# Searching for Class Models

Maxim Bragilovski, Yifat Makias, Moran Shamshila, Roni Stern, and Arnon Sturm

Ben-Gurion University of the Negev, Beer Sheva, Israel
maximbr@post.bgu.ac.il, yifatmakias@gmail.com, moranshe9@gmail.com, sternron@bgu.ac.il, sturm@bgu.ac.il

**Abstract.** Models in model-based development play a major role and serve as the main design artifacts, in particular class models. As there are difficulties in developing high-quality models, different repositories of models are established to address that challenge, so developers would have a reference model. Following the existence of such repositories, there is a need for tools that can retrieve similar high-quality models. To search for models in these repositories, we propose a greedy algorithm that matches the developer's intention by considering semantic similarity, structure similarity, and type similarity. The initial evaluation indicates that the algorithm achieved high performance in finding the relevant class model fragments. Though additional examination is required, the sought algorithm can be easily adapted to other modeling languages for searching models and their encapsulated knowledge.

**Keywords:** Model Driven Development · Model Repository · Model Search

## 1 Introduction

Model-Based Development is continuously evolving [1]. In such a development, models play a major role and serve as the main design artifacts that developers aim at achieving. Due to the difficulties in developing high quality models developers aim at starting from an established point in which an existing model can be used for the planned application/system [2]. To facilitate that approach, repositories of models start emerging including for example [21, 8]. There are many challenges in establishing such repositories [7]. These include technical issues in terms of model representation, scalability, and heterogeneity and semantic issues of how to analyze and query such repositories. In this paper, we aim at addressing the challenge of querying model repositories, and in particular class models. Indeed, existing repositories provide searching capabilities, yet, these mainly refer to meta data (e.g., size and type) or to textual and keyword similarity. However, similarity between model artifacts requires consideration of both or at least semantic similarity and structural similarity [26]. In recent work the graph structure is also taken into account when searching model repositories [11]. Nevertheless, the capability provided neglect the semantic similarity. One

of the main benefits of measuring semantics similarity between software artifacts (e.g., code, State Chart, Class Diagram...) is the ability to reuse them so to save time during the software development process [10, 14]. In this paper, we aim at addressing this gap by proposing an algorithm that searches for class models that match the developers intention. The algorithm is based on a greedy search approach where in each iteration the algorithm tries to find the most similar class to the next class of the query using similarity function that considers type similarity, structure similarity and label similarity. We also evaluated the performance of the algorithm by various information retrieval metrics and found the result promising.

The paper is organized as follows. Section 2 analyze related studies for searching and measuring similarities between models. Section 3 introduces the sought algorithm and Section 4 presents its evaluation. Finally, Section 5 concludes and sets plans for future research.

## 2 Related Work

Several approaches have addressed the challenge we introduced. In this section we review related studies with respect to the following:

- **Type of similarity** refers to the way similarity is being calculated.
- **Query form** refers to the considered query form (e.g., text, model, type).
- **Experimental settings and datasets** refer to the way the suggested solutions were evaluated.
- **Examined metrics** refers to the way similarity and related algorithms are being considered.
- **Performance** refers to the values of the mentioned metrics.

Robles et al. suggest a new approach for representing knowledge of UML Class Diagrams (UCD) based on Information Retrieval (IR) techniques and calculating their semantic similarity using ontology [22] graphs. The type of similarity is calculated by a formula that considers semantic similarity between the concepts of the diagrams and the topological similarity between the types of elements in the diagrams. The queries are represented as UCD. To evaluate the proposed method, they used fifteen queries of different levels of complexity and performed the search on a corpus that contains sixty-five UCDs. To demonstrate the importance of using ontology similarity they performed their experiment twice, with and without ontology that is similar to the Boolean model in the IR area. The F-score indicated that using ontology leads to significantly better results of 0.83 compared to 0.36 on average.

Similar to [22], Al-Rhman et al. suggest different type of similarity equations, for example: lexical name similarity, attribute similarity, and operation similarity [4]. Al-Rhman et al. show how weighing each of the elements in the equation affects the total similarity between UCDs. They further suggest a greedy-based matching algorithm that finds the similarity of classes of two models in polynomial time [3]. They proposed several different metrics to measure similarity

between UCDs. They examined which metric is the most accurate for UCDs in the same domain or from different domains. The first case contains five diagrams grouped up into ten pairs in the same domain. The results indicate that the lexical name similarity that measures the semantic similarity between classes gave the best results in terms of precision, recall, and accuracy. The second case contains four UCD formed into six pairs. In this case, the neighboring similarity that measures the similarity of two classes is superior in terms of precision, recall, and accuracy. Nikpforova et al. [16] suggest an algorithm that converts each class diagram to a vector and then refers to the similarity between them based on distance calculation. The experiment consists of three small similar UCDs.

Even though these methods show promising results, they rely on the fact that similar UCDs have to be in the same context. To overcome this problem, several works have been proposed to incorporate structural similarity. For UML Sequence Diagram, for example, Salami et al. [24] proposed a genetic algorithm that compares two sequence diagrams based on adjacency matrices. They calculate the Mean Average Precision (MAP) of 10 queries to determine how well their method performs for retrieving similar UML Sequence diagrams. The MAP was approximately 0.75 in all ten different repositories that they searched in.

Yuan et al. [26] categorized UML Class Diagrams into two aspects: intra-structure and inter-structure. Intra-structure represents the inner structure of classes like attributes and operations, whereas inter-structure represents the relationships between classes like association and composition. Based on this categorization, they converted UCDs into UML Class Graphs (UCG) and measured the similarity of the inter and intra-structure of the graphs. The results were promising based on the fact that the similarity of pairs of graphs was similar to the expert's results. Despite that, they only applied the approach to two domains.

López et al. [11] propose a search engine called MAR for retrieving any EMF model [25] focusing on converting a structure of models into an inverted index [5]. The queries take the form of a model that describes the desired results. They convert each model and query to a bag of paths and use the Okapi BM25 measure [20] for ranking the most relevant models. The similarity function considers the number of times a path appears in both query and model graph and whether those paths are unique.

Another option to consider is the similarity flooding algorithm [13] that aims at finding pairs of elements that are similar in two data schemes or two data instances. Yet, in this work we are interested in paths considering also the labels of the vertices and edges. Nevertheless, such algorithm, can be used as another node similarity function.

In Table 1 we summarized the analysis of the methods we mentioned, along the facets we specified before. The methods measure the similarity between queries and models and assume that similar UCDs have much in common. This assumption may lead to poor performance in cases where users are not fully knowledgeable in the domain that they search for. In this study, we aim at addressing this problem by searching for similar sub-diagrams in UCD and measure

the similarity between queries and sub-diagrams of the diagrams. We consider the semantic similarity between labels and attributes of the classes, the type similarity of classes and relationships (for example 'Class' and 'Interface'), and consider the structural path between UCDs while other approaches consider only parts of these similarities.

Table 1: Related Work Summary

| Work | Query form | Models | Type of similarity | Evaluation Data | Search method |
|---|---|---|---|---|---|
| [22] | UCD | UCD | Semantic + type | 65 UCDs + 15 queries | Inverted index |
| [4] | UCD | UCD | Semantic + type | 9 UCDs | Different metrics |
| [3] | UCD | UCD | Semantic + type | 9 UCDs | Greedy-based matching algorithm |
| [16] | UCD | UCD | Semantic + type | 3 UCDs | Similarity function |
| [24] | USD | USD | Type+structural | 60 USD + 10 queries | Genetic algorithm |
| [26] | UCD | UCD | Structural | 60 UCDs + 10 queries | Similarity function |
| [11] | Any | Correlated to the model | Structural | 17975 Ecore meta models[1] | Inverted index |

USD=UML Sequence Diagram; UCD=UML Class Diagram;

## 3 Greedy Search in UCD

The problem we are aiming to address is the search for relevant UCD fragments within UCDs given a specific query in the form of class diagram. For that purpose we devised a greedy search algorithm in UCDs that addresses the concerns of semantics. The algorithm adopts the similarity considerations appears in [18] and refers to label matching (exact match among labels), structure matching (in terms of links), semantic matching (in terms of labels semantic similarity and links' semantics), and type matching. In the following we first set the ground for the algorithm and then elaborate on its design and execution.

### 3.1 Settings

**Definition 1.** *A **UML Class Diagram (UCD)** is a diagram (D) that consists of classes (C), attributes (A), and relationships (R). D=(C,A,R).*

- *C is a set of classes.*
- *A is set of attributes* $A = \{A_1, A_2, ..., A_k\}$ *where* $A_i$ *is a set of attributes of class* $c_i$*. Each* $c_i$ *has a* $A_i = \{a_1, a_2, ..., a_j\}$*.*
- *R is a set of relationships. Each D has a* $R = \{r_1, r_2, ..., r_k\}$*. Where* $r_i = \{(c_i, l, c_j) | c_i, c_j \in C, l \in ET\}$*.* $ET = \{Association, Composite, Generalization, Aggregation, and\ InterfaceRealization\}$

**Definition 2.** $sim_w$ *refers to the similarity of two words. This can be calculated using for example WordNet [15], Sematch [27], or google w2v [17].*

**Definition 3.** $sim_l$ *refers the similarity of two classes labels.*

$$sim_l(list_1, list_2) = \frac{\Sigma_{w_1 \epsilon list_1}(\arg max_{w_2 \epsilon list_2}(sim_w(w_1, w_2)))}{\arg\max(length(list_1), length(list_2))} \quad (1)$$

*the lists are ordered set of words (of the labels of the classes).*

**Definition 4.** *$sim_{att}$ refers to the similarity of two sets of attributes.*

$$sim_{att}(att_1, att_2) = \sum_{list_1 \epsilon att_1} \frac{\arg max_{list_2 \in att_2}(sim_l(list_1, list_2))}{\arg max(length(att_1), length(att_2))} \quad (2)$$

**Definition 5.** *$sim_t$ refers to a predefined similarity among classes types. Table 2 presents the similarity among UCD class types. The similarity matrix shown in Table 2 is based on similarities suggested in [22].*

Table 2: Class type similarity.

| | Class | Abstract | Interface | Association Class |
|---|---|---|---|---|
| Class | 1 | 0.75 | 0.4 | 0.8 |
| Abstract | 0.75 | 1 | 0.75 | 0.7 |
| Interface | 0.4 | 0.7 | 1 | 0.2 |
| Association Class | 0.6 | 0.7 | 0.0 | 1 |

**Definition 6.** *$sim_c$ refers to the similarity of two classes.*

$$sim_c(c_1, c_2) = \frac{\text{sim}_l(c_1, c_2) * 0.75 \ + \text{sim}_{att}(att_{c_1}, att_{c_2}) * 0.25 + sim_t(c_1, \ c_2)}{2} \quad (3)$$

The search process should consider both the semantic similarity of the elements as well as their types. Thus, we weight these equally. For the semantic similarity of classes we consider the class label of higher importance with respect to its attributes as these are of secondary relevance and besides they may contain "noise" like generic attributes (e.g., id and status). Therefore we weighted $sim_l$ higher then $sim_{att}$.

**Definition 7.** *$sim_r$ refers to a predefined similarity among relationship types. Table 3 presents the similarity among UCD relationships. The similarity matrix shown in Table 3 is based on similarities suggested in [22].*

Table 3: Relationship type similarity

| | ASS | CO | AG | GE | INR |
|---|---|---|---|---|---|
| ASS | 1 | 0.89 | 0.89 | 0.55 | 0.2 |
| CO | 0.89 | 1 | 0.89 | 0.55 | 0.23 |
| AG | 0.89 | 0.89 | 1 | 0.55 | 0.23 |
| GE | 0.51 | 0.51 | 0.51 | 1 | 0.4 |
| INR | 0 | 0 | 0 | 0.21 | 1 |

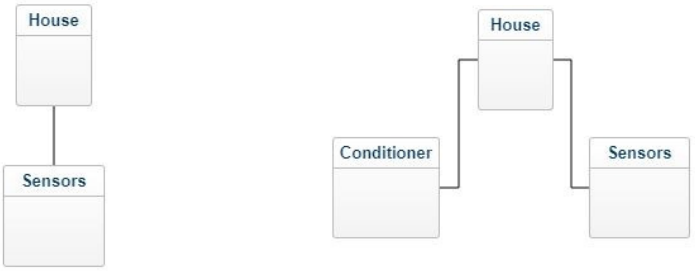
ASS=Association; CO=Composite; AG=Aggregation ; GE=Generalization; INR=Interface Realization

**Definition 8.** *class-relationship similarity is calculated as follow:*

$$sim_{ne}(c_1, r_1, c_2, r_2) = \frac{2 * \text{sim}_c(c_1, c_2) \ + sim_r(r_1, \ r_2)}{3} \quad (4)$$

This definition captures the similarity of a class along with it incoming relationship.

**Definition 9.** *A query Q is represented as a UCD.* ***Single path*** *query has a single path of classes and relationships (see Figure 1a).* ***Multi paths*** *query has multiple paths (see Figure 1b) without cycles.*




(a) Single path

(b) Multi paths

Fig. 1: Single path and Multi paths queries

**Definition 10.** *Similarity of paths of the query with those of the diagram is calculated as follow (classRelationshipSim is a set of $sim_{ne}$):*

$$Path - Sim\left(D^1, D^2, classRelationshipSim\right) = sim_c\left(c_1^{D^1}, c_1^{D^2}\right) + \sum_{i=1}^{|classRelationshipSim|} classRelationshipSim_i$$

In the following we elaborate the algorithm for single path and multi-path queries. We differentiate between two types of class similarity. The first relies solely on the class information, we call this class-based similarity. The second also refers to the relationship connecting the class from the previous one, we call this class-relationship based similarity.

- The function of **class-based similarity** returns the most similar class in the UCD to a class within the query as calculated by Definition 6.
  Input: $D = (C, A, R)$, query diagram, similarity function. The similarity function is the one appears in Definition 2.
  Output: a class from D.
- The function of **class-relationship based similarity** is responsible for retrieving up to K most similar neighbors' classes that exceed a predefined threshold as determined by Definition 8. If there is no single class that its similarity exceed the threshold, the function returns all the neighbor's classes.
  Input: $D = (C, A, R)$, query class, parent class, similarity function, threshold, K.
  Output: list of classes and their similarities from D.

### 3.2 The Greedy Search Algorithm

To start the execution of the Greedy Search (GS) algorithm, the number of classes ($K$) the algorithm handles in each round (that refers to the next step of the query) and the threshold for the similarity need to be determined. The algorithm uses the class-relationship based similarity function to find the appropriate list of classes for its execution in each iteration. We also have to determine the UCD(s) in which the search should take place and *similarClasses* where the search should start (this task can be achieved by the class-based similarity function). For the query, we have the *queryDiagram* that need to be found in the set of $UCD$s. Additional required parameters include the $queryID$ that refers to the id of the class in a *queryDiagram* and is initialized with '-1' that represents the first class of the query where we want to start search from. $isVisited$ is a list that contains all the classes that GS algorithm already iterates over in the $UCD$ to allow GS algorithm handles diagrams with cycles. The output of the algorithm is a set of sub-diagrams from the $UCD$ that are the most similar to the *queryDiagram*. Algorithm 1 describes the entire procedure.

**Algorithm 1** GS in UCD

**Input** $similarClasses, UCD(s), queryDiagram, K, threshold,$
$queryID, isVisited$
**Output** Set of sub diagrams with their similarities

```
results ← {θ}
if similarClasses ≡ {θ} || v_queryID ∉ V_queryDiagram then
    return results
end if
for class in similarClasses do
    if class ∈ isVisited then
        continue
    end if
    isVisited ← isVisited ∪ class

     similarClasses ← class − relationship_based_function(
UCD(s), queryDiagram[queryID], class, K, threshold)
    if similarClasses ≡ {θ} || arg max_{sim∈similarClasses}(sim) ≤ threshold then
results' ← GS(similarClasses, UCD(s), queryDiagram, K,
th, queryID, isVisited)
    else
        results' ← GS(similarClasses, UCD(s), queryDiagram, K
, threshold, queryID + +, isVisited)
    end if
    results ← results' ∪ results
end for
return results
```

The GS algorithm steps are as following: (line 1) Initialize results with an empty set; (lines 2-4) Stop GS if there are no classes in the *similarClasses* or there are no classes with a given ID in the *queryDiagram*; (line 5) iterate over all neighbors of the current class; (lines 6-8) skip already visited classes; (line 9) For each class in *similarClasses* we add it to $isVisited$ list; (line 10) Get up to K most *similarClasses* by the class-relationship based function described above; (lines 11-12) If there are no classes in *similarClasses* or the similarity of the classes in the *similarClasses* is below the threshold the algorithm will run recursively for the selected *similarClasses* and $queryID$, without moving on the next class; (lines 12-14) If there are classes in *similarClasses* and the similarity of the classes in the *similarClasses* are above the threshold - the algorithm will run recursively for the selected *similarClasses* (up to K classes) and $queryID$, each time moving on to the next class – in other words, in order to allow the return of results that are transitive, the class in the query is promoted only when the similarity obtained for the class is higher than the threshold; (line 15) Merge the results that the algorithm retrieved with the results within the function arguments. The algorithm continues until it passes all the classes and relationships of the query. At the end of the algorithm, sub diagrams from the UCD that answer the query will be obtained. The sub diagrams will be sorted by their similarity ranking, as determined by Definition 10. The ranking follows the weighted similarity of the relationships and classes and is represented by a number between 0 and 1.

### 3.3 GS Execution Example

In the following we demonstrate the execution of GS algorithm on a UCD of a machine architecture shown in Figure 2 with the query shown in Figure 3a. We set the threshold to 0.75 and the number of lookup classes (K) to 2.

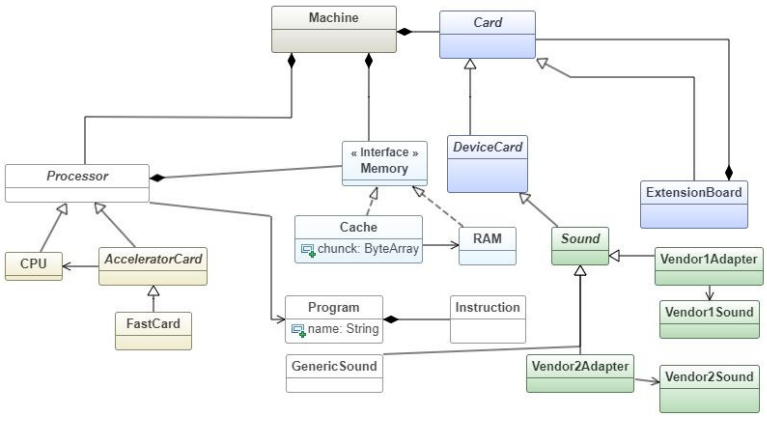



Fig. 2: The Machine Architecture UCD


(a) Query diagram

(b) Expected output

Fig. 3: Query and Expected output

1. In the first stage, the class-based similarity function for finding the most similar class in the UCD is used to find the first class of the query diagram. The following result demonstrates the output for the class "Machine" in the query diagram:

```
[-4]: ['Cache'] :: ['Computer']
similarity: 0.624
[-5]: ['RAM'] :: ['Computer']
similarity: 0.634
[-6]: ['Machine'] :: ['Computer']
similarity: 0.809
```

   (for all other classes, the similarity was below 0.3)

2. Following the highest similarity of 0.747, the GS algorithm starts from the class "Machine". In this stage, the algorithm keeps trying to find the K (K=2 in our case) most similar adjacent classes to the selected class in the UCD to the next class of the query diagram, that is, "Sounds". The results are the following:

```
Start finding 2 most similar classes in the UCD to the class of
a query:['Sounds']
The neighbors of vertex: -6 are: [-9, -2, -11]
[-9]: ['Memory'] :: ['Sounds']
similarity: 0.541
[-2]: ['Processor'] :: ['Sounds']
similarity: 0.583
[-11]: ['Card'] :: ['Sounds']
similarity: 0.624

Here, as no class the exceed the threshold we continue with all
classes: [(0.541, -9), (0.583, -2), (0.624, -11)]
```

3. This stage splits into three paths of which the algorithm iterates over neighbors of the relevant classes ['-2'], ['-9'] and ['-11']. Here, we only show the results after iterating over class ['-11']:

```
Start finding 2 most similar classes in the UCD to the class of
a query: ['Sounds']
```

```
The neighbors of vertex: -11 are: [-10, -18]
[-18]: ['Extension', 'Board'] :: ['Sounds']
similarity: 0.574
[-10]: ['Device', 'Card'] :: ['Sounds']
similarity: 0.497
[-18]: ['Extension', 'Board'] :: ['Sounds']
similarity: 0.574
The most similar node is: [(0.497, -10), (0.574, -18)]
```

Here again, since all the classes $sim_{ne}$ value (Definition 8) is below the threshold, all the classes advance recursively to the next step without advancing to the next query class.

4. The GS algorithm terminates only when one of the following scenarios occur:
   (a) The GS algorithm finished iterating over the classes in the *queryDiagram* therefore, GS algorithm found one sub-diagram among all the potential sub-diagrams:
   ```
   Start finding 2 most similar classes in the UCD to the class of
   a query: ['Sounds']
   The neighbors of vertex: -10 are: [-11, -12]
   [-11]: ['Card'] :: ['Sounds']
   similarity: 0.497
   [-12]: ['Sound'] :: ['Sounds']
   similarity: 0.670
   The most similar node is: [(0.670, -12)]
   ```
   (b) There are no more classes to iterate over in the UCD. This happens due two reasons: (1) There are no neighbors; (2) The neighbors already visited by the algorithm. Therefore, all the sub-diagrams that end with class '-18' are not relevant answers:
   ```
   Start finding 2 most similar classes in the UCD to the class of
   a query: ['Sounds']
   The neighbors of vertex: -18 are: []
   The most similar node is: []
   ```

5. Finally, the results paths are calculated for their similarity based on Definition 10 and are sorted accordingly.

   ```
   [('->Machine->Memory->Cache->RAM', 0.606),
   ('->Machine->Card->Device Card->Sound', 0.650),
   ('->Machine->Memory->Processor->Program', 0.666)]
   ```

   Indeed, the expected outcome that appears in Figure 3b is included within the result set of the GS algorithm. The final similarity of the expected path is calculated by definition 10 as follow: $\frac{0.809+0.624+0.497+0.670}{4} = 0.650$.

## 4 Evaluation

To evaluate the GS algorithm, we conducted an experiment to check the algorithm performance in various settings.

### 4.1 Settings

We considered nine domains from which we had UCDs: (1) Bank Management; (2) Library; (3) Machine; (4) Pizza store; (5) Restaurant (6) University; (7) Climate; (8) Tank; and (9) Store. Diagrams (1)-(6) were adjusted from GenMyModel[2], (7) and (8) were adjusted from [19], (9) was adjusted from [12]. The domains were chosen in order to check whether the algorithm can distinguish between domains that have similar semantic like Pizza store (5) and Store (9), and similar structure like Pizza store (5) and Library (3). Each UCD contains in average 9.44 classes and 9.55 relationships. Each class has in average 1.37 attributes and 2.04 neighbors. The label that might be a class name or an attributes name has 1.12 words in average.

We were also interested in exploring how the query complexity affects the algorithm performance. For that reason we defined three complexity categories based on two parameters:

1. **Path:** the number of paths from the first class.
2. **N-hop:** supporting hop relationship in the map according to the expected result. That is, the number of skips on the diagram relationships required to get the desired result.

The categories were the following:

1. **Simple** the query has one path and n-hop equals to 0
2. **Medium** the query has one path and n-hop is more then 0
3. **Complex** the query has more then 1 path

We set 20 queries that were classified into three categories of complexity: simple - 7 queries, medium - 7 queries, and complex - 6 queries.

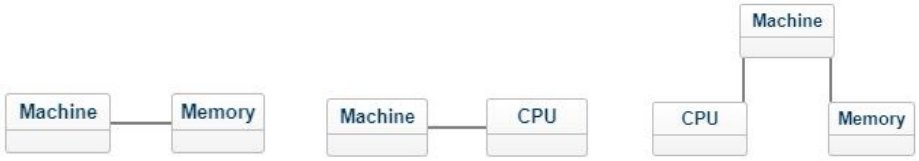
Examples of a simple, medium, and complex queries are shown in Figure 4. The names of the classes of the queries are matched to the class names in Figure 2 to simplify the demonstration of how the parameters influence the difficulty of the categories. Figure 4a is a simple query because the expected result is $Machine- > Memory$ therefore, the *Path* and *N-hope* results in zero. Figure 4b is a medium query because the expected output is $Machine- > Processor- > CPU$ and as a result *N-hope*'s value is 1 and *Path*'s value remains 0. Finally Figure 4c is a complex query due to the number of *Paths* in the expected output resulting in two, $Machine- > Memory\ and\ Machine- > Processor- > CPU$.

The experiment material and results can be found in an online appendix [6]. To evaluate the performance of the GS algorithm we used the following metrics:

– **Domain Mean Reciprocal rank (Domain MRR)** indicates whether the right model was retrieved:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{5}$$

[2] https://www.genmymodel.com/




(a) Simple (b) Medium (c) Complex

Fig. 4: Three complexity query level

where $Q$ is the set of the queries, and $rank_i$ is the rank position of the first relevant domain result to the $i$-th query.

- **Diagram Similarity (D-sim)** that measures the similarity between a query diagram and a sub-diagram from the set of UCDs as defined in Definition 10. The D-sim takes into account the highest D-sim score of each query and ignores cases in which no answer was found.
- **Recall@k** that determines whether the expected result appears within the top k results.
- The **number of failures** in which the search did not succeed.

We executed the GS algorithm with K=2 and searched for a threshold that maximized the **Domain MRR**. Figure 5 shows how the **Domain MRR** metric is affected by the threshold. Following the examination, it seems that a value of 0.65 leads to the best results. Therefore, we set the threshold to 0.65 and measured the metrics for each complexity category.

### 4.2 Results

Table 4 presents the results of the experiment. The numbers in each cell indicate the average result after running each of the relevant set of queries. Examining the Recall@1, Recall@3, Recall@5, Domain MRR and similarity metrics. The results indicate that the algorithm's performance decreases as the query complexity increases.

Table 4: Results

| Complexity | Recall@1 | Recall@3 | Recall@5 | Domain MRR | D-sim | Search failure | TOT. Answers |
|---|---|---|---|---|---|---|---|
| Easy | 0.571 | **0.857** | 0.857 | 0.801 | 0.783 | 1 | 7 |
| Medium | 0.571 | **0.714** | 0.714 | 0.785 | 0.675 | 2 | 7 |
| Complex | 0.333 | 0.333 | 0.416 | **0.509** | 0.813 | 7 | 12 |
| Mean | 0.490 | 0.634 | 0.662 | **0.698** | 0.757 | 3.33 | - |
| Std | 0.113 | 0.221 | 0.183 | 0.134 | 0.059 | 2.62 | - |

### 4.3 Discussion

In this section, we discuss the results with respect to the parameters of the GS algorithm and the input characteristics.

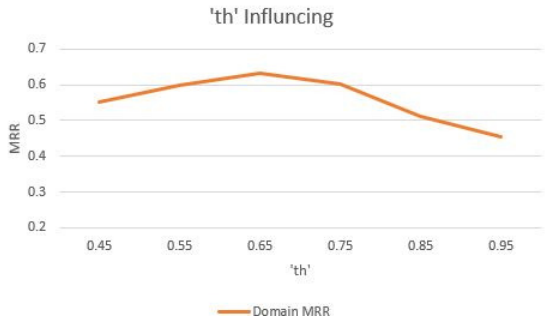



Fig. 5: 'th' influencing on Domain MRR

In the experiment we set the number of neighbours (K) to 2. It might be beneficial to increase K in correlation with the actual neighbours of each class (In our case, it was 2.04 in average). This might increase the accuracy of the search and allow the exploration of additional paths.

The threshold parameter also affects the results as it determines the relevant classes from which the algorithm moves forward. It might be that the threshold should be changed based on the domain and the $sim_w$ function. The semantic of two classes ($sim_l$) depends on two parameters: similarity between two words ($sim_w$) and the length of a class's label in terms of how many words the vertex contains. For $sim_l$ and $sim_{att}$ we noticed that it is affected by the number of words. In the case of long labels the similarity decreases.

### 4.4 Threats to Validity

The initial evaluation we performed should be taken with caution and should consider the following threats to validity:

- **Small UCD**: The diagrams in the experiment are of limited size (9.44 classes and 9.55 relationships in average). We should explore the algorithm with much larger diagrams.
- **Self authoring**: We as the authors of the paper, developed the queries, so some biases might exist. Nevertheless, our aim in this evaluation was to challenge the algorithm, so the queries we devised aimed to accomplish that.
- **Simple queries**: Both the domains and the queries are quite small. There is a need to incorporate more complex domains and queries.
- **Lack of tuning**: Based on informal experience, we set the algorithm parameters as constants. Yet, there is a need to perform sensitivity analysis to learn about the impact of these parameters.
- **Comparing to other works**: Indeed, the results should be compared to other alternatives. Yet, such alternatives need to be adjusted for our UCDs repository. For example, converting UCDs into corresponding ontology graph and then run an ontology matching algorithm.

## 5 Summary

In this paper, we propose a greedy algorithm for searching UCDs. The algorithm takes into structure, semantic, and type similarity compering to other works shown in Table 1. The initial evaluation we preformed shows a promising results in terms of accuracy.

Nevertheless, in the future, we intend to improve the $sim_l$ function using text mining techniques like RNN [23] and LSTM [9] for sentence similarity task. Moreover, we want to test how well the algorithm performs with other domains and examine alternatives for calculating the similarity of the classes and relationships. This includes the tuning of the algorithm's parameters, either a-priory or during its execution.

Although we evaluate the algorithm on UCDs, we believe it can be used for searching other models. Thus, we also plan to test our conjecture regarding this matter.

## References

1. S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Stöerrle, and J. Whittle. User experience for model-driven engineering: Challenges and future directions. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 229–236, 2017.
2. Henning Agt-Rickauer, Ralf-Detlef Kutsche, and Harald Sack. DomoreFIX ME!!!!? a recommender system for domain modeling. In *the 6th International Conference on Model-Driven Engineering and Software Development*, page 71–82, 2018.
3. Mojeeb Al-Rhman Al-Khiaty and Moataz Ahmed. Similarity assessment of uml class diagrams using a greedy algorithm. In *2014 International Computer Science and Engineering Conference (ICSEC)*, pages 228–233. IEEE, 2014.
4. Mojeeb Al-Rhman Al-Khiaty and Moataz Ahmed. Uml class diagrams: Similarity aspects and matching. *Lecture Notes on Software Engineering*, 4(1):41, 2016.
5. Arvind Arasu, Junghoo Cho, Hector Garcia-Molina, Andreas Paepcke, and Sriram Raghavan. Searching the web. *ACM Transactions on Internet Technology (TOIT)*, 1(1):2–43, 2001.
6. Maxim Bargilovski, Yifat Makias, Moran Shamshila, Roni Stern, and Arnon Sturm. Searching Models. http://dx.doi.org/10.17632/6685g76r9y.1, March 2021.
7. Francesco Basciani, Juri Di Rocco, Davide Di Ruscio, Ludovico Iovino, and Alfonso Pierantonio. Model repositories: Will they become reality? In *the 3rd International Workshop on Model-Driven Engineering*, pages 37–42, 2015.
8. Regina Hebig, Truong Ho Quang, Michel R. V. Chaudron, Gregorio Robles, and Miguel Angel Fernandez. The quest for open source projects that use uml: Mining github. In *the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, page 173–183, 2016.
9. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
10. Christina Lau. Reusing code in object-oriented program development, January 30 2001. US Patent 6,182,274.
11. José Antonio Hernández López and Jesús Sánchez Cuadrado. Mar: A structure-based search engine for models. In *the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 57–67, 2020.

12. Francisco Martínez and Ambrosio Toval. A precise approach for the analysis of the uml models consistency. pages 74–84, 10 2005.
13. Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In *Proceedings 18th International Conf. on Data Eng.*, pages 117–128. IEEE, 2002.
14. Hafedh Mili, Fatma Mili, and Ali Mili. Reusing software: Issues and research directions. *IEEE transactions on Software Engineering*, 21(6):528–562, 1995.
15. George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.
16. Oksana Nikiforova, Konstantins Gusarovs, Ludmila Kozacenko, Dace Ahilcenoka, and Dainis Ungurs. An approach to compare uml class diagrams based on semantical features of their elements. In *the Tenth International Conference on Software Engineering Advances*, pages 147–152, 2015.
17. Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA.
18. Iris Reinhartz-Berger. Towards automatization of domain modeling. *Data and Knowledge Engineering*, 69(5):491 – 515, 2010.
19. Iris Reinhartz-Berger and Arnon Sturm. Utilizing domain models for application design and validation. *Inf. and Software Technology*, 51(8):1275–1289, 2009.
20. Stephen Robertson and Hugo Zaragoza. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc, 2009.
21. Gregorio Robles, Truong Ho-Quang, Regina Hebig, Michel R. V. Chaudron, and Miguel Angel Fernandez. An extensive dataset of uml models in github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, page 519–522. IEEE Press, 2017.
22. Karina Robles, Anabel Fraga, Jorge Morato, and Juan Llorens. Towards an ontology-based retrieval of uml class diagrams. *Inf. and Software Technology*, 54(1):72–86, 2012.
23. David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.
24. Hamza Onoruoiza Salami and Moataz Ahmed. Retrieving sequence diagrams using genetic algorithm. In *2014 11th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 324–330. IEEE, 2014.
25. Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.
26. Zhongchen Yuan, Li Yan, and Zongmin Ma. Structural similarity measure between uml class diagrams based on ucg. *Requirements Engineering*, pages 1–17, 2019.
27. Ganggao Zhu and Carlos A Iglesias. Sematch: Semantic similarity framework for knowledge graphs. *Knowledge-Based Systems*, 130:30–32, 2017.