Graph Search of Software Models Using Multidimensional Scaling

Bojana Bislimovska[1], Güneş Aluç[2], M. Tamer Özsu[2] and Piero Fraternali[1]

[1] *Politecnico di Milano,* [2] *University of Waterloo*

{bojana.bislimovska, piero.fraternali}@polimi.it {galuc,tamer.ozsu}@uwaterloo.ca

ABSTRACT

Software models formalize the requirements, structure and behavior of a system or application. They represent essential artifacts that simplify the process of software development. Software repositories have been developed to store models in order to facilitate the reuse of know-how from software projects; however, methods for searching these model repositories are not very efficient. Specifically, while being more scalable, general-purpose keyword search is not suitable for model search because it does not consider the structure that is inherent in software models: a good search algorithm should consider the model structure as well as the knowledge concentrated in the metamodel. On the other hand, existing approaches that consider the structure while querying software models are limited to only specific domains such as Business Process Models (BPMs).

In this paper, we introduce MultiModGraph, an efficient approach for indexing and searching model repositories. MultiModGraph preserves the model structure and metamodel information by representing models as graphs. To enable efficient search, the approach employs multidimensional scaling to approximately map vertices of the model graph to points in space. We evaluate MultiModGraph both with respect to speed and quality of results using a real-word repository of web application models.

1. INTRODUCTION

Models facilitate software development in multiple ways: They raise the level of abstraction to help deal with the increasing complexity in software development; they help organizations improve source code quality and adapt faster to changes in the requirements of a project; and they improve communications within an organization. Models have a specific structure, which is expressed using a well-defined syntax of a modeling language. Each modeling language conforms to a metamodel, which defines the structure, semantics and constraints for building a model [10].

Model repositories are used for storing collections of software models. Most (model) repositories offer elementary tools to search these collections of models, which is particularly important for model re-usability [12]. For example, instead of designing a model from scratch, developers can retrieve an already existing modeling pattern (from the repository) and tailor it according to their needs to build new software models. This can significantly improve the model development process by decreasing its time and cost, while at the same time improving its quality.

Broadly speaking, model repositories employ two techniques for search: general-purpose keyword search [13], or content-based search that incorporates the model structure in the query [16, 19]. However, each technique has its shortcomings. While being more scalable, general-purpose keyword search is not suitable for model search because it does not consider the structure that is inherent in software models [2]. Furthermore, most keyword-based approaches allow only exact matching of keywords, where a set of keywords is matched against the models' description (e.g. model element labels). On the other hand, those approaches that consider the structure while querying software models are limited to only specific domains such as Business Process Models (BPMs). In contrast, methods are needed that are (i) more general, (ii) sensitive to the knowledge about the model structure and (iii) are at the same time scalable. Specifically, these solutions should allow users to pose queries (to the model repository) in the form of a model sketch, which captures the intended requirements in a native modeling language supported by the model repository. Then, the repository should rank and return a sub-collection of the "most relevant" modeling patterns from these models.

In this paper, we propose an algorithm for efficient search of Web Modeling Language (WebML[1]) models, namely, MultiModGraph. The algorithm uses a representation of models as attributed graphs, which allows mapping of the model structure and hierarchies among the model elements to a graph. Queries, which represent model fragments, can also be transformed into graphs, and they can be used for searching similar models in a model repository. Our algorithm uses multidimensional scaling to represent the graph vertices as points in a multidimensional space. These points are used to build an index that allows for efficient pruning during search. This way, given a query vertex, those vertices in the graph that are relevant to the query can be located efficiently (i.e., they correspond to points within a specified distance in the

[1]WebML is a modeling language for Web application front-ends, recently generalized into the OMG IFML standard (www.ifml.org)

multidimensional space). Furthermore, the algorithm considers neighborhood information for each graph vertex in order to locally expand already matched vertices. Metamodel information is incorporated in the search and indexing, as well as in the ranking function which sorts retrieved models with respect to their similarity.

The paper is organized as follows: Section 2 presents related work; Section 3 describes the WebML modeling language and the process of model to graph transformation; Section 4 gives the system architecture; Section 5 describes the proposed approach; Section 6 illustrates our results, and finally, Section 7 concludes and proposes directions for future work.

2. RELATED WORK

Existing works can be classified into 3 areas: (i) keyword-based model search, (ii) content-based model search, which present specific techniques for search of models, and (iii) graph databases whose indexing and querying approaches can be employed for model search.

2.1 Keyword-Based Model Search

Keyword-based approaches for model search use a set of keywords for querying models. Their main limitation is that they do not consider the model structure in the query or the hierarchies and relationships among model elements. Furthermore, they return exact but not approximate matches to the query, which may be relavant to the user.

Moogle [13] is a keyword-based model search engine that uses metamodels to create indexes for the evaluation of keyword queries. In comparison to our approach, Moogle supports only textual queries with just a simple filter on the type of the model element to be returned. Another keyword based search solution for WebML models is presented in [2]. It incorporates metamodel information in the search process, used only in deciding how weights are assigned to different index terms. In contrast, MultiModGraph supports relationships among model elements through graph model representation, as well as some additional metamodel information such as references to the Data Model.

2.2 Graph-Based Model Search

Existing approaches that rely on a graph-based representation of models predominantly target Business Process Models (BPM) and their corresponding notations. However, BPMs are not as rich as WebML models in terms of syntax and semantics. Moreover, they are not suitable for searching large collections of models, since they only rely on a scan of the set of models without any indexes. One example of such technique is [6], which proposes discovering and ranking of BPEL process models. This is achieved by using behavioral similarity measure and a graph matching algorithm.

The approach in [16] retrieves process models by combining related pairs' clustering and a set of metrics for comparison of vertex labels. The main limitation of this approach is that the similarity between two process models is mainly based on the similarity of vertex labels rather than the structural similarity of the model graphs.

Some recent approaches [19, 8] exploit indexing for more efficient retrieval of business process models. These indexes are mostly feature-based, containing subgraphs that are most representative features of the model graphs in the repository. However, this type of indexing cannot be applied to models with complex metamodels, such as WebML.

In [2], we compare keyword-based approach with a graph-based approach for searching web application(WebML) models, but we do not apply any indexing in the graph-based approach.

2.3 Graph Databases

The approaches for indexing and querying that allow efficient search of large graph databases can be employed for efficient search of models. NeMa (Network Match) [9] is a neighborhood-based top-k subgraph matching technique that uses a minimal cost function to evaluate a goodness of a match. It considers structure and label similarities and it uses a neighborhood-based vector index to improve efficiency. Unlike NeMa, MultiModGraph uses different kind of indexing based on multidimensional scaling. TALE (Tool for Approximate Subgraph Matching of Large Queries Efficiently) [18] is a general tool for approximate subgraph matching. It employs neighborhood-based indexing. TALE allows for vertex mismatches and vertex and edge gaps. The basic differences are that in MultiModGraph the queries are small model fragments, and the graphs are attributed.

There also exist some approaches for search of attributed graph databases [11, 20] whose graphs contain multiple labels for both vertices and edges. Attributed graphs are used to represent WebML models, because they exploit the richness of the WebML metamodel. These techniques for matching attributed graphs use indexing methods that contain neighborhood information for each vertex. MultiModGraph also uses neighborhood information, but for a different purpose, i.e., to expand the matching candidates. The main limitation of these approaches is that they do not consider approximate, but only exact graph matching.

3. BACKGROUND AND PRELIMINARIES

In this section, we give a brief introduction to WebML, and describe the transformation from WebML models to attributed graphs.

3.1 Web Modeling Language (WebML)

WebML is a Domain Specific Language (DSL) for designing complex web sites [4], recently generalized into OMG IFML standard. It consists of two parts (i) data model and (ii) web model. The data model describes the data requirements of an application, using entity-relationship notation. The web model describes the organization of the front-end interfaces of a web application. It contains three main building blocks, namely pages, units and links which are organized hierarchically into larger container elements such as areas and site views. A *site view* represents a model element that includes a well-defined set of requirements for a specific category of users. Site views can contain *area*s, container elements that cluster pages with a homogeneous subject and can be nested recursively [3]. Pages are the actual interface elements delivered to the user and they contain *content units* which represent atomic elements for specifying the content of a web page. Another type of units is an *operation unit*, contained in the areas and site views. Operation units denote operations on data or arbitrary business actions; they can be activated as a result of a link navigation, performing manipulation with data, or execution of an external service. Content and operation units are connected by links. Links allow sequencing of units, passing parame-

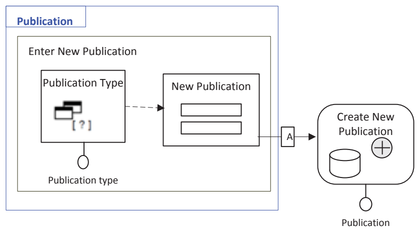


Figure 1: Example of a WebML model

ters, navigating the hypertext front-end, changing the page content or accessing a page.

Each WebML model has a structure determined by the WebML metamodel, and an inherent hierarchy determined by the container WebML model elements. Figure 1 shows an example of a part of a WebML model. The area *Publication* contains a *Enter New Publication* page, which allows the user to insert a new publication through the *New Publication* entry unit. The selection of a publication type is enabled by the selector unit *Publication Type*. After the insertion of the data for a publication, a new publication instance is created, which is performed by the *Create New Publication* create unit.

3.2 Model to Graph Transformation

We represent WebML models as attributed graphs such that every model element is represented as a vertex in the graph, while containment relationships and links among the model elements are represented as graph edges. This type of (graph) representation preserves as much as possible the model structure and the hierarchies present among the model elements. Specifically in this case, each graph vertex is annotated with three attributes: (i) name, (ii) type and (iii) data, (Figure 2), where:

- *Name* represents the textual label of the model element;
- *Type* represents the corresponding model element type, derived fro mthe metamodel;
- *Data* represents the entity or relationship to which the model element refers, in case such reference exists.

Likewise, each edge is annotated with the attribute *type* which refers to the type of the corresponding relationship/link in the model, as represented in Figure 2.

Thus, after the transformation, each WebML model from the repository yields an attributed graph, and the model search becomes the problem of searching over a collection of attributed graphs. Since queries also represent model fragments, they can be transformed in the same way into graphs.

4. SYSTEM ARCHITECTURE

Figure 3 presents the architecture of our graph-based model search system. The *Content Processing* component takes every model from the repository and transforms it into a format suitable for indexing. First, the *Project Analysis* sub-component extracts general informaton from the model such as the model name and id. Then, the *Model to Graph Transformation* sub-component transforms each model into a graph considering its metamodel features, as explained

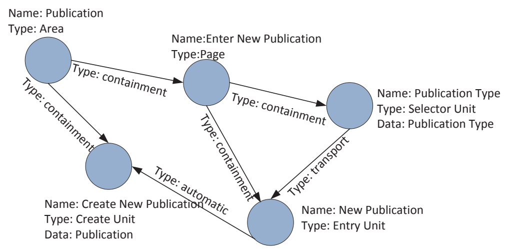


Figure 2: Graph representation of the WebML model in Figure 1

in Section 3 for the case of WebML models. These model graphs are used to build the index in the *Indexing* component, which is elaborated in more detail in Section 5.1.2.

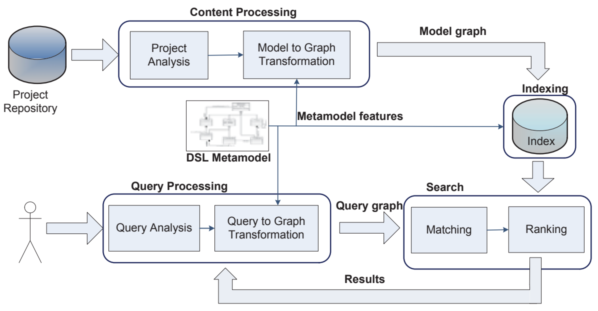


Figure 3: Architecture of a graph-based model-driven information retrieval system.

On the user side, a query is expressed as a model fragment, which can be formulated in the same modeling language in which models in the repository are encoded. The *Query Processing* component transforms the query model fragment into a format that is more suitable for search, the same way models are transformed into graphs by the *Content Processing* component. When the query is transformed into a graph, the system is ready for search.

The *Search* component has two tasks, which are discussed in more detail in Section 5.1.3. The *Matching* sub-component uses a specific algorithm and the help of the index to find model fragments (subgraphs) from the repository that match the query graph under certain criteria. Finally, the *Ranking* sub-component performs sorting on the retrieved model subgraphs with respect to their relevance to the query. These ranked subgraphs (along with their computed ranking scores) are returned to the user.

5. DETAILS OF MULTIMODGRAPH

Our main objective is, for a given query, to find a ranked list of modeling patterns in the repository such that the returned patterns are as similar as possible to the modeling pattern that represents the query. In our approach, the problem can be rephrased as discovering a ranked list of subgraphs in the set of project graphs similar to the query graph. We define the notion of similarity as follows: A query graph is similar to a retrieved project subgraph based on the similarity of the textual content represented by the labels of the vertices and edges in the attributed graphs, as well as the similarity in their corresponding graph topologies. Moreover, the size of these similar subgraphs should be comparable to the size of the query graph, so that upon retrieval, these subgraphs (or the modeling patterns they represent) can be reused to build new software models with as few modifications as possible. One may note that in one large project graph there might be multiple subgraphs similar to a query graph, since a given task can be presented with different modeling patterns, and we would like to capture also those kind of modeling patterns.

Our approach consists of an indexing phase, in which vertices of the model graphs are indexed for efficient search, and a search phase, in which (i) an index lookup is performed on the vertices to find potentially matching candidates, (ii) the matched vertices are expanded to form subgraph patterns that are similar to the query graph, and (iii) the subgraphs are ranked with respect to their similarity to the query graph.

In the indexing phase, illustrated in Figure 4, we build three types of indexes. The first index is a grid index, that uses multidimensional scaling to cluster similar graph vertices from all of the project graphs for each attribute that represents a different model feature: name, type and data (cf., Section 3.2).

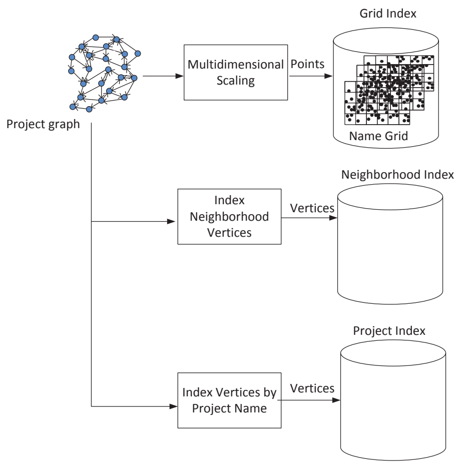


Figure 4: Indexing in MultiModGraph.

We use multidimensional scaling because our preliminary evaluations (detailed results are available in [1]) showed that it allows efficient pruning of most of those vertices that are beyond a distance threshold from a given query vertex. For expansion of the vertices matched using the grid index, two more auxiliary indexes are built: (i) a neighborhood index that considers the vertex neighborhood, and (ii) a project index that considers the vertices belonging to a graph (project), specified by their name. The indexing phase is discussed in more detail in Section 5.1.2.

In the search phase, shown in Figure 5, query vertices are also transformed into points in space through the same multidimensional scaling algorithm [5]. These points are used to search the grid index, retrieving only those vertices similar to the corresponding query vertices with respect to a specific attribute. Then, the project and the neighborhood index are searched to expand the matching candidates and form local subgraph matches, which are subsequently ranked considering a graph-edit distance metric as a similarity measure. Further details of search and match expansion can be found in Sections 5.1.3 and 5.1.4, respectively.

5.1 Graph search using multidimensional scaling

In this section, we present MultiModGraph in more detail. We illustrate the concept of graph similarity, i.e. how a query graph is defined to be similar to a subgraph of the project graph (Section 5.1.1); we describe the process of indexing the project graphs (Section 5.1.2), the search of similar query vertices to find matching candidate vertices (Section 5.1.3), the expansion of the matching candidate vertices to produce local subgraph matching patterns, and the ranking of the matching patterns with respect to the query (Section 5.1.4).

5.1.1 *Graph Similarity*

The similarity between the query graph and each of the project subgraphs is computed through graph-edit distance, a measure that specifies the number of graph-edit operations that transform one graph into the other. The considered operations are:

- *Vertex substitution*: a vertex in the project subgraph is substituted with a vertex in the query graph if they are similar. Two vertices are similar if the project graph vertex is retrieved as a result of searching the grid index for a specific query vertex considering at least one attribute.
- *Edge substitution*: an edge in the project subgraph is substituted with an edge in the query graph if their type labels belong to a similar type, and if their incident vertices are substituted. Two edge labels are similar if they are identical, or if they both belong to the set of WebML links, excluding the containment relationships.
- *Vertex deletion*: a vertex from the query graph that does not have corresponding similar vertices in the project subgraph is deleted from the query graph.
- *Vertex insertion*: a vertex from the project subgraph that does not have corresponding similar vertices in the query graph is inserted in the query graph.
- *Edge insertion*: an edge from the project subgraph that does not have corresponding similar edges in the query graph is inserted in the query graph.
- *Edge deletion*: an edge from the query graph that does not have corresponding similar edges in the project subgraph is deleted from the query graph.

5.1.2 *Indexing*

A. *Grid Index*
Given a set of data objects and the distance values between each pair of objects, multidimensional scaling assigns coordinates to each data object, such that distances computed from the assigned coordinates are as representative as possible to the actual distances. While this technique is used mainly in data visualization [15], we exploit this idea for clustering and then efficiently indexing the vertices of the model graphs.

We perform clustering of graph vertices with respect to the vertex attributes that correspond to the metamodel attributes in a model element. For our specific context, we consider the name, type and data attributes. Clustering

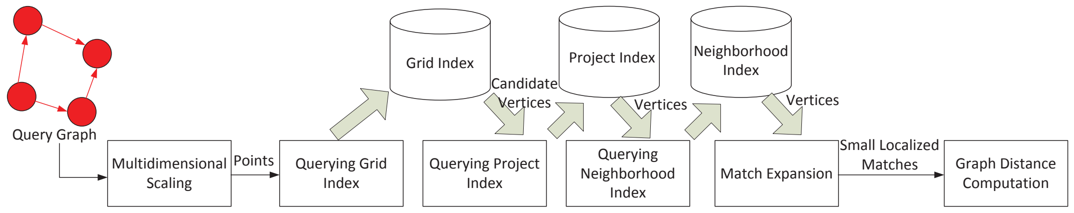


Figure 5: Search and Match Expansion in MultiModGraph.

is achieved by transforming vertex attributes' values representing a specific attribute class (name, type and data) as points in multidimensional space. The distance between points, preserved by the multidimensional scaling, is computed by the Euclidian distance. These computed distances help to find for a graph vertex, its "nearby" graph vertices with respect to a single attribute. The transformed points are placed into multidimensional grids, as shown in Figure 6. Each grid corresponds to a metamodel attribute, i.e., name, type and data. Therefore, the total number of grids is the same as the number of attributes (in our case three). The number of dimensions of each grid is equal to the number of dimensions of the points representing a specific attribute. These grids are used to build the grid index which allows for efficient pruning of all vertices that are not within a specified distance from a query vertex. In this work we chose the Chalmers algorithm [5] for performing multidimensional scaling, because it has lower computational overhead (quadratic) than other multidimensional scaling approaches without introducing too much noise. It is a heuristic-based approach, hence it does not provide tight error bounds. The quality evaluation of the algorithm is presented in [1].

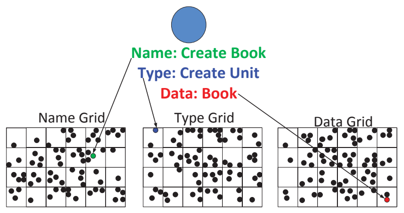


Figure 6: Grid Index in MultiModGraph.

B. *Auxiliary Indexes*

Besides the grid index, two other index structures are constructed and used in the search algorithm.

- The neighborhood index is an inverted index that keeps track of the neighborhood of each vertex, where for each vertex, all the vertices wthin its 2-hop neighborhood considering both ancestors and descendants are stored. This index considers the local structure around a graph vertex for expansion of already matched vertices. The 2-hop neighborhood has been selected for two reasons: (i) to better respond to the diversity of modeling patterns expressing a given task; (ii) to allow vertex mismatches between a query graph and a local subgraph match, since we perform approximate, and not exact matching. For scalability reasons, we do not exceed 2-hop neighborhood.
- The project index is an inverted index that for each vertex stores the corresponding project name. It is used to form subgraphs of vertices that belong to the same project.

5.1.3 Search

The search algorithm is described in *Algorithm 1* and presented in Figure 5. V_Q represents the set of vertices of the query graph, while *attribute* is the set of attribute types, namely, name, type and data attribute. The search process, takes each vertex v_q of the query graph and transforms it into a set of points in space $queryPoint$, using the Chalmers algorithm, where each point represents a specific attribute a. These points are used to query the grid index. Each query point is positioned in the corresponding grid in a similar way as the points that represent vertex label attributes from the project graphs are positioned by multidimensional scaling (using a single iteration of the Chalmers algorithm). In this way, the query point is "querying" the grid to find points *points* that are within an acceptable distance value *distance* (a user-specified parameter) with respect to a specific attribute a. Since the grids are independent, different acceptable distances can be assigned for each attribute (grid). This step performs pruning of points that are distant (dissimilar) from the query point, keeping only those points *points* that are within the specifed distance values[2].

When all the points for each attribute are retrieved, a union is performed across the different attribute classes, thus merging the results obtained from all the three grids into a set of candidates *candidates*. To further reduce the number of vertices, the candidate points are pruned by checking whether the real distance of the vertex labels they represent is within the acceptable distance values. As a real distance for name and data attribute we consider the string-edit distance, while for the type attribute we consider the distance between two types in the metamodel tree, normalized with respect to the longest distance in the tree. Finally, for each query vertex we obtain a set of real candidate vertices $realCandidates$ which represent the potential matches.

5.1.4 Match Expansion and Ranking

Each matching candidate is expanded in order to form small localized subgraph structures, denoted as matching patterns, achieved with the help of the project and the neighborhood indexes, as illustrated in Figure 5. The process of match expansion is described in Algorithm 2. At the beginning, for each query vertex v_q, a set of patterns is created, such that each pattern consists of one real candidate of v_q. Then, if query vertex v_q is not the first exam-

[2]Note that we will use the terms vertex and its representing point interchangeably.

Algorithm 1 Search Algorithm

```
Require: $V_Q, attribute, distance$
  for all $v_q \in V_Q$ do
    $candidates \leftarrow \emptyset$
    for all $a \in attribute$ do
      $queryPoint \leftarrow ChalmersQuery(v_q, a)$
      $points[a] \leftarrow findCandidates(queryPoint,$
          $grid[a], distance[a])$
      $candidates \leftarrow candidates \cup points[a]$
    end for
    $realCandidates[v_q] \leftarrow prune(v_q, candidates)$
  end for
```

ined vertex, the algorithm checks whether its set of candidates *realCandidates* can extend already existing patterns. Namely, a project vertex from the candidates set *realCandidates* is added to an already created pattern in the set of existing patterns *patternSet*, if all of the following conditions are met:

- The project vertex represents a matching candidate for different query vertices with respect to the already matched query vertices.
- The project vertex belongs to the same project graph as the current matching pattern. This information is retrieved from the project index.
- The project vertex is within the 2-hop neighborhood of any of the project vertices present in the matching pattern. This information is retrieved from the neighborhood index.

As a result, a new set of patterns is created (*newPatternSet*), used to update the set of patterns *patternSet*.

The matching is complete when, for a matching subgraph pattern, all the query vertices have been examined for potential matches. Additional vertices from the project graph are added to the matching pattern if they are in the intersection *intersection* of the neighborhoods of the pattern's vertices (retrieved from the neighborhood index). In this way, all the vertices of a pattern are found. They are used to build a subgraph *subgraph* by finding the edges that connect these vertices from the project graph. Thus, *subgraph* represents a subgraph of the project graph. Once a matching subgraph is built, it is compared to the query graph with respect to the similarity, as explained in Section 5.1.1. Graph-edit distance is used as a similarity metric to rank the modeling patterns with respect to their similarity to the query. In the graph-edit distance computation, the subsituted vertices from a subgraph pattern are those that represent the real candidates, while the inserted vertices are the vertices that were added additionally to the pattern as a result of the intersection with the neighborhood index.

6. RESULTS

We evaluated our approach on a repository of WebML models[3] which consists of 12 real-word WebML projects from different application domains. The projects were divided into segments such that each segment represents a different WebML area in the project (i.e., areas group pages with similar purpose). This way, we obtained 341 segments in total.

The test queries were generated as follows. First, a set of exemplary models were selected by considering different

[3]Provided by the WebRatio company `www.webratio.com`

Algorithm 2 Match Expansion Algorithm

```
Require: $V_Q, realCandidates$
  $patternSet \leftarrow \emptyset$
  for all $v_q \in V_Q$ do
    $patternSet \leftarrow patternSet \cup createPatterns(\emptyset,$
        $realCandidates[v_q])$
    if $notFirstVertex(v_q)$ then
      for all $pattern \in patternSet$ do
        if $realCandidates[v_q]$ within two-hop neighborhood of
        $pattern$ and $realCandidates[v_q]$ in the same project as
        $pattern$ then
          $newPatternSet \leftarrow$
              $addToExistingPattern(pattern,$
              $realCandidates[v_q])$
          $patternSet \leftarrow$
              $updatePatternSet(patternSet, newPatternSet)$
        end if
      end for
    end if
  end for
  for all $pattern \in patternSet$ do
    for all $v_p \in pattern$ do
      $intersection \leftarrow neighborhood(v_p) \cap$
          $neighborhood(pattern - \sum_{i=1}^{p-1} v_i)$
      if $intersection \neq \emptyset$ then
        $pattern \leftarrow addAdditionalVertices(pattern,$
            $intersection)$
      end if
    end for
    $subgraph \leftarrow buildGraph(pattern)$
  end for
```

WebML modeling patterns, a variety of metamodel concepts and a vocabulary of labels present in the repository. Then, three experienced model developers selected 10 models from the initial set of exemplary models that they believed better represented the typical user needs of a model developer [2]. Subsequently, these models were transformed into graphs (as explained in Section 3.2), which constitute the test queries in our evaluations.

To obtain the ground truth (used in our evaluations), we asked the same model developers to manually evaluate the relevance of each query against each project segment, where a relevance score of (i) 0 implies no relevance, (ii) 1 implies *either* textual or structural similarity, and (iii) 3 implies *both* textual and structural similarity. The final scores were computed by averaging the scores reported by the three domain experts, which was then rounded to the nearest integer [2]. Note that we did not use 2 as a score value to achieve greater diversity in the aggregate scores.

Given a query, MultiModGraph returns a set of modeling patterns (that it believes are relevant to the query). Hence, to assess the quality of the algorithm, we evaluate the relevance of each returned modeling pattern to the given query (based on the ground truth). However, note that a modeling pattern might span multiple project segments. Therefore, to assess a modeling pattern's relevance to a query, we consider all of the project segments that the modeling pattern spans. The final relevance of a modeling pattern is computed as an average of the relevance of the project segments.

In our evaluations, parameters of the Chalmers algorithm such as (i) number of iterations, (ii) max number of points in the random set and (iii) number of dimensions that are used for representing points in space, have been manually tuned to their optimal values [2]. As for the distance thresholds, we performed a preliminary evaluation to discard certain combinations of distance values across the three attribute classes: name, type and data. In Table 1, we show the acceptable distance values we use for each attribute class.

We have observed that using smaller distance values for the name and the type attribute classes does not retrieve sufficient number of candidate points. On the other hand, using greater distance values loosens the distance constraints, i.e., precision drops.

Table 1: Distance values configurations.

Name distance	Type distance	Data distance
0.4	0.4	0.2
0.4	0.4	0.4
0.6	0.4	0.2
0.6	0.4	0.4

As the baseline in our evaluations, we use the A-star algorithm [17] that we adapted for searching WebML models (details are in [2]). First, we compare MultiModGraph against the A-star algorithm with respect to their 11-point interpolated average precision, a metric that combines precision and recall by measuring the highest precision obtained at 11 standard levels of recall (ranging from 0.0 to 1.0) [14]. Namely, for each recall level i, the precision is computed as the maximum precision value for recall levels $j > i$, which is averaged across *all* of the test queries. In the computation of precision and recall values, we consider every modeling pattern with $relevance > 0$ relevant, while every modeling pattern with $relevance = 0$ irrelevant.

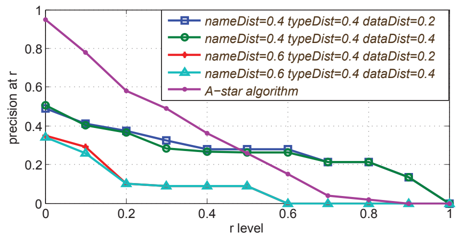


Figure 7: 11-point Interpolated Average Precision for different distance value configurations and the A-star algorithm.

Figure 7 shows the 11-point interpolated average precision for the four different configurations of MultiModGraph and the baseline algorithm. Each algorithm was configured to return the top 150 results. The values denoted as $nameDist$, $typeDist$ and $dataDist$ represent the acceptable distance values for the name, type and data attribute classes, respectively.

MultiModGraph achieves the best configuration with values of $nameDist = 0.4$ and $typeDist = 0.4$, while increasing $dataDist$ from 0.2 to 0.4 does not affect precision/recall. Increasing the $nameDist$ value further to 0.6, however, significantly worsens performance. Compared to the A-star (baseline) algorithm, MultiModGraph performs better for recall values greater than 0.5. This is important because it means that the algorithm still continues to retrieve relevant models at high levels of recall.

Figure 8 presents the best-case behaviour of MultiModGraph for configuration values $nameDist = 0.4$, $typeDist = 0.4$ and $dataDist = 0.4$, where queries with the best two 11-point interpolated curves are reported. The algorithm achieves a maximum precision of 1 even at lower levels of recall: up to 0.3 for Query 6, and up to 0.8 for Query 2. For other queries, the algorithm performs worse, which influences the overall algorithm performance when averaged across all of the queries.

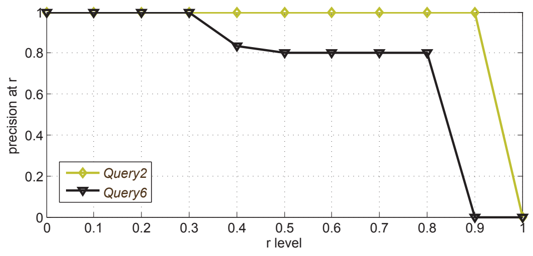
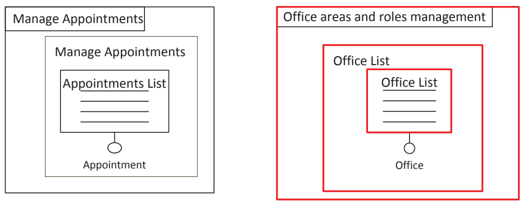


Figure 8: 11-point interpolated average precision for the best performing queries for $nameDist = 0.4$, $typeDist = 0.4$ $dataDist = 0.4$ configuration: Query 2 and Query 6.

MultiModGraph's lower performance in precision in some queries can be attributed to the way the algorithm selects candidate vertices. For a query vertex, a vertex in the project graph is considered a match candidate if at least one of the distances for a specific attribute class are within the specified distance thresholds. This generates patterns similar to the query, where each model element (in the result) is similar to a query model element with respect to a different attribute class. Some of these alternative matching patterns are still "reusable" (indeed, a closer manual inspection of the results further confirms this fact), but not all of them have been considered relevant by the ground truth.



(a)

Figure 9: Example of a query and its corresponding "irrelevant" result as deemed by the ground truth.

For Query 7, its highly ranked result by MultiModGraph is depicted in Figure 9, where each matched element in the result is highlighted by a red rectangle. Note the similarity between the query (on top) and its result (at the bottom). The query is about management of appointments and the result is about management of office areas and roles, but otherwise, they are structurally equivalent.

This highlights two possible future directions. First, ground truth generation can be improved to include relevance assessments at more fine-grained segments (i.e., currently, a project segment corresponds to a WebML area in the project). Second, improvements to the ranking function can be made to capture users' varying notions of relevance across different attribute classes (i.e., name, type and data attributes).

Lastly, we examine the runtime performance of MultiModGraph and compare it against the runtime performance of the baseline. We consider the average execution times over the entire query set. For this experiment, we varied the number of indexed vertices. The experiments were performed on a machine with Intel dual Core Processor 2.4 GHz, 6 GB RAM and Windows 7 (64-bit) operating system.

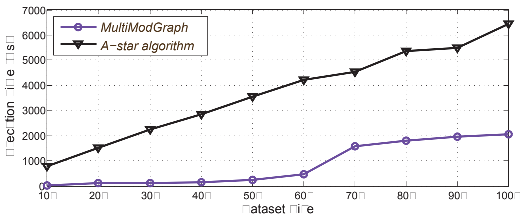


Figure 10: Runtime performance of MultiModGraph and A-star algorithm.

The runtime performance of the MultiModGraph is much better than the runtime performance of the A-star algorithm (on average, MultiModGraph is 12 times faster than the A-star algorithm), as demonstrated in Figure 10. The improvement in runtime performance is due to the use of indexing, however, it comes at a cost of some loss in the quality of the retrieved results, which is due to the approximate nature of the multidimensional scaling process. However, further optimizations are possible to improve both the quality and the runtime performance of MultiModGraph.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a graph-based approach for searching WebML repositories that uses multidimensional scaling. We have evaluated our approach on a real-world WebML repository using 10 test queries. Our preliminary results show that MultiModGraph has a better runtime performance than the baseline algorithm, but this comes at the cost of accuracy. We have argued that some of this inaccuracy could be attributed to the way the ground truth is generated. However, it may also be possible to improve performance by considering alternative objective functions for ranking, which is an integral part of our future work. Some other future work directions include: (i) applying MultiModGraph to different types of models by modifying the graph representation and the grid index according to the metamodel of the modeling language; (ii) testing the scalability of the approach on larger model collections; (iii) automatic tuning of the Chalmers algorithm parameters; (iv) performing efficiency comparison with existing indexing techniques for graph-edit distance (e.g. Closure tree [7]); and (v) tuning the search order in the search algorithm, by matching vertices with less candidates first.

8. REFERENCES

[1] B. Bislimovska. *Textual and content based search in software model repositories.* PhD thesis, Politecnico di Milano, 2014.

[2] B. Bislimovska, A. Bozzon, M. Brambilla, and P. Fraternali. Textual and content-based search in repositories of web application models. *ACM Transactions on the Web (TWEB)*, 8(2):11, 2014.

[3] A. Bongio, P. Fraternali, M. Brambilla, S. Comai, and M. Matera. *Morgan Kaufmann series in data management systems: Designing data-intensive Web applications.* Morgan Kaufmann, 2003.

[4] S. Ceri, P. Fraternali, and A. Bongio. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, 33(1-6):137–157, 2000.

[5] M. Chalmers. A linear iteration time layout algorithm for visualising high-dimensional data. In *Visualization'96. Proceedings.*, pages 127–131. IEEE, 1996.

[6] D. Grigori, J. C. Corrales, M. Bouzeghoub, and A. Gater. Ranking bpel processes for service discovery. *Services Computing, IEEE Transactions on*, 3(3):178–192, 2010.

[7] H. He and A. K. Singh. Closure-tree: An index structure for graph queries. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 38–38. IEEE, 2006.

[8] T. Jin, J. Wang, M. La Rosa, A. Ter Hofstede, and L. Wen. Efficient querying of large process model repositories. *Computers in Industry*, 64(1):41–49, 2013.

[9] A. Khan, Y. Wu, C. C. Aggarwal, and X. Yan. Nema: Fast graph search with label similarity. *Proceedings of the VLDB Endowment*, 6(3):181–192, 2013.

[10] A. G. Kleppe, J. B. Warmer, and W. Bast. *MDA explained: the model driven architecture: practice and promise.* Addison-Wesley Professional, 2003.

[11] V. Krishna, N. Ranga Suri, and G. Athithan. Mugram: An approach for multi-labelled graph matching. In *International Conference on Recent Advances in Computing and Software Systems (RACSS), 2012*, pages 19–26. IEEE, 2012.

[12] M. La Rosa, H. A. Reijers, W. M. Van Der Aalst, R. M. Dijkman, J. Mendling, M. Dumas, and L. García-Bañuelos. Apromore: An advanced process model repository. *Expert Systems with Applications*, 38(6):7029–7040, 2011.

[13] D. Lucrédio, R. P. d. M. Fortes, and J. Whittle. Moogle: a metamodel-based model search engine. *Software & Systems Modeling*, 11(2):183–208, 2012.

[14] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval*, volume 1. Cambridge university press Cambridge, 2008.

[15] A. Morrison, G. Ross, and M. Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. *Information Visualization*, 2(1):68–77, 2003.

[16] M. Niemann, M. Siebenhaar, S. Schulte, and R. Steinmetz. Comparison and retrieval of process models using related cluster pairs. *Computers in Industry*, 63(2):168–180, 2012.

[17] A. Sanfeliu and K.-S. Fu. A distance measure between attributed relational graphs for pattern recognition. *Systems, Man and Cybernetics, IEEE Transactions on*, (3):353–362, 1983.

[18] Y. Tian and J. M. Patel. Tale: A tool for approximate large graph matching. In *IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008.*, pages 963–972. IEEE, 2008.

[19] Z. Yan, R. Dijkman, and P. Grefen. Fast business process similarity search. *Distributed and Parallel Databases*, 30(2):105–144, 2012.

[20] J. Yang, S. Zhang, and W. Jin. Delta: indexing and querying multi-labeled graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, pages 1765–1774. ACM, 2011.