal settings like presentations and meetings, as well as informal when talking with classmates. This was a great experience and I would encourage others to try and combine their work for a Math capstone with other research when possible.

Bibliography

[1] ISO/IEC 21838-2:2021 Top-level ontologies (TLO) Part 2: Basic Formal Ontology (BFO). `https://www.iso.org/standard/74572.html`. Accessed: 2023-11-18.

[2] Robert Arp, Barry Smith, and Andrew D. Spear. *Building Ontologies with Basic Formal Ontology*. MIT Press, 2015.

[3] Jørgen Bang-Jensen and Gregory Z. Gutin. *Digraphs Theory, Algorithms and Applications*. Springer, 2010.

[4] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, Mar 1953.

[5] Guruprasad Nayak, Sourav Dutta, Deepak Ajwani, Patrick Nicholson, and Alessandra Sala. Automated assessment of knowledge hierarchy evolution: Comparing directed acyclic graphs. *Information Retrieval Journal*, 22(3–4):256–284, Aug 2019.

[6] Alan Ruttenberg. Basic Formal Ontology (BFO). `https://basic-formal-ontology.org/`. Accessed: 2023-11-18.

[7] Alan Ruttenberg. Basic Formal Ontology (BFO) Github Repository. `https://github.com/BFO-ontology/BFO`. Accessed: 2023-11-18.

[8] Selja Seppälä, Alan Ruttenberg, and Barry Allen. Guidelines for writing definitions in ontologies. *Ciência da Informação*, 46(1):73–88, 2017.

Appendix

This appendix includes the code written to implement calculating the KGS in Java. The code can also be found online at in this GitHub repository https://github.com/KeithTAllen/Katz-Similarity.

5.1 Main class

Main class currently set up to show the *laptop* example discussed in Section 1.3.

```
import java.io.FileNotFoundException;
import java.util.HashMap;

public class Main {
    public static void main(String[] args) throws FileNotFoundException {
        /*
         * This main method calculates both the close and far assignments
         * of terms discussed in the paper. Attempts were made to also
         * calculate using both depth first and slow searches. However,
         * since the BFO ontology is small, accurate measurements were
         * not possible.
         */
        double closeKGS = getKgsBreadthFirst(
                "src/main/resources/LaptopCorrect",
                "src/main/resources/LaptopClose");
        System.out.println("Close assignment KGS: " + closeKGS);

        double farKGS = getKgsBreadthFirst(
                "src/main/resources/LaptopCorrect",
                "src/main/resources/LaptopFar");
        System.out.println("Far assignment KGS: " + farKGS);
    }

    /*
     * Calculates the KGS of two given graph given as edge lists, using
     * breadth first search. Order of steps to find KGS:
     *  create AuxUtils
     *  auxUtils read edge lists
     *  create CalcUtils
     *  get hashmap returned from a calcKS function
     *  call KGS
     */
    public static double getKgsBreadthFirst(String fileA, String fileB)
            throws FileNotFoundException {
        // Create representation of each graph
        AuxUtils auxUtilsA = new AuxUtils();
        auxUtilsA.readEdgeList(fileA);

        AuxUtils auxUtilsB = new AuxUtils();
        auxUtilsB.readEdgeList(fileB);

        // Create CalcUtils for each graph
        CalcUtils calcUtilsA = new CalcUtils(auxUtilsA.parents,
```

```
                auxUtilsA.children,
                auxUtilsA.allVertices,
                auxUtilsA.root);

        CalcUtils calcUtilsB = new CalcUtils(auxUtilsB.parents,
                auxUtilsB.children,
                auxUtilsB.allVertices,
                auxUtilsB.root);

        // Calculate KSV
        HashMap<String, Double> KSVA =
                calcUtilsA.breadthFirstCalculateKSV(auxUtilsA.root);

        HashMap<String, Double> KSVB =
                calcUtilsB.breadthFirstCalculateKSV(auxUtilsB.root);

        // Calculate and return KGS
        return calcUtilsA.calculateKGS(KSVA, KSVB);
    }

    public static void printGraph(AuxUtils auxUtils){
        System.out.println("Root: " + auxUtils.root);
        System.out.println("Full vertex list: " + auxUtils.allVertices);
    }
}
```

Main class output in which it can be seen that there is not enough precision to detect small classification and these must instead be solved by approximation.

```
Close assignment KGS: 1.0
Far assignment KGS: 0.9999999921875001

Process finished with exit code 0
```

5.2 AuxUtils class

Used to convert from a parent child end list into an Object representing the graph.

```
/*
 * Functions needs to organize and parse through the data are stored here
 * and can be accessed by creating an AuxUtils object. Originally AuxUtils
 * was also supposed to handle topological ordering and other bookkeeping.
 * This was instead handle via traversal order in the CalcUtils
 */

import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.Scanner;

public class AuxUtils {
    /*
```

```
     * variable for AuxUtils
     */

    // list of parents within a graph | Key = Child |
    // Value = ArrayList of the child's parents | No root
    HashMap<Vertex, ArrayList<Vertex>> parents = new HashMap<>();
    // list of children within the graph | Key = Parent |
    // Value + ArrayList of all the parents children
    HashMap<Vertex, ArrayList<Vertex>> children = new HashMap<>();
    // distinct list of all vertices in the graph
    HashSet<Vertex> allVertices = new HashSet<>();
    // the root of the graph determined by finding the vertex key in
    // parents that has a null ArrayList
    Vertex root;

    /*
     * readEdgeListFile
     *
     * Creates a parents list, a child list, find the root , and a list
     * of all vertices.
     */
    public void readEdgeList(String fileName)
            throws FileNotFoundException {
        // establish scanner with the file
        Scanner scanner = new Scanner(new File(fileName));

        // for every line in the File
        while(scanner.hasNextLine()){
            // grab the line
            String line = scanner.nextLine();

            // create a vertex for the lines parent and child
            Vertex parent = new Vertex(line.substring(
                    0, line.indexOf(' ')));
            Vertex child = new Vertex(line.substring(
                    line.indexOf(' ') + 1));

            // add to the HashSet of all vertices
            allVertices.add(parent); allVertices.add(child);

            // Create parents set
            // if the Vertex already exists in the parents map.
            ArrayList<Vertex> parentSet;
            if(parents.containsKey(child)) {
                parentSet = parents.get(child);
            }
            // not in the parents map
            else { parentSet = new ArrayList<>(); }
            // add the current parent and place back in HashMap
            parentSet.add(parent);
            parents.put(child, parentSet);

            // Create children set
            // if the Vertex already exists in the child map.
```

```
            ArrayList<Vertex> childrenSet;
            if(children.containsKey(parent)) {
                childrenSet = children.get(parent);
            }
            // not in the children
            else { childrenSet = new ArrayList<>(); }
            // add the current child and place back in HashMap
            childrenSet.add(child);
            children.put(parent, childrenSet);
        }

        // find the root of the graph
        for(Vertex v : allVertices){
            if (parents.get(v) == null){
                root = v;
            }
        }
    }
}
```

5.3 CalcUtils class

Used to calculate the KS, KSV, and KGS of a graph.

```
import java.util.*;

public class CalcUtils {

    // instance variables
    HashSet<Vertex> allVertices;
    HashMap<String, Double> katzSimilarities;
    HashMap<Vertex, ArrayList<Vertex>> parents;
    HashMap<Vertex, ArrayList<Vertex>> children;
    HashMap<String, Double> ksv;
    Vertex root;
    double alpha = 0.05;
    double gamma = 0.05;

    /*
     * Constructors
     */

    // Used default value of 0.05 for alpha and gamma
    public CalcUtils(HashMap<Vertex, ArrayList<Vertex>> parents,
                     HashMap<Vertex, ArrayList<Vertex>> children,
                     HashSet<Vertex> allVertices, Vertex root){
        this.katzSimilarities = new HashMap<>();
        this.ksv = new HashMap<>();
        this.parents = parents;
        this.children = children;
        this.allVertices = allVertices;
        this.root = root;
    }
```

```
30
31      // sets alpha and gamma to the given parameters
32      public CalcUtils(HashMap<Vertex, ArrayList<Vertex>> parents,
33                       HashMap<Vertex, ArrayList<Vertex>> children,
34                       HashSet<Vertex> allVertices, Vertex root ,
35                       double alpha, double gamma){
36          this.katzSimilarities = new HashMap<>();
37          this.ksv = new HashMap<>();
38          this.parents = parents;
39          this.children = children;
40          this.allVertices = allVertices;
41          this.root = root;
42          this.alpha = alpha;
43          this.gamma = gamma;
44      }
45
46      /*
47       * Used to calculate the Katz Similarity Vector (ksv). Does assume
48       * that parents is an exhaustive list of vertices from the graph.
49       * Each graph gets its own calcUtils
50       */
51      public HashMap<String, Double> slowCalculateKSV(){
52          HashMap<String, Double> slowKsv = new HashMap<>();
53          // for every vertex u in the graph
54          for (Vertex u : allVertices) {
55              // for every vertex v in the graph
56              for (Vertex v: allVertices){
57                  // create the key for the ksv
58                  String key = u.toString() + v.toString();
59                  // calculate ks(u,v) and place it in the hashmap
60                  slowKsv.put(key, calculateKS(u,v));
61              }
62          }
63          return slowKsv;
64      }
65
66      // depthFirst search of the child nodes
67      public HashMap<String, Double> depthFirstCalculateKSV(Vertex u){
68          // for every vertex
69          for (Vertex v : allVertices) {
70              // create the key for the ksv
71              String key = u.toString() + v.toString();
72              // calculate ks(u,v,alpha) and place it in the hashmap
73              ksv.put(key, calculateKS(u,v));
74          }
75          // base case: if u has no children
76          if(children.get(u) == null || children.get(u).isEmpty()) {
77              // do nothing
78          }
79          // recursive step
80          else {
81              // for each of the children
82              for (Vertex v : children.get(u)) {
83                  // call this recursively on all the child's children
```

```
84                  depthFirstCalculateKSV(v);
85              }
86          }
87          // once done return the ksv
88          return ksv;
89      }
90
91      // breadthFirst search of the child nodes
92      public HashMap<String, Double> breadthFirstCalculateKSV(Vertex root){
93          // create a queue of what node to visit next and a HashSet of
94          // nodes we have visited
95          Queue<Vertex> vertexQueue = new LinkedList<>();
96          HashSet<Vertex> visited = new HashSet<>();
97          HashMap<String, Double> tempKSV = new HashMap<>();
98
99          // add the root to the queue
100         visited.add(root);
101         vertexQueue.add(root);
102
103         // while we still have nodes to visit
104         while ( !vertexQueue.isEmpty()) {
105             Vertex u = vertexQueue.poll();
106
107             for (Vertex m : allVertices) {
108                 // create the key for the ksv
109                 String key = u.toString() + m.toString();
110                 // calculate ks(u,v) and place it in the hashmap
111                 tempKSV.put(key, calculateKS(u, m));
112             }
113
114             // if u has no children we have nothing to do. Skip ahead to
115             // next dequeue
116             if(children.get(u) == null || children.get(u).isEmpty()) {
117                 // do nothing
118             }
119
120             // otherwise u has children, then add them to the queue.
121             else {
122                 for ( Vertex v : children.get(u)) {
123                     // if we have already visited v
124                     if ( !visited.contains(v)) {
125                         // we have visited v
126                         visited.add(v);
127                         // add v to the queue, so we can do the same in
128                         // the next run.
129                         vertexQueue.add(v);
130                     }
131                 }
132             }
133         }
134         // once done return the ksv
135         return tempKSV;
136     }
137
```

```
    // calculates the KS of two given nodes using the global alpha
    public double calculateKS(Vertex u, Vertex v){
        String key = u.toString() + v.toString();
        // base case
        if(u.equals(v)){ return 0.0; }
        // the value has already been calculated, just return it
        else if(katzSimilarities.containsKey(key)){
            return katzSimilarities.get(key);
        }
        // otherwise calculate the value
        else {
            double parentsSum = 0;
            if (parents.get(v) == null) { return 0; } // special root case
            // get the sum of the values of all the parents
            for ( Vertex p : parents.get(v) ) {
                parentsSum += calculateKS(u, p);
            }
            // calculate the final katz similarity and place in the map
            double katzSim = (alpha * parentsSum) +
                    (alpha * isConnected(u,v));
            katzSimilarities.put(key, katzSim);
            return katzSim;
        }
    }

    // returns 1 if there is a path from parent to child, otherwise 0
    private double isConnected(Vertex u, Vertex v) {
        if (parents.get(v).contains(u))
            return 1.0;
        else
            return 0.0;
    }

    // calculate the Katz Graph Similarity measurement
    public double calculateKGS(HashMap<String, Double> ksva,
                               HashMap<String, Double> ksvb){
        double denominator = (1 + (Math.exp(gamma * lpNorm(ksva, ksvb))));
        return 2/denominator;
    }

    private double lpNorm(HashMap<String, Double> ksva,
                          HashMap<String, Double> ksvb) {
        double total = 0.0;
        for (String key : ksva.keySet()){
            try {
                double v = Math.abs(ksva.get(key)) -
                        Math.abs(ksvb.get(key));
                total += v;
            }
            // if the current vertex pair is not one of the ksvs
            catch (NullPointerException n){
                System.out.println(n + "KSV is not complete");
                System.out.println(key);
            }
```

```
        }
        return total;
    }
}
```

5.4 Vertex class

Used to represent a vertex, or node, within the graph. This ended up being a poor implementation choice.

```
/*
 * At one point the Vertex was supposed to know much more about itself,
 * in hindsight a String would have worked just as well.
 */

import java.nio.ByteBuffer;

public class Vertex {
    // instance variables
    private String name;

    // constructor
    public Vertex(String name) {
        this.name = name;
    }

    // simple getters and setters
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    // toString and compare
    public String toString() {
        return name;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof Vertex))
            return false;
        Vertex other = (Vertex) o;
        boolean nameEquals = (this.name == null && other.name == null)
                || (this.name != null && this.name.equals(other.name));
        return nameEquals;
    }

    @Override
    public int hashCode() {
        ByteBuffer bytes = ByteBuffer.wrap(name.getBytes());
        return bytes.hashCode();
    }
}
```

5.5 Importer class

Reads in and converts a .owl file into a parent child edge list which can then be used to calculate the KGS.

```
/*
 * Class to import an .owl file and create a parent child edge set list
 * needed to calculate the KGS. Based heavily upon the work of
 * Dr. Daniel Schlegel (GitHub: digitalneoplasm) from the parent project
 * of this project, that I also worked on.
 */

import org.semanticweb.owlapi.apibinding.OWLManager;
import org.semanticweb.owlapi.model.*;
import org.semanticweb.owlapi.reasoner.OWLReasoner;
import org.semanticweb.owlapi.reasoner.structural
        .StructuralReasonerFactory;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Importer {

    // LABEL_NAMING_MODE decides not to use IRIs in the LaptopCorrect,
    // instead naming terms after their labels.
    private static final boolean LABEL_NAMING_MODE = true;

    private static final Map<String, String> iriLabelMap =
            new HashMap<>();

    public static void main(String[] args)
            throws OWLOntologyCreationException, IOException {
        OWLOntology ontology = loadOntology(
                "src/main/resources/bfo.owl");

        FileWriter writer = new FileWriter(
                "src/main/resources/output");

        // Write OWL Subclass / Superclass Data //
        List<OWLClass> classes = ontology.classesInSignature()
                .filter(c -> !isDeprecated(c, ontology)).toList();
        List<OWLObjectProperty> properties = ontology
                .getObjectPropertiesInSignature().stream()
                .filter(c -> !isDeprecated(c, ontology)).toList();

        // get a reasoner for the ontology
        OWLReasoner reasoner = (new StructuralReasonerFactory())
                .createReasoner(ontology);

        /*
```

```
        go through each class and create the edge list file that
        we are looking for
         */

        if(LABEL_NAMING_MODE) resolveLabels(classes,
                properties, ontology);

        // for every class in the ontology
        for(OWLClass c : classes){
            // for every subclass write the relation of
            // the two into the file
            for(OWLClass sub : getDirectSubclasses(c, reasoner)){
                String cl = c.getIRI() + "";
                String su = sub.getIRI() + "";
                writer.write(cl.substring(
                        cl.indexOf("#") + 1) + " " +
                        su.substring(su.indexOf("#") + 1) +
                        "\n");
            }
        }

        writer.close();
    }

    public static boolean isDeprecated(OWLEntity oc, OWLOntology ont){
        if (oc.getIRI().getShortForm().contains("ObsoleteClass"))
            return true;
        for(OWLAnnotationAssertionAxiom oaaa :
                ont.getAnnotationAssertionAxioms(oc.getIRI())){
            if (oaaa.getProperty().getIRI().getShortForm()
                    .equals("IAO_0100001")) // term replaced by
                return true;
        }
        return ont.getAnnotationAssertionAxioms(oc.getIRI()).stream()
                .anyMatch(a -> a.getProperty().isDeprecated() &&
                        a.getValue() instanceof OWLLiteral &&
                ((OWLLiteral) a.getValue()).getLiteral().equals("true"));
    }

    // takes in a filename and creates an ontology
    // from the given owl document
    public static OWLOntology loadOntology(String inputFilename)
            throws OWLOntologyCreationException {
        OWLOntologyManager manager = OWLManager
                .createOWLOntologyManager();
        return manager.loadOntologyFromOntologyDocument(
                new File(inputFilename));
    }

    // get the subclasses of a class in the onotlogy
    public static List<OWLClass> getDirectSubclasses(
            OWLClassExpression oce, OWLReasoner reasoner){
        return reasoner.getSubClasses(oce, true).entities()
                .filter(oc -> !isNothingClass(oc)).toList();
```

```
104     }
105
106     // checks if the class is the BFO nothing?
107     public static boolean isNothingClass(OWLClass oc){
108         return oc.getIRI().getShortForm().equals("Nothing");
109     }
110
111     public static void resolveLabels(List<OWLClass> classes,
112                                      List<OWLObjectProperty> properties,
113                                      OWLOntology ont){
114         for (OWLClass c : classes){
115             for(OWLAnnotationAssertionAxiom oaaa :
116                     ont.getAnnotationAssertionAxioms(c.getIRI()))
117                 if (oaaa.getProperty().getIRI().getShortForm()
118                         .equals("label")) {
119                     String oaaastr = oaaa.getValue().toString();
120                     int langat = oaaastr.lastIndexOf("@");
121                     int typestr = oaaastr.lastIndexOf("^^xsd");
122                     oaaastr = (langat > -1) ? oaaastr.substring(0, langat)
123                             : oaaastr;
124                     oaaastr = (typestr > -1 && typestr > langat && typestr
125                             <= oaaastr.length()) ?
126                             oaaastr.substring(0, typestr) : oaaastr;
127                     iriLabelMap.put(c.getIRI().getShortForm(),
128                             oaaastr.toLowerCase());
129                 }
130         }
131         for (OWLObjectProperty p : properties){
132             for(OWLAnnotationAssertionAxiom oaaa :
133                     ont.getAnnotationAssertionAxioms(p.getIRI()))
134                 if (oaaa.getProperty().getIRI().getShortForm()
135                         .equals("label")) {
136                     String oaaastr = oaaa.getValue().toString();
137                     int langat = oaaastr.lastIndexOf("@");
138                     int typestr = oaaastr.lastIndexOf("^^xsd");
139                     oaaastr = (langat > -1) ? oaaastr.substring(0, langat)
140                             : oaaastr;
141                     oaaastr = (typestr > -1 && typestr > langat && typestr
142                             <= oaaastr.length()) ?
143                             oaaastr.substring(0, typestr) : oaaastr;
144                     iriLabelMap.put(p.getIRI().getShortForm(),
145                             oaaastr.toLowerCase());
146                 }
147         }
148
149     }
150 }
```

5.6 Edge List Inputs

Three ontologies are used in the Main class to demo; LaptopCorrect, Laptop Close, and LaptopFar. These are three edge list text files that represented each ontology and differ

only in the parent of the term **Laptop**.
LaptopCorrect:

```
Entity Continuant
Entity Occurrent
Continuant IndependentContinuant
Continuant SpecificallyDependentContinuant
Continuant GenericallyDependentContinuant
IndependentContinuant MaterialEntity
IndependentContinuant ImmaterialEntity
MaterialEntity Object
MaterialEntity FiatObjectPart
MaterialEntity ObjectAggregate
ImmaterialEntity Site
ImmaterialEntity ContinuantFiatBoundary
ImmaterialEntity SpatialRegion
SpatialRegion OneDimensionalSpatialRegion
SpatialRegion ThreeDimensionalSpatialRegion
SpatialRegion TwoDimensionalSpatialRegion
SpatialRegion ZeroDimensionalSpatialRegion
ContinuantFiatBoundary OneDimensionalFiatBoundary
ContinuantFiatBoundary TwoDimensionalFiatBoundary
ContinuantFiatBoundary ThreeDimensionalFiatBoundary
ContinuantFiatBoundary ZeroDimensionalFiatBoundary
SpecificallyDependentContinuant Quality
SpecificallyDependentContinuant RealizableEntity
SpecificallyDependentContinuant Role
SpecificallyDependentContinuant Function
SpecificallyDependentContinuant Disposition
RealizableEntity Role
RealizableEntity Disposition
Disposition Function
Occurrent Process
Occurrent ProcessBoundary
Occurrent TemporalRegion
Occurrent SpatiotemporalRegion
Process History
Process ProcessProfile
TemporalRegion ZeroDimensionalTemporalRegion
TemporalRegion OneDimensionalTemporalRegion
Object Laptop
```

LaptopClose:

```
Entity Continuant
Entity Occurrent
Continuant IndependentContinuant
Continuant SpecificallyDependentContinuant
Continuant GenericallyDependentContinuant
IndependentContinuant MaterialEntity
IndependentContinuant ImmaterialEntity
MaterialEntity Object
MaterialEntity FiatObjectPart
MaterialEntity ObjectAggregate
ImmaterialEntity Site
```

```
ImmaterialEntity ContinuantFiatBoundary
ImmaterialEntity SpatialRegion
SpatialRegion OneDimensionalSpatialRegion
SpatialRegion ThreeDimensionalSpatialRegion
SpatialRegion TwoDimensionalSpatialRegion
SpatialRegion ZeroDimensionalSpatialRegion
ContinuantFiatBoundary OneDimensionalFiatBoundary
ContinuantFiatBoundary TwoDimensionalFiatBoundary
ContinuantFiatBoundary ThreeDimensionalFiatBoundary
ContinuantFiatBoundary ZeroDimensionalFiatBoundary
SpecificallyDependentContinuant Quality
SpecificallyDependentContinuant RealizableEntity
SpecificallyDependentContinuant Role
SpecificallyDependentContinuant Function
SpecificallyDependentContinuant Disposition
RealizableEntity Role
RealizableEntity Disposition
Disposition Function
Occurrent Process
Occurrent ProcessBoundary
Occurrent TemporalRegion
Occurrent SpatiotemporalRegion
Process History
Process ProcessProfile
TemporalRegion ZeroDimensionalTemporalRegion
TemporalRegion OneDimensionalTemporalRegion
ObjectAggregate Laptop
```

LaptopFar:

```
Entity Continuant
Entity Occurrent
Continuant IndependentContinuant
Continuant SpecificallyDependentContinuant
Continuant GenericallyDependentContinuant
IndependentContinuant MaterialEntity
IndependentContinuant ImmaterialEntity
MaterialEntity Object
MaterialEntity FiatObjectPart
MaterialEntity ObjectAggregate
ImmaterialEntity Site
ImmaterialEntity ContinuantFiatBoundary
ImmaterialEntity SpatialRegion
SpatialRegion OneDimensionalSpatialRegion
SpatialRegion ThreeDimensionalSpatialRegion
SpatialRegion TwoDimensionalSpatialRegion
SpatialRegion ZeroDimensionalSpatialRegion
ContinuantFiatBoundary OneDimensionalFiatBoundary
ContinuantFiatBoundary TwoDimensionalFiatBoundary
ContinuantFiatBoundary ThreeDimensionalFiatBoundary
ContinuantFiatBoundary ZeroDimensionalFiatBoundary
SpecificallyDependentContinuant Quality
SpecificallyDependentContinuant RealizableEntity
SpecificallyDependentContinuant Role
SpecificallyDependentContinuant Function
```

```
SpecificallyDependentContinuant Disposition
RealizableEntity Role
RealizableEntity Disposition
Disposition Function
Occurrent Process
Occurrent ProcessBoundary
Occurrent TemporalRegion
Occurrent SpatiotemporalRegion
Process History
Process ProcessProfile
TemporalRegion ZeroDimensionalTemporalRegion
TemporalRegion OneDimensionalTemporalRegion
ImmaterialEntity Laptop
```