Graph-Based Search over Web Application Model Repositories

Bojana Bislimovska, Alessandro Bozzon, Marco Brambilla, and Piero Fraternali

Politecnico di Milano, Dipartimento di Elettronica e Informazione
P.za L. Da Vinci, 32. I-20133 Milano - Italy
{name.surname}@polimi.it

Abstract. Model Driven Development may attain substantial productivity gains by exploiting a high level of reuse, across the projects of a same organization or public model repositories. For reuse to take place, developers must be able to perform effective searches across vast collections of models, locate model fragments of potential interest, evaluate the usefulness of the retrieved artifacts and eventually incorporate them in their projects. Given the variety of Web modeling languages, from general purpose to domain specific, from computation independent to platform independent, it is important to implement a search framework capable of harnessing the power of models and of flexibly adapting to the syntax and semantics of the modeling language. In this paper, we explore the use of graph-based similarity search as a tool for expressing queries over model repositories, uniformly represented as collections of labeled graphs. We discuss how the search approach can be parametrized and the impact of the parameters on the perceived quality of the search results.

1 Introduction

Software project repositories are a main asset for modern organizations, as they store the history of competences, solutions, and best practices that have been developed in time. Such invaluable source of knowledge must be easily accessible to analysts and developers through easy and efficient querying mechanisms. This is especially true in the context of Model Driven Development, where models can be reused and shared to improve and simplify the design process, while providing better management and development of software.

Search engine can be a winning solution in this scenario. Source code search engines are already widely adopted and exploit the grammar of the programming languages as an indexing structure of artefacts. Innovative model-based search engines should index their content based on the metamodels of the stored models: Domain Specific Languages (DSL) models, UML models, Business Process Models and others.

Even if traditional keyword querying [8] is the user interaction paradigm for search engines, query by example (content-based) approaches for search have often proven effective in finding results which more closely reflect the user needs

S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.): ICWE 2011, LNCS 6757, pp. 90–104, 2011.

[2]. In the context of model-based search, the query by example interaction can be enacted by providing as queries model or model fragment, so to consider as part of the user information need also the relationships between elements. Such an approach calls for matching and ranking algorithms that rely on techniques such as schema matching [16] or graph matching [7] for finding similarities between models.

The goal of this paper is to propose a general-purpose approach using graph query processing, as a form of querying by example, for searching repository of models represented as graphs. This is realized by performing similarity search using graph matching based on the calculation of graph edit distance. The contribution of this paper includes: 1) a graph based approach for content-based search in model repositories; 2) a flexible indexing strategy adaptive to the relationships among model elements found in the DSL metamodel; 3) a similarity measure that exploits semantic relationships between model element types; and 4) implementation and evaluation of the proposed framework by using projects and queries encoded in a Web Domain Specific Language called WebML (Web Modeling Language)[1]. Although the evaluation is performed on a repository of WebML projects, the approach is general and can be applied to any DSL described through a metamodel.

The paper is organized as follows: Section 2 presents the state of the art for repository search engines; Section 3 outlines the fundamentals of model-based search; Section 4 illustrates our graph-based solution for model search; Section 5 discusses the experiments performed on a Domain Specific Language, showing the experiment design and the results obtained; Section 6 draws the conclusions and the future work directions.

2 Related Work

The problem of searching relevant software artifacts in software repositories has been extensively studied in many academic works and widely adopted by the community of developers.

Model search requires some knowledge about the model structure for indexing and querying models. Moogle is a model search engine that uses UML or Domain Specific Language (DSL) metamodels to create indexes for evaluation of complex queries [8]. Our previous work [1] performs model search using textual information retrieval methods and tools, including model segmentation, analysis and indexing; the query language adopted is purely keyword-based. *Nowick et al.* [13] introduce a model search engine that applies a user-centric and dynamic classification scheme to cluster user search terms. Existing approaches performing content-based search rely on graph matching or schema matching techniques. Graphs allow a general representation of model information. Graph matching determines the similarity of a query graph and the repository graphs by matching approximately the query and the model. An indexing approach for business

[1] http://www.webml.org

process models based on metric trees (M-Trees), and a similarity metric based on the graph edit distance is illustrated in [7]. *TALE* [19] provides approximate subgraph matching of large graph queries through an indexing method that considers the neighbors of each graph mode. Three similarity metrics for querying business process models are presented by *Dijkman et al.* in [4], based on the label matching similarity, structural similarity which also includes the topology of models, and behavioral similarity which besides the element labels considers causal relations in models.The same authors propose the structural similarity approach in [15] which computes similarity of business process models, encoded as graphs, by using four different graph matching algorithms: a greedy algorithm, an exhaustive algorithm with pruning, a process heuristic algorithm, and the A-star algorithm. [2] uses extended subgraph isomorphism for matching approximate queries over databases of graphs.

Schema matching is the correlated problem of finding semantic correspondences between elements of two database schemas [9] [14]. An integrated environment for exploring schema similarity across multiple resolutions using a schema repository is given in [16]. HAMSTER [12] uses clickstream information from a search engine's query log for unsupervised matching of schema information from a large number of data sources into the schema of a data warehouse.

Other content-based approaches utilize specific algorithms for search. The work in [18] uses domain-independent and domain-specific ontologies for retrieving Web Service from a repository by enriching Web service descriptions with semantic associations. [10] presents a framework for model querying in the business process modeling phase, enabling reuse, support of the decision making, and querying of the model guidelines.

The focus of this paper is on content-based search on web application models, by adopting a graph matching method. For the graph matching we chose the *A-star algorithm* described in [15]. The original work applied *A-star* only to business process models, while our final aim is to propose an approach that is metamodel-aware, and therefore general enough to be applied to different models and languages. We also provide an extensive analysis of the search framework, to show how different parameter settings influence the quality of results.

3 Fundamentals of Model-Based Search

Search of model repositories can be performed in several ways. In *text-based model search*, a project in the repository is represented as an unstructured document composed of (possibly weighted) bag of words; projects are retrieved by matching keyword-based queries to the textual information extracted from a model, i.e., its metamodel grammar and the annotations produced by developers. Facets can be used to allow the user to further filter the information using model properties; advanced result visualization may facilitate the identification of query results within the examined repository [1].

In *content-based model search*, the role of the metamodel is more prominent, as it allows for a finer-grained information retrieval that takes into account the rela-

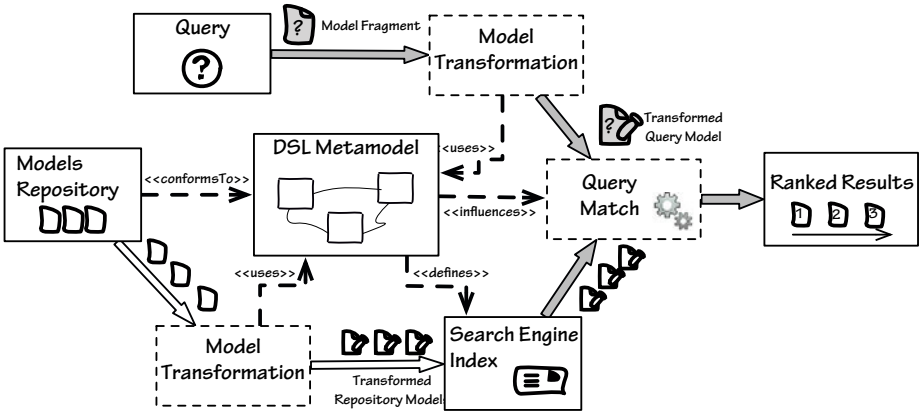


Fig. 1. The content analysis (white arrows) and query (grey arrows) flows of a content-based model search system

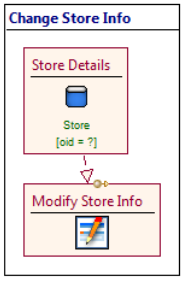
tionships between elements, sometimes also considering their semantic similarity. In addition, the metamodel deeply influences the way models are represented in the search engine, as the indexes must contain information about the properties, hierarchies and associations among the model concepts.

Queries over the model repository are expressed using models fragments; a query conforms to the same metamodel as the projects in the repository, and results are ranked by their similarity w.r.t. the query. The DSL metamodel can also influence the search engine matching and ranking operation by providing domain-specific information that can help fine tuning its behavior.

The prominent role of the metamodel demand for a revisitation of the classical IR systems architecture: Figure 1 shows the organization of the two main information flows in a *content-based model search* system: the *content processing* and the *query* flows. In a general-purpose content-based search system, the indexing and querying processes are designed to be model-independent, in the sense that they can be configured based on the meta-model of interest. The designer of the system only has to decide which is the information that needs to be extracted, and the meta-model based rules for extraction.

Content processing is applied to the projects in the repository in order to transform them and extract the relevant information for the index creation. Projects are transformed into a suitable representation that captures the model structure and also contains general information about the project and information for each model element, like element name, type, attributes, comments, relationships with other elements, or any other information that can be expressed in a model. Such a transformation is driven by the models' metamodel, and produces as output a metamodel-agnostic representation (e.g., a graph) that is used to build the structured index subsequently used for queries.

The **Query processing** phase of content-based search applies the same transformations to the query as to the projects in the repository, because the query




```
<SiteView id="sv1" name="Administrator">
 <Page id="page1" name="Change Store Info">
    <ContentUnits>
      <DataUnit id="dau1"name="Store Details”>
        <Link  id="ln1" name="Modify Store" to="enu1"/>
      </DataUnit>
      <EntryUnit id="enu1" name="Modify Store Info"/>
    </ContentUnits>
 </Page>
</SiteView>
```

(b)

Fig. 2. Example of WebML model (a) and its XML representation (b)

itself is a model or model fragment. Then, the query is matched against the index using different algorithms. As a result, a ranked list of projects is obtained, according to the similarity with the query. Several model comparison techniques can be employed, e.g., based on alternative graph matching approaches.

4 A Graph-Based Approach to Model-Based Search

In this section we elaborate on a general-purpose *content-based model search* approach using query on graph (by example) for searching repository of Web application models represented as graphs. Graphs provide a convenient way for representing data in many different application domains such as computer vision, data mining and information retrieval. To fully exploit the information encoded in graphs, effective and efficient tools are needed for their querying. We implement content-based search by using a graph matching (subgraph isomorphism) technique, to retrieve the most relevant projects from a repository with respect to a query expressed as a model (fragment). We assume that both the models in the repository and the query model conform to the same metamodel.

4.1 Scenario: Repository of WebML Models

To ease the discussion, we exemplify our approach using WebML (Web Modeling Language) [3] models. WebML allows high-level description of a Web site considering different dimensions. The main WebML constructs are pages, units and links, organized into areas and site views. A site view is a coherent hypertext, fulfilling a well-defined set of requirements for a user role. Pages are composed by units. Content units are atomic elements that determine the web page content while operation units support arbitrary business logic triggered by navigation. Units are connected through links forming a hypertext structure.

WebML models can be represented with a graphic notation or with an XML syntax. Figure 2 shows a very simple excerpt of WebML model: the *ChangeStoreInfo page* of the Administrator site view contains a *StoreDetails data unit* that

shows the details about a store, and a link to the *ModifyStoreInfo entry unit* that allows the user to submit new data through a form; a link from the data to the entry unit denotes that the fields of the latter are preloaded with values extracted from the former.

4.2 Graph Representation of Models

Our work exploits graph matching by finding a mapping between two graphs: the query graph and the project graph. Graphs allow abstract representation of models, encoding the specific model features into nodes and edges. Therefore, the models from the repository and the queries are represented as directed annotated graphs, preserving model topologies, and encapsulating the model information as annotations.

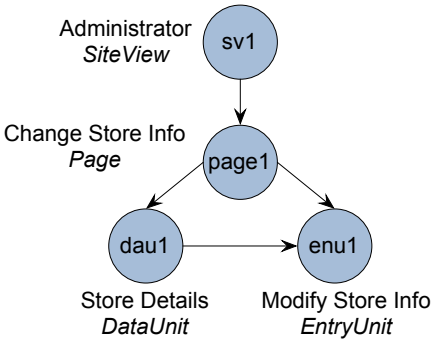
In the example scenario, WebML model elements (e.g., siteviews, pages, units) are represented as nodes in a graph, while the containment relationships (e.g., a page containing a unit) and links (between pages or units) are represented as edges. To simplify the discussion, we assume that each node is annotated only with the name and the type of the concept it represents. In a real setting, model elements and relations can be annotated with all the properties specified in the language metamodel.

The graph representation is obtained exploiting the metamodel to create a model transformation that is applied to every project in the repository. Such a transformation can be expressed in any model transformation language (such as QVT or ATL). Considering our case study, the model depicted in Figure 2(a) can be transformed in a graph (encoded as a GraphML model[2].) as in Figure 3, where both a graphical (a) and XML (b) representations are shown. The whole project repository is transformed into a set of graphs offline, so to build the structured index required to perform queries. The query is processed exactly in the same way as the projects in the repository, thus obtaining graphs with equal representation, suitable for comparison.

4.3 Node Matching

The basic building block for any graph matching algorithm is *node matching*, i.e., the way in which one single node from a graph is matched to each node in the other graph. In our approach, we adopt a structured node matching approach, based on node distance. Node distance measures how similar are two nodes, considering one or more properties of the node itself. In our approach, the node matching operation is expressed as a function of the considered metamodel: given that a model element is described by its properties, node matching can be expressed as a multi-dimensional distance function that induces a topology (in a metric space) on the given nodes set. In our discussion, model elements are described by nodes annotated with a *name* and a *type* labels, and we express the distance function between two generic nodes n_1 and n_2 as follows:

[2] http://graphml.graphdrawing.org/




```
<graphml>
  <graphedge default="directed">
    <node id="sv1">
      <name>Administrator</name>
      <type>SiteView</type>
      <occurence>1</occurence>
    </node>
    <node id="page1">
      <name>Change Store Info</name>
      <type>Page</type>
      <occurence>3</occurence>
    </node>
        ...

    <edge id="edge page1">
      <source>sv1</source>
      <target>page1</target>
    </edge>
  </graph>
</graphml>
```

(b)

Fig. 3. Pictorial (a) and XML (b) representations of the graph used for indexing and search purposes for the WebML example of Figure 2

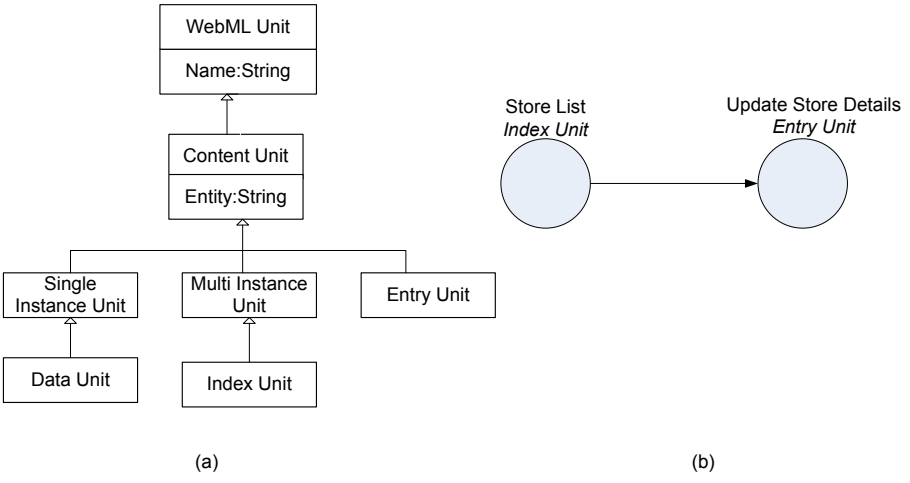


Fig. 4. (a) An extract of the WebML metamodel for element type comparison; (b) Graph representation of two WeML units

$$Dist(n_1, n_2) = \lambda \cdot levDist(name_{n_1}, name_{n_2}) + (1 - \lambda) \cdot typeDist(n_1, n_2) \quad (1)$$

where: *levDist* is the normalized Levenshtein distance (string-edit similarity normalized with the length of the longer string) applied to the node names; *typeDist* is defined based on semantic relationships between the compared types by exploiting the information contained in the project metamodel. In particular, it can be defined as the normalized node distance between the two classes in the metamodel graph.

The range of $Dist$, $levDist$, and $typeDist$ is always *[0,1]*. Given the above definition, $Dist(n_1, n_2)$ computes a standard Levenshtein distance when $\lambda = 1$, (thus

only considering the name of the involved nodes), and only the distance between model elements when $\lambda = 0$. For intermediate values of λ in the interval $[0, 1]$, a linear combination of both criteria is calculated. Figure 4(b) shows an example of two WebML elements for which the node distance calculation can be performed. From the example WebML metamodel of Figure 4(a) it can be calculated that the type distance between an *Index Unit* and an *Entry Unit* of Figure 4(b) is 0.75 (because the distance between the two classes is 3 and the maximum node distance in the graph is 4), while $levDist(\text{“}StoreList\text{”}, \text{“}UpdateStoreDetails\text{”})$ is 0.65. Therefore, with $\lambda = 0.5$ the distance between the two nodes is 0.7.

4.4 Graph Matching

Graph matching is the procedure of finding a mapping between two graphs for the purpose of calculating their similarity. Finding a complete node-to-node correspondence without violating both structure and label constraints of a graph is the problem of *graph isomorphism*. In most cases, graphs involved in the comparison have different dimensions. In our approach, the problem is related to *subgraph isomorphism*, which can be seen as a way to evaluate subgraph equality. A subgraph isomorphism is a weaker form of matching, as it requires only that an isomorphism holds between a query graph and a subgraph of the model graph [11].

Graph Edit Distance. In order to compare two graphs, a metric based on graph edit distance is used. The graph edit distance between two graphs is the minimal total cost of edit operations needed to transform one graph into the other [15]. The edit operations considered are:

- *Node substitution*: A node from one graph is substituted with a node from the other graph if they are similar;
- *Node insertion/deletion*: A node is inserted into a graph to match an existing one in the other or viceversa;
- *Edge insertion/deletion*: An edge is inserted into a graph to match an existing one in the other or viceversa.

Each of the above mentioned operations has a cost heuristically associated to it. The costs for insertion/deletion of nodes and edges are fixed to a value in the interval *[0,1]*, while the cost for node substitution is defined as *1 minus the node similarity*. We also define a *cutoff threshold* on the similarity between nodes, so as nodes with similarity above threshold are considered similar and therefore substituted, while nodes below the threshold need to be inserted/deleted. In order to quantify how much two graphs are similar we normalize the graph edit distance to the range of [0,1]. Then, the graph edit similarity between two graphs G_1 and G_2 is defined as [15]:

$$G_{Sim}(G_1, G_2) = 1 - \frac{w_{nI} \cdot f_{nI}(G_1, G_2) + w_{eI} \cdot f_{eI}(G_1, G_2) + w_{nS} \cdot f_{nS}(G_1, G_2)}{w_{nI} + w_{eI} + w_{nS}} \tag{2}$$

where f_{nI}, f_{eI} are the fractions of inserted nodes and of inserted edges respectively (i.e., the ratio between the inserted items and the total number of items), while f_{nS} is the average distance of the substituted nodes. The constant values w_{nI}, w_{nS}, and w_{eI} represent the weights for node insertion, node substitution, and edge insertion respectively, and can be varied to give more or less importance to each matching operation.

A-star Algorithm. The labeled graph representation of objects introduces noise and distortions. Moreover, often the query graph and the project graph do not have the same dimensions. Hence, it is necessary to include an error model and incorporate the concept of errors into graph matching. The graphs are then compared to each other by means of error-correcting subgraph isomorphism [11]. The most common approach to this problem is based on a search performed with the A-star algorithm on a tree representing the incrementally calculated *query to project distances*. A-star algorithm originally described by [5], is the best-first algorithm that finds the minimum cost path from one node to another in the distances tree. This algorithm is optimal since it examines the smallest number of nodes necessary to guarantee a minimum cost solution. The search space of the A-star algorithm can be greatly reduced by applying heuristic error estimation functions. By always expanding the node with the least cost, the algorithm is guaranteed to find the optimal mapping. Therefore, we adopt the A-star algorithm for error-correcting subgraph isomorphism detection that has been first applied in [11] and then adapted in [15]. The computational complexity of the algorithm depends on the size of the graphs, the number of labels and the number of errors [11].

The algorithm starts from a node n_1 in the query graph, and creates all the possible partial mappings from this node to every node in the project graph. Additionally, an extra mapping with a dummy node ϵ is created,(n_1, ϵ), equivalent to the case where n_1 is deleted. The partial mapping with the maximal graph edit similarity is selected, and expanded into a number of larger mappings. The algorithm proceeds with the next node from the query graph, and creates partial mappings with every node from the project graph, excluding the ones already in the mapping. The algorithm always selects the mapping with maximal graph edit similarity, and expands it, by adding a mapping for the next node. The algorithm finishes when all the nodes from the query graph are mapped. To reduce the memory requirements, only mappings between similar nodes are allowed. In other words, only nodes whose distance defined by equation 1 is greater than a threshold parameter can form a partial mapping.

The algorithm that calculates the similarity between a query graph G_1 and a project graph G_2 is represented by the pseudocode shown in Algorithm 1, where N_1, N_2 are the sets of nodes (and n_1, n_2 are the nodes) of graphs G_1, G_2 respectively; *open* is the set of all allowed mappings, and *map* is the partial mapping having the maximal graph edit similarity *s(map)*. The algorithm finishes when all the nodes from the query graph are examined. The returned value is the maximal graph edit similarity for the two graphs.

Algorithm 1. A-star algorithm

Require: $open \leftarrow (n_1, n_2) \mid n_2 \in N_2 \cup \{\epsilon\}, sim(n_1, n_2) > threshold \vee n_2 = \epsilon$, for some $n_1 \in N_1$
  while $open \neq 0$ **do**
    select $map \in open$, such that $s(map)$ is max
    $open \leftarrow open - map$
    if $dom(map) = N_1$ **then**
      return s (map)
    else
      select $n_1 \in N_1$, such that $n_1 \notin dom(map)$
      for all $n_2 \in N_2 \cup \{\epsilon\}$, such that ($n_2 \notin cod(map)$ and $sim(n_1, n_2) > threshold$) xor ($n_2 = \epsilon$) **do**
        $map' \leftarrow map \cup \{(n_1, n_2)\}$
        $open \leftarrow open \cup map'$
      end for
    end if
  end while

Complexity analysis. Let us assume a query with n nodes and a project with m nodes. The complexity of the comparison algorithm [11] in the best case is $O(n^2m)$ (when all the nodes are uniquely labeled and the query graph contains an isomorphic copy of the model graph), while in the worst case is $O(nm^n)$ (when the error in the query graph is very large).

5 Experimental Evaluation

This section discusses how the graph-based model similarity search described in the previous section can be parameterized, and highlights the impact of the parameter tuning on the perceived quality of the search results. Our experiments were conducted on a project repository composed of 30 real-world WebML projects [1] from different application domains (e.g., trouble ticketing, human resource management, multimedia search engines, Web portals, etc.), all encoded as XML files conforming to the WebML DTD. Projects' size varied from 100 to 1700 nodes. We manually built a query set of 15 queries with different sizes for the evaluation of the approach: 5 small queries (1-25 nodes), 5 medium-size queries (26-450 nodes), and 5 large queries (451-680 nodes). The queries were chosen based on their size, and different structure coverage.

We performed a manual assessment of the query-to-project relevance, so to establish a ground truth for performance evaluation.The assessment was carried out by one WebML expert analyst with modeling and application proficiency. Each query was manually compared against the project repository using the WebML models XML representation, considering element names, element types, and hierarchical positioning of the elements in the query and the project. Then, a relevance for each query against every project has been assigned, ranging from $0 - 10$ (with steps of 0.5), where 0 implies no relevance (i.e. the query has no match in the project), and 10 indicates maximal relevance. This evaluation

Table 1. Experimental parameters and Spearman's coefficient (mean and std on 15 queries) for experiments with node type similarity

λ		**Exp.1**	**Exp.2**	**Exp.3**
	$w_{edgeIns}$	0.1	0.1	1.0
	$w_{nodeIns}$	0.1	1.0	0.1
	$w_{nodeSub}$	1.0	0.1	0.1
0.25	**mean**(ρ)	0.66	0.40	0.52
	std(ρ)	0.11	0.16	0.18
0.5	**mean**(ρ)	0.65	0.50	0.54
	std(ρ)	0.11	0.18	0.17
0.75	**mean**(ρ)	0.67	0.64	0.63
	std(ρ)	0.12	0.15	0.16

induced a ranking of projects with respect to each query. Such ranking has been used as a reference ground truth for the evaluation of the algorithm.

To assess the robustness of the approach with respect to parameter changes, and to identify the optimal set of parameter values for the considered DSL, we tested our approach with different configurations of *node distance*, *node insertion*, *node substitution*, and *edge insertion* weights. Our analysis focused on two aspects: the quality of the results and the performance of the algorithm. The result quality is evaluated with respect to different parameterizations of the algorithm, while the performance is independent on the parameters, and thus is reported for one scenario only.

5.1 Analysis of the Quality of the Results

Our aim is to evaluate the impact of the parameters of the algorithm on the results quality. According to the definition of the algorithm, we can tune two types of parameters: the weights w_{nI}, w_{nS} and w_{eI} assigned to the frequencies of the graph edit distance (Eq. 2), and the value of the parameter λ in the node distance function (Eq. 1). In our experiment we defined 3 different configurations of these weights, and three different values of λ. The weight configurations have been selected in order to highlight the contribution of each component in the graph distance calculation. The values of λ have been set to respectively 0.25, 0.5 and 0.75. Overall, we obtained 9 different experimental scenarios. The node type similarity cutoff threshold has been set to 0.6.

For the evaluation we performed the 15 queries in all the 9 scenarios and we calculated the average values for the quality measures. Each scenario has been evaluated considering the quality of the produced rankings (w.r.t. the manual assessment) according to the Spearman's Rank Correlation Coefficient and the Discounted Cumulative Gain (DCG).

The **Spearman's Rank Correlation Coefficient** [17] is a non-parametric correlation coefficient typically used in information retrieval to measure the correlation between ranks (i.e., synthetic linear relationships imposed over non-linearly associated variables), as:

Table 2. Spearman's coefficient (mean and std on 3 experiments) for the 15 queries

λ	*Size*	**Small queries**					**Medium queries**					**Large queries**				
	Query	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**	**Q15**
0.25	*mean(ρ)*	0.58	0.71	0.51	0.76	0.49	0.48	0.56	0.34	0.47	0.44	0.60	0.63	0.58	0.33	0.45
	std(ρ)	0.10	0.12	0.06	0.01	0.10	0.14	0.04	0.05	0.22	0.28	0.09	0.14	0.18	0.31	0.29
0.5	*mean(ρ)*	0.60	0.78	0.57	0.77	0.53	0.51	0.57	0.38	0.51	0.43	0.62	0.69	0.62	0.38	0.46
	std(ρ)	0.08	0.00	0.04	0.00	0.09	0.11	0.07	0.19	0.15	0.25	0.05	0.04	0.07	0.22	0.24
0.75	*mean(ρ)*	0.64	0.78	0.58	0.81	0.59	0.57	0.66	0.39	0.65	0.72	0.64	0.72	0.69	0.47	0.81
	std(ρ)	0.01	0.00	0.04	0.00	0.02	0.00	0.01	0.21	0.06	0.14	0.05	0.05	0.10	0.24	0.05

$$\rho = 1 - 6 \cdot \frac{\sum_{i=1}^{n} d_i^2}{n \cdot (n^2 - 1)} \tag{3}$$

where d_i is the difference (in terms of number of positions) of two items in the two compared ranks. It has values in the range of [-1,1], where: -1 corresponds to perfect negative correlation, 1 to perfect positive correlation, and 0 represents no correlation between the variables. In our case, we calculate the correlation between the results produced by our algorithm and the ground truth built manually. The more these are correlated, the best is the quality of the algorithm. Hence, values closer to 1 demonstrate better performances.

The **Discounted Cumulative Gain (DCG)** [6] is a well-known metrics for assessing the relevance of search result sets, which measures the usefulness (gain) of a document based on its relevance w.r.t. the query and its position in the result list. DCG computes the cumulated gain by introducing a logarithmic discount factor based on the rank position i of the retrieved item as follows:

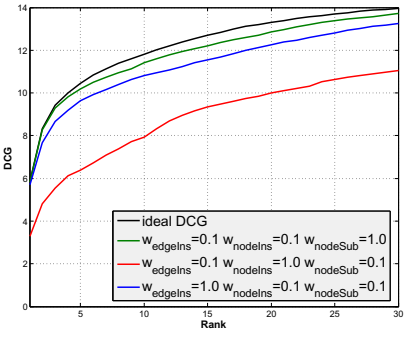
$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(1+i)} \tag{4}$$

where rel_i is the relevance for the i-th rank position.

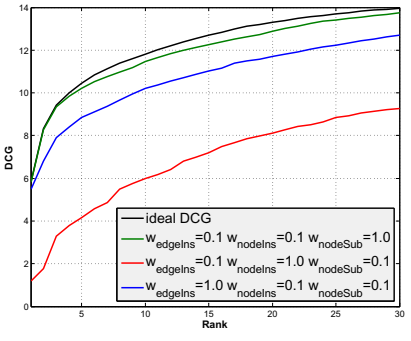
Table 1 shows the different parameter combinations and the Spearman's coefficient results. The table shows that the mean Spearman's correlation coefficient between the ground truth and the calculated results is good for all the experiment configurations (with low standard deviation), thus showing good retrieval performance. For all three experiment configurations, the best performance is obtained when the maximal weight is given for node substitution. As λ increases, the difference in the results among the experiments decreases.

Table 2 delves into the detailed behaviour of the 15 different queries (averaged on the 3 experiments). As λ increases, the performance of the algorithm improves, since the mean value for the Spearman's correlation coefficient increases, and the standard deviation decreases, independently from the query size.

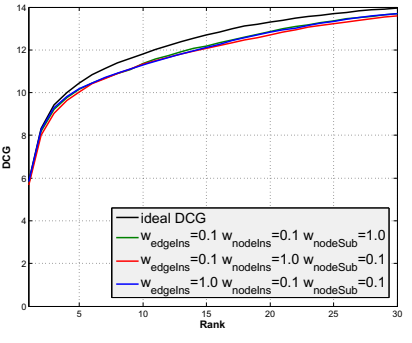
These results are compatible with the DCG analysis, that has been performed to evaluate the perceivable quality of the result sets. Figure 5 shows the behaviour of DCG in the ideal manual assessment and the ones obtained by our



(a) lambda=0.5



(b) lambda=0.25



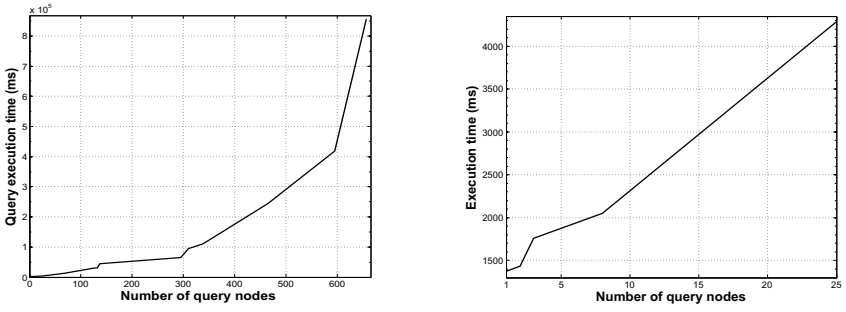
(c) lambda=0.75

Fig. 5. DCG for $\lambda = 0.5$ (a) $\lambda = 0.25$ (b) and $\lambda = 0.75$ (c)

algorithm depending on the different weight values. The algorithm generally performs well, since the values for all the experiment settings are close to the ideal DCG curve corresponding to the manual assessment. This means that the algorithm is able to retrieve relevant projects at highly ranked positions.

The best result is obtained with $w_{nodeSub} = 1$, i.e., when the substituted nodes are accounted as more important in the overall similarity comparison; this is aligned with the intuition that similar (i.e., substituted) nodes are more significant than inserted nodes and edges in the calculation of the graph similarity. Inserted edges ($w_{edgeIns} = 1$) are deemed as slightly less significant, while inserted nodes ($w_{nodeIns} = 1$) behave even worse.

The DCG analysis on λ assessed that higher values of λ imply that the DCG curves tend to get closer, as shown in Figure 5. This means that the role of the three weights becomes less and less important when λ increases. Once more, this is intuitive since higher λ values represent the fact that text distance between nodes is assumed as more relevant while type distance (and the three weights together with it) are considered less important. In any case, the experiment with with $w_{nodeSub} = 1$ is consistently the best one independently on the value of λ.



(a) (b)

Fig. 6. Query execution time w.r.t. query size(a) and detail for small queries(b)

5.2 Performance Analysis

Figure 6 (a) shows the query execution time for all the 15 queries considered in the experiments with respect to the query size, i.e. number of nodes in the query. The image highlights the exponential behavior of the algorithm, as expected given the complexity analysis in Section 4. However, while this is interesting from a theoretical viewpoint, the exponential complexity is not so crucial when dealing with queries of reasonable size (that can be defined in terms up to 50 nodes or so). Indeed, the behavior in this scenario can be assimilated to linear, as Figure 6 (b) shows, granting results query execution time in the order of 1 to 5 seconds. Notice that no query execution optimization (including optimized indexing of the repository) has been adopted during experiments and therefore we foresee a wide range of possibility for improving the performance of the system.

6 Conclusions

In this paper we presented a graph-based approach for content-based search of models using the A* algorithm. We evaluated our approach upon a realistic setting, obtaining good quality of results with respect to a manually assessed ground truth, together with a clear profiling of the parameters behavior.

Future work includes: increasing the size of the project repository and of the ground truth; comparing our graph search solution with keyword based ones [1]; studying techniques for automatically tuning the parameters of the A* algorithm, such as e.g. neural networks or genetic algorithms ; and experimenting the approach with other metamodels, such as e.g. BPMN.

References

1. Bozzon, A., Brambilla, M., Fraternali, P.: Searching Repositories of Web Application Models. In: International Conference on Web Engineering, pp. 1–15 (2010)
2. Brügger, A., Bunke, H., Dickinson, P., Riesen, K.: Generalized Graph Matching for Data Mining and Information Retrieval. Advances in Data Mining. Medical Applications, E-Commerce, Marketing, and Theoretical Aspects, 298–312

3. Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S., Matera, M.: Morgan Kaufmann series in data management systems: Designing data-intensive Web applications. Morgan Kaufmann Pub., San Francisco (2003)
4. Dijkman, R.M., Dumas, M., van Dongen, B.F., Käärik, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. Inf. Syst. 36(2), 498–516 (2011)
5. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics 4(2), 100–107 (1968)
6. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Transactions on Information Systems (TOIS) 20(4), 422–446 (2002)
7. Kunze, M., Weske, M.: Metric trees for efficient similarity search in large process model repositories. In: Proceedings of the 1st International Workshop Process in the Large (IW-PL 2010), Hoboken, NJ (September 2010)
8. Lucrédio, D., Fortes, R.d.M., Whittle, J.: MOOGLE: A model search engine. Model Driven Engineering Languages and Systems, 296–310 (2010)
9. Madhavan, J., Bernstein, P.A., Rahm, E.: Generic schema matching with cupid. In: Proceedings of the International Conference on Very Large Data Bases, Citeseer, pp. 49–58 (2001)
10. Markovic, I., Pereira, A.C., Stojanovic, N.: A framework for querying in business process modelling. In: Proceedings of the Multikonferenz Wirtschaftsinformatik (MKWI), Munchen, Germany (2008)
11. Messmer, B.: Efficient Graph Matching Algorithms for Preprocessed Model Graphs. PhD thesis, University of Bern, Switzerland (1996)
12. Nandi, A., Bernstein, P.A.: HAMSTER: using search clicklogs for schema and taxonomy matching. In: Proceedings of the VLDB Endowment, vol. 2(1), pp. 181–192 (2009)
13. Nowick, E.A., Eskridge, K.M., Travnicek, D.A., Chen, X., Li, J.: A model search engine based on cluster analysis of user search terms. Library Philosophy and Practice 7(2) (2005)
14. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. The VLDB Journal 10(4), 334–350 (2001)
15. Remco Dijkman, M.D., Garcıa-Banuelos, L.: Graph matching algorithms for business process model similarity search. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) BPM 2009. LNCS, vol. 5701, pp. 48–63. Springer, Heidelberg (2009)
16. Smith, K., Bonaceto, C., Wolf, C., Yost, B., Morse, M., Mork, P., Burdick, D.: Exploring schema similarity at multiple resolutions. In: Proceedings of the 2010 International Conference on Management of Data, pp. 1179–1182. ACM, New York (2010)
17. Spearman, C.: The proof and measurement of association between two things. The American Journal of Psychology 100(3-4), 441 (1904)
18. Syeda-Mahmood, T., Shah, G., Akkiraju, R., Ivan, A.A., Goodwin, R.: Searching service repositories by combining semantic and ontological matching
19. Tian, Y., Patel, J.M.: Tale: A tool for approximate large graph matching. In: International Conference on Data Engineering, pp. 963–972 (2008)