UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

APPROXIMATE SEMANTIC TREE MATCHING IN OPENKNOWLEDGE

Fausto Giunchiglia, Fiona McNeill and Mikalai Yatskevich

October 2006

Technical Report # DIT-07-031

OpenKnowledge Deliverable D4.1

Approximate Semantic Tree Matching in OpenKnowledge

Fausto Giunchiglia, Fiona McNeill and Mikalai Yatskevich
with input from
Zharko Alekovski, Alan Bundy, Frank van Harmelen, Spyros Kotoulas, Vanessa Lopez, Marta Sabou, Ronny Siebes and Annette ten Tejie

October 27, 2006

1 Introduction

The OpenKnowledge (OK) project is predicated on the idea that the peers involved in the peer-to-peer network do not have to have prior agreement on the representation and vocabulary they use or conform to pre-established standards. Instead, peers are permitted to represent their knowledge in any way they choose and the interaction between these disparate peers is controlled via interaction models(IMs), which describe how specific interactions proceed. These IMs describe the order of the message passing, the messages to be passed and the constraints on those messages for every peer in the interaction and are written in LCC; see Figure 2 for an example. Naturally, since representation and vocabulary are not fixed, identifying, interpreting and communicating via these IMs requires complex matching techniques. In the basic case, this matching can be done by hand, with users choosing IMs that fit with their knowledge representation or developing bridges that link the representation in the IM with their own representation. However, this puts an onerous burden on users; the system can only be usable on a large-scale if these matchings can be made automatically or interactively.

This document is intended to outline where the need for approximation occurs in OK and how appropriate approximations can be discovered. Once approximations have been discovered, we also need to be able to evaluate how good they are. The complex issue of how we evaluate approximations to decide how good they are and then match this against some notion of 'good enough', and how interactions with the user can be used to guide this, is introduced in this document but will be discussed in detail in the following deliverable.

Approximations are necessary where matching needs are discovered and cannot be met with perfect matches. We therefore briefly describe the matching needs of OK and then go on to describe how matching would be done in these cases, assuming a perfect match

can be found, before discussing how we deal with failure to find these perfect matches through introducing approximate matching and methods for evaluating how good they are.

At this time, we consider only the matching requirements of a single peer once an IM has been found for the interaction, which can be divided into three different areas:

- finding appropriate roles to play through interpreting role identifiers (which may be determining which (if any) of the roles it wishes to play in the IM it has chosen to invoke or, for a peer that is requested to join the interaction, deciding whether it wishes to play the role it is invited to play);
- interpreting messages;
- fulfilling constraints on the messages to be passed.

The document is arranged as follows. Section 2 describes where the matching needs arise in an OK interaction and what we can determine about the objects that needs to be matched. Section 3 outlines a motivating scenario in which matching is necessary. Section 4 describes how the context of the interaction is exploited to provide more information to the matching process: determining more about what the interaction is doing can help determine what objects within the interaction might mean, so this information is processed before the matching process begins. Sections 5 and 6 describe how structured objects are matched against one another either exactly or approximately, assuming that matching at node level can be done. Sections 7 and 8 describe how this node matching is done, either exactly or approximately. Section 9 discusses the notions of local and global good enough answers and how we intend to approach these issues, and summarises the paper and draws conclusions.

2 Matching Needs in OpenKnowledge

We describe the precise scenario in which the matching will take place, explaining the basic OK system and describing when and why these may have to be approximate. Since interactions are conducted according to message passing defined by the IMs, which are shared by all the peers taking part in the interaction, there is no possibility of mismatches in the ordering or types of messages sent, and thus we do not need to consider the matching of complete IMs.

We identify three areas in which mismatches could occur:

1. **Role identifiers**: these are usually atomic but may have some number of arguments (concerning information that the role must carry). These indicate to the peer whether the role is likely to be appropriate;
2. **Constraints**: these give conditions on and meaning to the messages. A peer cannot fulfil a role unless it can interpret and satisfy the constraints on the messages it must send within that role. These are first-order terms;

3. **Message contents**: the contents of messages are determined to a large extent by the IMs and the constrains; however, even with this information, peers cannot predict how other peers will instantiate variables in messages and must be able to interpret these themselves during run-time.

A key aspect of the matching in OK is that these areas of mismatch (with the exception of some role identifiers) have structure, and thus matching must both deal with atomic semantic mismatches (what the terms mean) and with structural mismatches (how these atomic terms are combined). In this paper, we describe the precise problem in more detail, discuss existing techniques for these problems and introduce a technique for combining known semantic and structural matching techniques to produce a *semantic tree matching* technique. We then go on to detail how these can form approximate matchings between structured terms if no precise matchings can be found. This is a new approach to the problem of mismatches in knowledge bases with structure as well as semantics, and may prove to be important not only to the OpenKnowledge project but also to a much broader domain, particularly database querying.

The techniques we describe are applicable to first-order or atomic structures, and can thus deal with mismatches in role identifiers, constraints and some kinds of message contents. However, if message contents are expressed according to some other kind of representation, such as OWL, these techniques would not, as they stand, be applicable. Further investigation into what kinds of message contents representations the system should be able to deal with and how these techniques should be extended to allow for that will be necessary. The focus on this document is on matching that allows peers to interpret IMs and not on the messages that are passed as part of these interactions.

We assume that every peer has the ability to perform matching to suit its own requirements because the matching component is downloaded and installed, together with the query routing, visualisation and interpreter components, when initially setting up as an OK peer. This avoids the problem of forced sharing of information: if matching were necessarily done by a specialised peer, this would require peers with matching needs to reveal their full knowledge base to that peer. Nevertheless, it may be the case that certain peers are able to provide matching assistance, through a knowledge of previous matchings that have been used to peers that require such assistance.

3 Motivating scenario

Consider the situation in which a user wishes to purchase a particular item. The user's peer has already participated in a buying role and thus has an interaction model describing how this is done, illustrated in Figure 2. Each user can organise his information about IMs as a classification that gives context to the meaning of the IM. For example, the classification of the user may be as illustrated in Figure 1. The particular buying IM that the user intends to use is the one found under `car`. Notice that there is also another buying IM in the user's classification found under `truck`. It may be that these IMs are different: the user has different considerations to consider when buying a car than when buying a truck. However, it may be that these two IMs are the same. A user could have many copies of the same IM that are stored in different categories. The purpose of this

is that information can be derived from the context of the IM in the classification: the fact that the IM is intended for buying cars as a means of transport can have a material effect on the matching process. This is discussed in Section 4.

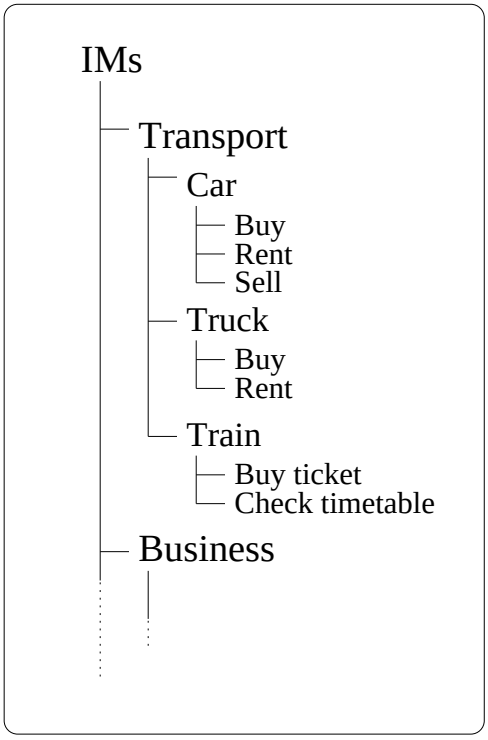


Figure 1: The user's IM classification

The user chooses this IM and then propagates a request for peers to play the roles it. The *buying* role will presumably be played by a peer owned by the user as the user wishes to buy the goods for himself. In this case, there will be no constraint matching problems for the buying peer, as the IM is one that the user, and hence its buying peer, has used before and the matching issues will therefore already have been dealt with. However, any peer that wishes to play the selling agent will have to ensure that it is able to satisfy the constraints, and thus must also be able to interpret the constraints by mapping them to terms in its own knowledge base. Each peer must return a score of how well it is able to play the role so that it can be decided which peer to initiate the interaction with; this score is derived from the quality of mappings between the constraints of the IM and the peer's knowledge base. If the peer already knows these constraints, the score will be perfect.

Consider a peer that wishes to play the shopkeeper role. It already has experience at playing a shopkeeper and therefore has encountered constraints that pertain to a shop-keeping role. These may (or may not) be useful in interpreting the constraints in this new IM, and additionally any other constraints that the peer has encountered in previous (thematically non-related) interactions could be pertinent. Figure 3 shows an IM that this peer has participated in before, the constraints of which it therefore can interpret. When

the peer is trying to interpret the constraints shown in Figure 2, these will be potential candidates for mapping.

```
a(buyer,B) ::
    ask(X) => a(shopkeeper,S) <- require(X) and shop(S) then
    price(X,P) <= a(shopkeeper,S) then
    buy(X,P) => a(shopkeeper,S) <- spare(money(P,C),X) then
    sold(X,P) <= a(shopkeeper,S) -> has(X)


a(storemanager,S) ::
    ask(X) <= a(buyer,B) then
    price(X,P) => a(buyer,B) <- carry(X,P) then
    buy(X,P) <= a(buyer,B) then
    sold(X,P) => a(buyer,B)
```

Figure 2: The buyer's IM

```
a(purchaser,B) ::
    ask(X) => a(shopkeeper,S) <- need(X) and shop(S) ther
    price(X,P) <= a(shopkeeper,S) then
    buy(X,P) => a(shopkeeper,S) <- afford(X,P) then
    sold(X,P) <= a(shopkeeper,S) -> have(X,S)


a(shopkeeper,S) ::
    ask(X) <= a(buyer,B) then
    price(X,P) => a(buyer,B) <- stock(N,P,X) then
    buy(X,P) <= a(buyer,B) then
    sold(X,P) => a(buyer,B)
```

Figure 3: The seller's IM

Before the shopkeeper peer can appropriately participate in this interaction, it must first check that it is happy with the role descriptor (*storemanager*) and then ensure that it can interpret the constraints in the interaction model. Any constraints that it cannot interpret it will certainly not be able to satisfy, and hence, if such exist, there is no point commencing the interaction.

The shopkeeper peer must first decide whether it wishes to play the role *storemanager* in place of *shopkeeper*. It can use the available node matching techniques to determine that *storemanager* is a synonym of *shopkeeper*, and is thus content to take on this role. The role name gives no definite information about what the peer will have to do in the interaction but rather gives a general idea so that the peer can decide whether to go to the expense of generating potential mappings.

In the IM to be used for the interaction (Figure 2), there is only one constraint to be satisfied in playing the *storemanager* role: $carry(X, P)$. IMs allow information to

be attached to the objects in them, so that it is possible, for example, to attach class information to the arguments of constraints. The information attached to this constraint is that the variable X is of type *item* and the variable P is of type *price*. The role of *shopkeeper*, described in Figure 3, which the peer has played before, also required one constraint to be fulfilled: $stock(?Number, ?Item, ?Price)$ (with class information included). The peer can therefore use this constraint (along with any other constraints it may know) to interpret the unknown constraint by attempting to map them. There constraints on the buying role do not need to be interpreted by the shopkeeping agent; the fact the description of buying in its original IM and in the new IM are different is not of direct concern to the shopkeeping agent, as it only needs to interpret constraints that it needs to fulfil as part of its role. Additionally, the buying agent will not be concerned with mapping these constraints to one another as it is not concerned with (and is unlikely to have access to) the buying description in Figure 3; it only needs to fulfil the constraints in the chosen IM, which it is already able to do, since it has used this IM before.

The matching process must find semantic links between the elements of the two constraints to be mapped: for example, WordNet lists *carry* as a synonym of *stock* and the types $Item$ and $Price$ in the shopkeeper's original IM are the same as the types $Item$ and $Price$ in the buyer's IM. Semantic tree matching is then employed to use this element-level matching information to find a map that takes account of the structure. The 'good enough' measures employed during this matching process then give a score reflecting the quality of the match between these two constraints. This score allows the shopkeeping peer to determine whether this map is the best it can find between the constraint it needs to fulfil and its known constraints. Once the peer has found its best possible scores for all the constraints it must map (in this case, only one), this information is fed back to the process so it can be determined whether our shopkeeping peer is better at performing the role than any other peers that wish to take part and also to ensure that the highest scoring peer meets some 'good enough' threshold; if not, the interaction will not proceed as it is unlikely that it could be performed satisfactorily.

4 Linguistic preprocessing

The goal of the linguistic preprocessing step is to automatically translate ambiguous natural language labels taken from the term tree elements into an internal logical language. We use a propositional description logic language (L^C) for several reasons. First, given its set-theoretic interpretation, it "maps" naturally to the real world semantics. Second, natural language labels, e.g., constant and type names in term tree, are usually short expressions or phrases having simple structure. These phrases can often be converted into a formula in L^C with no or little loss in the meaning [7]. Third, a formula in L^C can be converted into an equivalent formula in a propositional logic language with boolean semantics. Apart from the atomic propositions, the language L^C includes logical operators, such as conjunction ($\sqcap$), disjunction ($\sqcup$), and negation ($\neg$). There are also comparison operators, namely more general ($\sqsupseteq$), less general ($\sqsubseteq$), and equivalence ($=$). The interpretation of these operators is the standard set-theoretic interpretation.

We define the notions of concept of label and concept at node similarly to semantic

matching [6]:

- Concept of a label, which denotes the set of documents (data instances) that one would classify under a label it encodes;
- Concept at a node, which denotes the set of documents (data instances) that one would classify under a node, given that it has a certain label and that it is in a certain position in term tree and position of the term interaction model in IM classification.

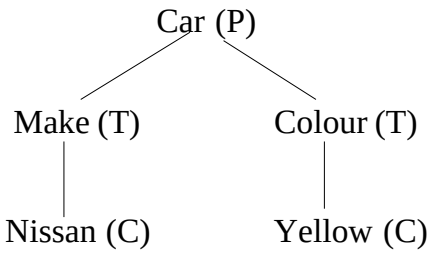
Notice that, as opposed to semantic matching, the concept of node depends not only on its position in the term tree but also on the position of the IM in IM classification (see Figure 1 for example).

Concepts of labels are computed in the following four macro steps:

1. Tokenization. Labels of nodes are parsed into tokens by a tokenizer which recognises punctuation, cases, digits, stop characters, etc. Thus, for instance, the label *Types of payment* becomes *types*, *of*, *payment*.
2. Lemmatization. Tokens of labels are further lemmatised, that is, they are morphologically analysed in order to find all their possible basic forms. Thus, for instance, *types* is associated with its singular form, *type*.
3. Building atomic concepts. An Oracle (such as WordNet) is queried to obtain the senses of lemmas identified during the previous phase. For example, the label *types* has the only one token *types*, and one lemma *type*. From WordNet we find out that type has six senses as a noun and two as a verb.
4. Building complex concepts. When existing, all tokens that are prepositions, punctuation marks, conjunctions (or strings with similar roles) are translated into logical connectives and used to build complex concepts out of the atomic concepts constructed in the previous phase. Thus, for instance, commas and conjunctions are translated into logical disjunctions, prepositions, such as *of* and *in*, are translated into logical conjunctions, and words like *except*, *without* are translated into negations. Thus, for example, the concept of label *Types of payment* is computed as $C_{Typesofpayment} = C_{Types} \sqcap C_{payment}$, where $C_{Types} = \langle type, senses_{WN}\#8 \rangle$ is taken to be the union of eight WordNet senses, and similarly for *payment*. Notice that natural language and is converted into logical disjunction, rather than into conjunction.

As the result the logical formula for concept of label is computed as a full propositional formula were literals stand for atomic concepts of labels. Concepts at nodes are written in the same propositional description logic language as concepts of labels. Term trees are hierarchical structures where the path from the root to a node uniquely identifies that node (and also its meaning). We define the logical formula for a concept at node as a conjunction of concepts of labels located in the path from the given node to the root. For example, in the tree A, the concept at node four is computed as follows: $C_4 = C_{buy} \sqcap C_{type} \sqcap C_{payment} \sqcap C_{cash}$.

In order to constrain the set of possible concept at node interpretations the sense filtering techniques are used. The main goal of sense filtering techniques is to filter out



P = predicate; T = tree; C = constant

Figure 4: Constraint $car(nissan, yellow)$ expressed as a tree

irrelevant (for the given matching task) Oracle senses from concepts of labels. For all concepts of labels we collect all their ancestors and descendants. We call them a focus set. Notice that interaction model itself can be classified in tree like structure (see Figure 4 for example). Therefore the focus set is enriched with concept of labels of the IM and its ancestors in the IM classification. Then, all Oracle senses of atomic concepts of labels from the focus set are compared with the senses of the atomic concepts of labels of the concept. If a sense of atomic concept of label is connected by an Oracle relation with the sense taken from the focus set, then all other senses of these atomic concepts of labels are discarded. Therefore, as a result of the sense filtering step we have (i) the Oracle senses which are connected with any other Oracle senses in the focus set or (ii) all the Oracle senses otherwise. After this step the meaning of concept of labels is reconciled with respect to the knowledge residing in the tree structure.

5 Exact Semantic Tree Matching

The simplest case that the matching component has to deal with is the case of discovering an exact match: for example, matching *need(?Item)* to *require(?Item)*. In this section, we describe the issues involved in exact matching and the approach taken by the matching process, which will then be extended in section 6 to deal with approximate matching.

In order to match first-order terms, we consider them as trees. LCC allows type information to be given about arguments of constraints, though we cannot always rely on designers providing this information and must be able to continue with matching even if type information is missing (though naturally this will make the matching less reliable). A constraint such as $car(nissan, yellow)$, with type information stating that the first argument was of type *make* and the second was of type *colour*, would be represented as shown in Figure 4.

We assume that constraints in LCC are written in CNF; role descriptors and message contents appear individually. In order to satisfy a set of constraints on a message in an IM, it is necessary, in the simple case, to find a particular known constraint that can be used to interpret, through matching, and then satisfy at least one of every disjunction in the CNF. In a more complex situation, we might match a single constraint to many constraints, or match many-to-many; however, we only consider one-to-one matching in

this document.

First-order terms are represented as trees and then matched against each other. Particular attention must be paid to the structure of these trees. Horizontal ordering within the trees is not important: a reordering of arguments of a predicates, for example, does not constitute a mismatch. However, vertical ordering is key and must be considered by the matching process. Information that is helpful for structural matching can be gleaned from the structural context of each node: the path between that node and the root node.

At each node, semantic node matching must be done to determine which node in the matching tree it corresponds to, and, when extended to deal with approximate mapping, how strong this correspondence is. Semantic structure matching is thus the combination of the results of semantic node matching in such a way as to respect the structure of the tree to be matched.

In order for this matching process to be possible, each peer must have a set of known constraints to which these IM constraints (or role descriptors or message contents) can be matched. It can be assumed that any peer will have some notion of how to play some role according some particular IM. Additionally, it may have been asked at some point to play that or a different role according to a new IM, in the course of which it will have encountered new constraints that it will have to have interpreted. These known constraints are what the matching process must refer to. Initially, when the peer knows of very few constraints, this process may fail quite often - though if the peer is performing a role similar to the one it intended to perform, many of the constraints may already have parallels in its knowledge base. It would be helpful to have some kind of user interaction process to assist the user to build new constraints into its knowledge base in the case where there is nothing already in the knowledge base that can be matched to. As the peer becomes more experienced, more constraints will be known about, so matching will be possible in more situations.

In the IM, the constraints have a structure: they are in CNF. The known constraints may also have a structure because these will often be constraints from some other IM. However, the structure of these known constraints is not relevant and can be discarded in the matching process. Therefore, each conjunction of disjunctions is mapped individually, with one constraint found to map to one of the disjunctions. In practice, constraints will most often be simple conjunctions, in which case every conjunct must be mapped to a known constraint.

The above discussion assumes that it is possible to perform one-to-one matching on constraints. This will not always be the case. We may find that one constraint is more appropriately covered through matching to two or more constraints, or that several constraints are correctly interpreted through mapping to several other constraints, but that this mapping cannot be broken down into one-to-one mappings. This matching process is rather more difficult and is not considered in this document, which focuses on the situation where a one-to-one correspondence can be found between constraints.

Definition 1 *(Semantic Node Matching) Let n_1 and n_2 be two nodes in two trees T_1 and T_2. We say that n_1 and n_2 (semantically) match iff $c@n_1$ iff $c@n_2$ holds given the available background knowledge, where $c@n_1$ and $c@n_2$ are the concepts at nodes of n_1 and n_2 respectively.*

Definition 2 *(Semantic tree matching) Let T_1 and T_2 be two trees. We say that T_1 and T_2 match iff for any node n_11 in T_1 there is a node n_21 in T_2 such that*

- *n_11 semantically matches n_21;*
- *n_11 and n_21 reside on the same depth in T_1 and T_2 respectively;*
- *all ancestors of n_11 are semantically matched to the ancestors of n_21;*

At this stage, we assume that the problem of semantic node matching has been dealt with and can be called as a subprocess of the semantic structure matching, the details of which are discussed in Section 7.

The following pseudo code illustrates an algorithm for exact structure matching:

```
1.Node struct of
2.  int nodeId;
3.  String label;
4.  String cLabel;
5.  String cNode;
6.MappingElement struct of
7.  int MappingElementId;
8.  Node source;
9.  Node target;
10. String relation;

11.MappingElement[] exactStructureMatch(Tree of Nodes source, target)
12. MappingElement[] result;
13. exactTreeMatch(source,target,result);
14. if (allNodesMapped(source,target,result))
15.  return result;
16. else
17.  return null;

18.void exactTreeMatch(Tree of Nodes source,target,MappingElement[] result)
19. Node sourceRoot=getRoot(source);
20. Node targetRoot=getRoot(target);
21. String relation= nodeMatch(sourceRoot,targetRoot);
22. if (relation=="=")
23.  addMapping(result,sourceRoot,targetRoot,"=");
24. Node[] sourceChildren=getChildren(sourceRoot);
25. Node[] targetChildren=getChildren(targetRoot);
26. For each sourceChild in sourceChildren
27.  Tree of Nodes sourceChildSubTree=getSubTree(sourceChild);
28.  For each targetNode in target
29.   Tree of Nodes targetChildSubTree=getSubTree(targetChild);
30.   exactTreeMatch(sourceChildSubTree, targetChildSubTree, nodesToMatch);
```

exactStructureMatch takes two trees of nodes (i.e., tree representation of LCC terms) *source* and *target* as an input. Here and throughout the paper we assume that the source tree is derived from an IM constraint and the target tree represents the term derived from the peer capability description. **exactStructureMatch** returns an array of mappings holding between the nodes of the trees if there is an exact match between

them and null otherwise. Array of MappingElements *result* is created (line 12) and filled by **exactTreeMatch** (line 13).

allNodesMapped checks whether all the nodes of source tree are mapped to the nodes of the target tree (line 14). If this is the case there is an exact structure match between the trees and the set of computed mappings is returned (line 15).
exactTreeMatch takes two trees of nodes (i.e., tree representation of LCC terms) *source* and *target* and array of MappingElements *result* as an input. It recursively fills *result* with the mappings computed by **nodeMatch** (line 23). **exactTreeMatch** starts from obtaining the roots of *source* and *target* trees (lines 19-20). The semantic relation holding between them is computed by **nodeMatch** (line 21) implementing the node matching algorithm. If the relation is equivalence, the corresponding mapping is saved to *result* array (lines 22-23) and the children of the root nodes are obtained (line 24-25). Finally the loops on *sourceChildren* and *targetChildren* (lines 26-30) allow to call **exactTreeMatch** recursively for all pairs of sub trees rooted at *sourceChildren* and *targetChildren* elements.

The above algorithm is designed to succeed for equivalent terms and to fail otherwise. It expects the trees to have the same depth and for all matching nodes to have the same number of children. When we come to consider approximate matching, we can no longer make such assumptions. At the same time the algorithm is not sensitive to the ordering of nodes considered to be equivalent by semantic node matching, e.g., *car(Nissan,navy)* is considered to be equivalent to *auto(dark blue, Nissan)* if node matching returns *car=auto* and *navy=dark blue*.

6 Approximate Semantic Tree Matching

Taking into account the heterogeneous nature of P2P network it is unlikely that most of constraints in IMs can be exactly matched with the peer capabilities. Hence approximate tree matching is a necessary requirement for OK system.

Definition 3 *(Approximate node matching) Let n_1 and n_2 be two nodes it two trees T_1 and T_2. We say that n_1 and n_2 approximately match iff $c@n_1$ R $c@n_2$ holds given the available background knowledge, where $c@n_1$ and $c@n_2$ are the concepts at nodes of n_1 and n_2 respectively, and where $R \in \{\equiv, \subseteq, \supseteq, \wedge, \perp, ?\}$.*

Definition 4 *(Approximate tree matching) Let T_1 and T_2 be two trees. We say that T_1 and T_2 match iff there is at least one node $n_1 1$ in T_1 and a node $n_2 1$ in T_2 such that*

- *$n_1 1$ approximately matches $n_2 1$;*
- *all ancestors of $n_1 1$ are approximately matched to the ancestors of $n_2 1$;*

Here and throughout the paper we distinguish between two dimensions of approximation:

- Model based approximation (e.g., *vehicle* is considered to be approximate answer on *car* query given that $car \sqsubseteq vehicle$ according to the matching process)

- Probability based approximation (e.g., *propel* is considered to be approximate answer on *proper* query given that *propel* = *proper* with 0.8 similarity (often treated as probability) according to string edit distance matcher)

In this and in the following section we concentrate on model based approximation. Probability based approximations are discussed in section 8 devoted to element level matchers.

The following pseudo code illustrates approximate tree matching algorithm.

```
1.MappingElement[] approximateStructureMatch(Tree of Nodes source, target, double threshold)
2. MappingElement[] result;
3. approximateTreeMatch(source,target,result);
4. double approximationScore=analyzeMismatches(source,target,result);
5. if (approximationScore>threshold)
6.  return result;
7. else
8.  return null;

9. void approximateTreeMatch(Tree of Nodes source,target,MappingElement[] result)
10. Node sourceRoot=getRoot(source);
11. Node targetRoot=getRoot(target);
12. String relation= nodeMatch(sourceRoot,targetRoot);
13. if (relation!="Idk")
14.  addMapping(result,sourceRoot,targetRoot,relation);
15. Node[] sourceChildren=getChildren(sourceRoot);
16. Node[] targetChildren=getChildren(targetRoot);
17. For each sourceChild in sourceChildren
18.  Tree of Nodes sourceChildSubTree=getSubTree(sourceChild);
19.  For each targetNode in target
20.   Tree of Nodes targetChildSubTree=getSubTree(targetChild);
21.   approximateTreeMatch(sourceChildSubTree, targetChildSubTree, nodesToMatch);
```

In contrast to **exactStructureMatch**, **approximateStructureMatch** takes as an input not only *source* and *target* term trees but also *threshold* allowing to select highly similar term trees. **approximateTreeMatch** fills *result* array (line 3) which stores the mappings holding between nodes of the trees. *approximationScore* is computed (line 4) by **analyzeMismatches** which decides the importance of *source* tree nodes mismatches (if any) and calculates the aggregate score of tree match quality. If *approximationScore* exceeds *threshold* the mappings calculated by **approximateTreeMatch** are returned (line 6).
In contrast to **exactTreeMatch**, **approximateTreeMatch** also considers semantic relations other then equivalence (line 13) and stores them in *result* array (line 14).

The example of approximately matched term trees is given on Figure 2. The trees on the figure are partially matched. For example the nodes brand, Nissan, type of payment, cash in the source tree are not matched to any node of the target tree. The importance of the mismatched nodes is determined by **analyzeMismatches** routine. For example assuming that all nodes of the tree are equivalently important and 4 out of 11 nodes of the tree are not matched the approximation score can be 1-4/11= 7/11.

Note that approximate semantic tree matchings can be found between any two trees for which even a single node from each has some kind of approximate semantic match; thus finding an approximate semantic tree match between two terms is not a very strong

result. The techniques described in this section provide a basis for which to add the 'good enough' measures that will be discussed later, and it is the combination of these two approaches that provide powerful matching techniques.

7 Approximate Node Matching

The node matching algorithm takes two nodes of the term trees as an input and produces a semantic relation $\{\equiv, \sqsubseteq, \sqsupseteq, \bot, Idk\}$ holding between them (where Idk stands for *I don't know*. It exploits both the knowledge produced by element level matchers and the knowledge encoded in the term structure. The node matching algorithm is exploited by both exact and approximate semantic tree matching algorithms in order to discover the mappings holding between nodes in the term trees.

7.1 Semantic Node Matching

The semantic node matching algorithm converts the node matching problem into a propositional validity problem. Semantic relations are translated into propositional connectives using the rules described in Table 1 (second column).

Table 1: The relationship between semantic relations and propositional formulas

rel(a,b)	rel(a,b) translation	CNF translation of Eq. 2
$a \equiv b$	$a \leftrightarrow b$	N/A
$a \sqsubseteq b$	$a \rightarrow b$	$axioms \wedge context_A \wedge \neg context_B$
$a \sqsupseteq b$	$a \leftarrow b$	$axioms \wedge context_B \wedge \neg context_A$
$a \bot b$	$\neg(a \wedge b)$	$axioms \wedge context_A \wedge context_B$

The only criterion for determining whether a relation holds between concepts of nodes is whether it is entailed by the premises. Thus, we have to prove that the following formula:

$$(axioms) \rightarrow rel(context_A, context_B) \tag{1}$$

is valid, that is, that it is true for all the truth assignments of all the propositional variables occurring in it. $context_A$, and $context_B$ are the propositional formulas for the concepts at nodes and *axioms* are produced from element level relations connecting atomic concepts of labels in the concepts at nodes. *rel* is the semantic relation that we want to prove holding between $context_A$ and $context_B$. The algorithm checks the validity of Eq. 1 by proving that its negation, i.e., Eq. 2, is unsatisfiable.

$$axioms \wedge \neg rel(context_A, context_B) \tag{2}$$

Table 1 (the third column) summarises how Eq. 2 is translated for the tasks of testing each semantic relation. We report the translated formulas in Conjunctive Normal Form (CNF), i.e., conjunction of disjunctions of propositional atoms, in order to simplify further discussions. Notice that the check for equality is omitted in Table 1, since $A \equiv B$ holds iff $A \sqsubseteq B$ and $A \sqsupseteq B$ hold.

Let us consider the pseudo code of semantic node matching algorithm.

```
1. String nodeMatch(Node source, target)
2. String contextA = getCnodeFormula (sourceNode);
3. String contextB = getCnodeFormula (targetNode);
4. String axioms=mkAxioms(contextA, contextB);
5. formula= And(axioms, contextA, contextB);
6. formulaInCNF=convertToCNF(formula);
7. boolean isOpposite= isUnsatisfiable(formulaInCNF);
8. if (isOpposite)
9.   return "!";
10. String formula=And(axioms,contextA,Not(contextB));
11. String formulaInCNF=convertToCNF(formula);
12. boolean isLG=isUnsatisfiable(formulaInCNF)
13. formula=And(axioms, Not(contextA), contextB);
14. formulaInCNF=convertToCNF(formula);
15. boolean isMG= isUnsatisfiable(formulaInCNF);
16. if (isMG && isLG)
17.   return "=";
18. if (isLG)
19.   return "<";
20. if (isMG)
21.   return ">";
22. return "Idk";
```

nodeMatch takes two *Nodes* as an input and computes a semantic relation holding between them. Firstly, the concept of nodes for *source* and *target* are obtained (line 2-3). Then the axioms holding between atomic concepts of labels in the concepts at nodes are computed by **mkAxioms** (line 4). In line 5, **nodeMatch** constructs the formula for testing disjointness. In line 6, it converts the formula into CNF, while in line 7 it checks the CNF formula for unsatisfiability. If the formula is unsatisfiable, the disjointness relation is returned. The same process is repeated for less and more generality relations. If both relations hold, then the equivalence relation is returned (line 17). If all the tests fail, the Idk relation is returned (line 22).

8 Approximate element level matching

Element level matchers take as an input either atomic concepts defined in the certain Oracle or atomic labels. They produce the semantic relations holding between them. Therefore in the context of OK system element level matchers produce relations holding among labels (and parts of labels) of LCC atomic terms. Notice that element level matchers do not consider the structure of complex terms.

Currently we distinguish between 3 classes of element level matchers:

- *String based matchers* exploit approximate string matching techniques. They are widely used in various schema matching systems [9, 16]. String based matchers presented in this section are modified in order to produce a semantic relation as output. They take as input two labels and produce a semantic relation if they satisfy the given criteria, which is specific for each matcher. Otherwise, Idk is returned.

¡

Table 2: Element level semantic matchers

¡ **Matcher name**	**Matcher type**	**Input info** ¡
Prefix	String based	¡ Labels
Suffix	String based	Labels ¡
Edit Distance	String based	Labels ¡
nGram	String based	Labels ¡
WordNet	Knowledge based	WordNet senses ¡
Leacock Chodorow Matcher	Knowledge based	WordNet senses ¡
Resnik Matcher	Knowledge based	WordNet senses ¡
Jiang Conrath Matcher	Knowledge based	WordNet senses ¡
Lin Matcher	Knowledge based	WordNet senses ¡
Hirst-St.Onge Matcher	Knowledge based	WordNet senses ¡
Context Vectors Matcher	Knowledge based	WordNet senses ¡
WordNet Gloss	Gloss based	WordNet senses ¡
WordNet Extended Gloss	Gloss based	WordNet senses ¡
Gloss Comparison	Gloss based	WordNet senses ¡
Extended Gloss Comparison	Gloss based	WordNet senses ¡
Semantic Gloss Comparison	Gloss based	WordNet senses ¡
MatchMiner	Knowledge based	Labels/WordNet senses ¡
PowerMap	Knowledge based	Labels/WordNet senses ¡
¡		

¡ ¡

- *Knowledge based matchers* take as input two concept (or synset) identifiers defined in an external Oracle such as WordNet. They produce semantic relations by exploiting the structural properties of the Oracle. In some cases they combine the knowledge derived from the Oracle with statistics collected from large scale corpora. Often knowledge based matchers are based on either similarity or relatedness measures. If the value of the measure exceeds the given threshold the certain semantic relation is produced. Otherwise Idk is returned.

- *Gloss based matchers*, similarly to knowledge based matchers, take two concept (synset) identifiers as input and return the semantic relation holding between them. However, gloss based matchers differ in that they use the information contained in natural language concept descriptions such as WordNet glosses.

Element level matchers are organised in the library. Table 2 presents currently available element level matchers.

¡

Currently the matchers in the library are executed sequentially in the order presented in Table 2 until the semantic relation other then Idk is produced. The element level matchers library is exploited in semantic node matching algorithm. The pseudo code illustrating the usage of element level matchers library is presented below.

```
1.String mkAxioms(String contextA, contextB);
2. String[] matchers;
```

```
3. String axioms, relation;
4. matchers=getMatchers();
5. for each sourceAtomicConceptOfLabel in contextA
6.  for each targetAtomicConceptOfLabel in contextB
7.   relation=getRelation(matchers,
        sourceAtomicConceptOfLabel,targetAtomicConceptOfLabel);
8.   addToAxioms(axioms,relation);
9. return axioms;

10.String getRelation(String[] matchers,
            AtomicConceptOfLabel source, target)
11. String matcher;
12. String relation="Idk";
13. int i=0;
14. while ((i<sizeof(matchers))&&(relation=="Idk"))
15.  matcher= matchers[i];
16.  relation=executeMatcher(matcher,source,target);
17.  i++;
18. return relation;
```

mkAxioms takes as input two concepts at nodes *contextA* and *contextB*. It produces as an output a set of axioms holding between the atomic concepts of labels in both concepts. First, the element level matchers which have to be executed (based on the configuration settings), are obtained in line 4. Then, for each pair of atomic concepts of labels in both trees, semantic relations holding between them are computed by **getRelation**(line 7) and added to *axioms* (line 8).
getRelation takes as an input an array of matchers and two atomic concepts of labels. It returns the semantic relation holding between atomic concepts of labels according to the element level matchers. They are executed (line 16) until the semantic relation different from Idk is produced.

8.1 Prefix

Prefix is a string based matcher. It checks whether one input label starts with the other. It returns an equivalence relation in this case, and Idk otherwise. The examples of relations Prefix produced are summarised in Table 3. Prefix is efficient in matching cognate words

Table 3: Semantic relations produced by prefix matcher

Source label	Target label	Semantic relation
net	network	=
hot	hotel	=
cat	core	Idk

and similar acronyms (e.g., *RDF* and *RDFS*) but often syntactic similarity does not imply semantic relatedness. Consider the examples in Table 3. The matcher returns equality for *hot* and *hotel* which is wrong but it recognises the right relations in the case of the pairs *net*, *network* and *cat*, *core*.

8.2 Suffix

Suffix is a string based matcher. It checks whether one input label ends with the other. It returns the equivalence relation in this case and Idk otherwise. The results produced by Suffix are summarised in Table 4. Suffix performs very similarly to Prefix. It correctly

Table 4: Semantic relations produced by suffix matcher

Source label	Target label	Semantic relation
phone	telephone	=
word	sword	=
door	floor	Idk

recognises cognate words (*phone*, *telephone*) but makes mistakes with syntactically similar but semantically different words (*word*, *sword*).

8.3 Edit Distance

Edit distance is a string based matcher. It calculates the edit distance measure between two labels. The calculation includes counting the number of the simple editing operations (delete, insert and replace) needed to convert one label into another and dividing the obtained number of operations with *max(length(label1),length(label2))*. The result is a value in [0..1]. If the value exceeds a given threshold (0.6 by default) the equivalence relation is returned, otherwise, Idk is produced. Edit Distance is useful with some unknown to

Table 5: Semantic relations produced by edit distance matcher

Source label	Target label	Semantic relation
street	street1	=
proper	propel	=
owe	woe	Idk

WordNet labels. For example, it can easily match labels *street1*, *street2*, *street3*, *street4* to *street* (edit distance measure is 0.86). In the case of matching *proper* with *propel* the edit distance similarity measure has 0.83 value, but equivalence is obviously the wrong output.

8.4 nGram

NGram is a string based matcher. It counts the number of the same ngrams (e. g., sequences of n characters) in the input labels. For example, trigrams for the word *address* are *add*, *ddr*, *dre*, *res*, *ess*. If the value exceeds a given threshold the equivalence relation is returned. Otherwise Idk is produced. The relations produced by NGram are summarised in Table 6.

Table 6: Semantic relations produced by nGram matcher

Source label	Target label	Semantic relation
address	address1	=
behaviour	behaviour	=
door	floor	Idk

Table 7: Possible relationships in WordNet

Relation	Description	Example
Hypernym	is a generalisation of	*motor vehicle* is a hypernym of *car*
Hyponym	is a kind of	*car* is a hyponym of *motor vehicle*
Meronym	is a part of	*lock* is a meronym of *door*
Holonym	contains part	*door* is a holonym of *lock*
Troponym	is a way to	*fly* is a troponym of *travel*
Antonym	opposite of	*stay in place* is an antonym of *travel*
Attribute	attribute of	*fast* is an attribute of *speed*
Entailment	entails	*calling on the phone* entails *dialling*
Cause	cause to	*to hurt* causes *to suffer*
Also See	related verb	*to lodge* is related to *reside*
Similar to	similar to	*evil* is similar to *bad*
Participle of	is participle of	*stored* is the participle of *to store*
Pertainym	pertains to	*radial* pertains to *radius*

8.5 WordNet

WordNet [17] is a lexical database which is available online and provides a large repository of English lexical items. WordNet contains synsets (or senses), structures containing sets of terms with synonymous meanings. Each synset has a gloss that defines the concept that it represents. For example the words *night*, *nighttime* and *dark* constitute a single synset that has the following gloss: *the time after sunset and before sunrise while it is dark outside*. Synsets are connected to one another through explicit semantic relations. Some of these relations (hypernymy, hyponymy for nouns and hypernymy and troponymy for verbs) constitute kind-of and part-of (holonymy and meronymy for nouns) hierarchies. In example, *tree* is a kind of *plant*, *tree* is hyponym of *plant* and *plant* is hypernym of *tree*. Analogously from *trunk* is a part of *tree* we have that *trunk* is meronym of *tree* and *tree* is holonym of *trunk*. The relations of WordNet 2.0 are presented on Table 7.

Figure 5 shows an example of nouns taxonomy.

WordNet matcher is a knowledge based matcher. It translates the relations provided by WordNet to semantic relations according to the following rules:

- $A \subseteq B$ if A is a hyponym, meronym or troponym of B;

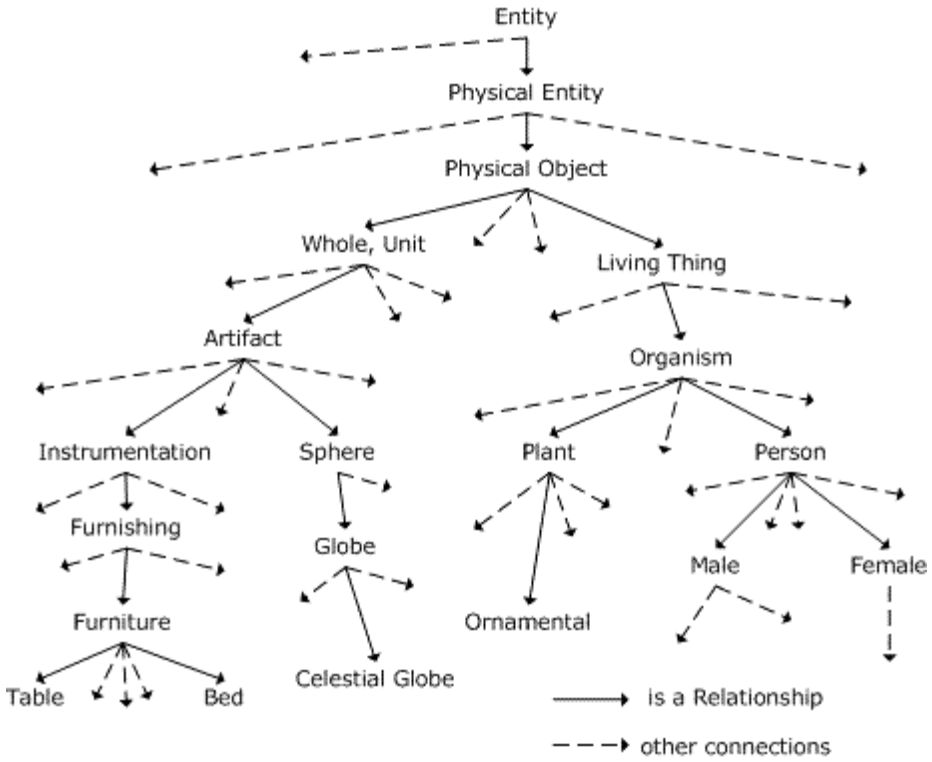


Figure 5: An example of WordNet nouns taxonomy

- $A \supseteq B$ if A is a hypernym or holonym of B;
- $A = B$ if they are connected by synonymy relation or they belong to one synset (night and nighttime from above-mentioned example);
- $A \perp B$ if they are connected by antonymy relation or they are the siblings in the part of hierarchy.

Notice that hyponymy, meronymy, troponymy, hypernymy and holonymy relations are transitive. Therefore, for example, from Figure 5 we can derive that $Person \subseteq LivingThing$.

If none of the above mentioned relations holds among the two input synsets *Idk* relation is returned.

Table 8 illustrates WordNet matcher results.

Table 8: Semantic relations produced by WordNet matcher

Source label	Target label	Semantic relation
car	minivan	$\sqsupseteq$
car	auto	$=$
tail	dog	$\sqsubseteq$
red	pink	Idk

8.6 Leacock Chodorow Matcher

The Leacock Chodorow matcher is a knowledge based matcher. It exploits Leacock Chodorow semantic similarity measure. It returns $\equiv$ if the measure exceeds the given threshold and Idk otherwise. The measure is based on counting the number of links between two input synsets. Intuitively, the shorter the path, the more related are the concepts under consideration. Leacock and Chodorow[13] considered the noun *is a* hierarchy. They proposed the following formula for estimating the similarity of two synsets:

$$sim_{lc}(c_1, c_2) = -\ln\left(\frac{spath\,(c_1, c_2)}{2 \cdot D}\right) \tag{3}$$

where $spath(s_1, s_2)$ is the length of the shortest path between the two synsets c_1 and c_2 and D is the depth of the tree.

The measure has a lower bound of 0 and upper bound $U_b = -\ln(1/(2 \cdot maxDepth))$, where $maxDepth$ is a maximum depth of the taxonomy.

Table 9 illustrates Leacock Chodorow matcher results with 3.0 threshold.

Table 9: Semantic relations produced by Leacock Chodorow matcher

Source synset	Target synset	Semantic relation
autograph	signature	=
actor	actress	=
dog	cat	Idk
sky	atmosphere	Idk

8.7 Resnik Matcher

The Resnik matcher is a knowledge based matcher. It exploits Resnik semantic similarity measure. It returns $\equiv$ if the measure exceeds the given threshold and Idk otherwise. This measure is based on the concept of *information content*[20]. Information content defines the generality or specificity of a concept in a certain topic.

Information content of the given concept is calculated as follows. Firstly the frequency[1] of concept occurrences F_C in *text corpus* is calculated. Then the frequencies of all subsuming concepts are calculated and added to F_C. Thus the root concept will count the occurrences of all the concepts in its taxonomy. In the case of WordNet synsets the frequency counts are precomputed for wide range of large scale corpora. In our preliminary experiments we exploited *Brown corpus of standard American English* [12].

Information content of a concept c is defined as:

$$IC(c) = -\ln\left(\frac{freq(c)}{freq(root)}\right) \tag{4}$$

[1]Here and throughout the paper, following NL tradition, we treat frequency as count (i.e., frequency of concept occurrences is a number of times the given concept occurs in the corpora).

where $freq(c)$ and $freq(root)$ are, respectively, the frequencies of the concept c and the *root* of the taxonomy. Note that the fraction represents the probability of occurrence of the concept in a large corpus.

Resnik defines the semantic similarity of the two concepts as the amount of information they share in common. To be more precise, the amount of information two concepts share in common is equal to the value of information content of their *lowest common subsumer*, that is the lowest node in the taxonomy that subsumes both concepts. For example the lowest common subsumer of *cat* and *dog* is *carnivore*. Therefore Resnik measure is defined as:

$$sim_{res}(c_1, c_2) = IC\left(lcs\left(c_1, c_2\right)\right) \tag{5}$$

where IC is the information content of a concept and $lcs(c_1, c_2)$ is the lowest common subsumer of concepts c_1 and c_2.

This measure has a lower bound of 0 and no upper bound.

Table 10 illustrates Resnik matcher results with 10.0 threshold.

Table 10: Semantic relations produced by Resnik matcher

Source synset	Target synset	Semantic relation
robot	android	=
actor	actress	Idk
dog	cat	Idk

8.8 Jiang Conrath Matcher

The Jiang Conrath matcher is a knowledge based matcher. It exploits the Jiang Conrath semantic similarity measure. It returns $\equiv$ if the measure exceeds the given threshold and Idk otherwise. This measure [11] incorporates both information content of the concepts and the information content of their lowest common subsumer. Originally Jiang Conrath defined the *distance* between two concepts as:

$$distance_{jc}(c_1, c_2) = IC(c_1) + IC(c_2) - 2 \cdot IC\left(lcs\left(c_1, c_2\right)\right) \tag{6}$$

where IC is the information content of a concept and lcs finds the lowest common subsumer of two given concepts.

Therefore the similarity of two concepts can be represented as

$$sim_{jc}(c_1, c_2) = \frac{1}{distance_{jc}(c_1, c_2)} \tag{7}$$

The formula has two special cases:

- In the first case all information content values are 0:

$$IC(c_1) = IC(c_2) = IC\left(lcs\left(c_1, c_2\right)\right) = 0 \tag{8}$$

  This happens when both concepts and their lowest common subsumer are either the *root node* or have a frequency count of 0. In both cases 0 similarity is returned.

- The second case is when

$$IC(c_1) + IC(c_2) = 2 \cdot IC\left(lcs\left(c_1, c_2\right)\right) \tag{9}$$

  which usually happens when

$$IC(c_1) = IC(c_2) = IC\left(lcs\left(c_1, c_2\right)\right) \tag{10}$$

  In this case c_1 and c_2 are the same concept and so we would like to return a maximum value of relatedness.

This measure has a lower bound of 0 and the upper bound $U_b = \frac{1}{-\ln((f_{root}-1)/f_{root})}$, where f_{root} is the frequency of the taxonomy root.

The pseudo code for the algorithm is presented below.

```
1 struct Synset
2   String ID;
3   String[] lemmas;
4   String gloss;
5   String[] relations;

6 float match( Synset synset1, Synset synset2 )
7   String shortestPathId = getShortestPath( synset1, synset2 );
8   Synset lcs = getLowCommonSubsumer( shortestPathId );
9   float ic_lcs = getInformationContent( lcs );
10  float ic_s1 = getInformationContent( synset1 );
11  float ic_s2 = getInformationContent( synset2 );
12  if ( ic_s1 == ic_s2 && ic_s1 == ic_lcs && ic_lcs == 0)
13   return 0;
14  if ( ic_s1 + ic_s2 == 2*ic_lcs )
15    return maxValue;
16  float distance = ic_s1 + ic_s2 - 2*ic_lcs;
17  return 1/distance;
```

where **ID** (line 2) is the unique identifier, **lemmas** (line 3) is the list of synonyms that represent this synset, **gloss** (line 4) is the definition associated to that synset and **relations** (line 5) is a list of pointers to other synsets connected to this by a WordNet relation. **maxValue** (line 12) is the upper bound.

Firstly the shortest path between two synsets is computed (line 7). Then the lowest common subsumer of the input synsets is obtained (line 8). The information content values are computed for both synsets and lowest common subsumer in lines 6-8. Finally after handling the special cases (lines 12-15) the **distance** (line 16) and similarity (line 17) are computed.

Table 11 illustrates Jiang Conrath matcher results with 1.0 threshold.

8.9 Lin Matcher

The Lin matcher is a knowledge based matcher. It exploits Lin semantic similarity measure. It returns $\equiv$ if the measure exceeds the given threshold and Idk otherwise. This measure is also based on information content[14]. It is defined as follows:

$$sim_{lin}(c_1, c_2) = \frac{2 \cdot IC\left(lcs\left(c_1, c_2\right)\right)}{IC(c_1) + IC(c_2)} \tag{11}$$

Table 11: Semantic relations produced by Jiang Conrath matcher

Source synset	Target synset	Semantic relation
trip	hallucination	=
actor	actress	=
dog	cat	Idk

In the case of $IC(c_1) = 0$ and $IC(c_2) = 0$ 0 similarity is returned.

Table 12 illustrates Lin matcher results with 0.9 threshold.

Table 12: Semantic relations produced by Lin matcher

Source synset	Target synset	Semantic relation
robot	android	Idk
actor	actress	=
dog	cat	Idk

8.10 Hirst-St.Onge Matcher

The Hirst-St.Onge matcher is a knowledge based matcher. It exploits Hirst-St.Onge semantic similarity measure. It returns $\equiv$ if the measure exceeds the given threshold and Idk otherwise. This measure in contrast to information content based measures is not restricted to noun hierarchies.

Hirst and St.Onge [10] classified links in WordNet in 3 different categories:

- *Upward* (e.g. hypernym);
- *Downward* (e.g. holonym);
- *Horizontal* (e.g. antonym);

According to them two words are connected by a *strong* relation if:

- they both belong to the same synset;
- they belong to synsets connected by a *horizontal* link;
- one is a compound word, the second one is substring of the first while their synsets are connected by an *is a* relation;

For strongly related words relatedness is computed as $2 \cdot C$, where C is a constant used in the formula for medium-strong relations. Its value is 8, so the coefficient is equal to 16.

A *medium-strong* relation exists if the two synsets are connected by a *valid path* in WordNet. A path is considered valid if it is not longer than 5 links and conforms to one of the eight predefined patterns. The relatedness between two words connected by a medium-strong relation corresponds to the *weight* of the path which is given by the following formula:

$$Weight = C - Pathlength - k \cdot Changesindirection \tag{12}$$

where C and k are constants and, in our case, they are assumed to equal 8 and 1 respectively. The pseudocode below illustrates the algorithm. In order to compute the relatedness of two concepts first the type of relation holding between them is determined (line 2). Then the given relation is compared with existing patterns (line 4).

```
1 float match( Synset synset1, Synset synset2 )
2   if ( checkStrongRelationship( synset1, synset2 ))
3     return 2*C;
4   int weight = getMedStrongWeight( 0, 0, 0, synset1, synset2 );
5   return weight;
```

checkStrongRelationship (line 2) takes in input the two synsets and returns true if they fulfil one of the requirements of *Strong relations* and false otherwise.
getMedStrongWeight (line 4) takes in input two synsets and three zero values, that are respectively defined as **state**, **distance** and **chdir**, and returns the weight as stated in the above formula. Basically it searches recursively for a path from **synset1** to **synset2**, taking trace of the **distance**, changes in direction (**chdir**) so far and giving the **state** value for the next call. There are 8 possible states (from 0 to 7) in which the function may find itself, every state defines the rules the path has to follow. In particular, they specify the categories of links used in the last iteration, the ones that are allowed in the current step and the ones to use as next, taking into account the possible changes in direction.

This measure has a lower bound of 0 and an upper bound of 16.

Table 13 illustrates Hirst-St.Onge matcher results with 4.0 threshold.

Table 13: Semantic relations produced by Hirst-St.Onge matcher

Source synset	Target synset	Semantic relation
school	private school	=
actor	actress	=
dog	cat	Idk
sky	atmosphere	Idk

8.11 Context Vectors Matcher

The Context Vectors Matcher is a knowledge based matcher. It exploits context vectors semantic similarity measure. It returns $\equiv$ if the measure exceeds the given threshold and Idk otherwise. This measure is based on context vector notion introduced by Schütze in [22]. Originally exploited for *word sense disambiguation*, context vectors were adapted for semantic similarity computation exploiting WordNet in [19].

The context vectors computation process starts from the selection of the highly topical words which will define the dimensions of our *word space*. For our experiments we used WordNet glosses as a corpus. We stemmed all the words in the glosses and filtered out the function words. Then we counted the frequencies of the words in the corpus. Then we cut off the words with frequencies lower then 5 and higher then 1000. This allowed us

to keep the most informative words. We also added a *tf-idf*[2] cutoff with an upper bound of 1500. This allowed us to perform additional filtering of the frequent words. Further we will call the remaining words content words.

Afterwards we have created word vectors for all content words w as follows:

1. Initialise a vector $\vec{w}$ to zero
2. Find every occurrence of w in WordNet glosses
3. For each occurrence, search that gloss for words in the word space and increment the dimensions of $\vec{w}$ that correspond to those words.

The basic idea here is to have a matrix of word vectors, where every row corresponds to a word in the content words list and every column corresponds to the respective frequencies of each word in the word space.

The final step is to calculate gloss vectors for every synset in WordNet, which is done by adding the word vectors for each content word in the gloss. For example, if we want the gloss vector of *clock* we have to consider its gloss: *a timepiece that shows the time of day* and add the word vectors of *timepiece*, *shows*, *time* and *day*. Notice that this is a simplified example because for our experiments we use *extended glosses*, thus we had to take into account the glosses of every concept connected to *clock* by a WordNet relation.

As soon as gloss vectors are calculated they are stored in database and the preprocessing phase is finished.

Semantic similarity of two synsets is defined as follows

$$sim_{cv}(c_1, c_2) = \cos(angle(\vec{v_1}, \vec{v_2})) \tag{13}$$

where c_1 and c_2 are the concepts, $\vec{v_1}$ and $\vec{v_2}$ are the respective gloss vectors and *angle* is the angle between vectors. This formula can be rewritten using vector products, it becomes:

$$sim_{cv}(c_1, c_2) = \frac{\vec{v_1} \cdot \vec{v_2}}{|v_1|\,|v_2|} \tag{14}$$

where at the denominator we have the *magnitude*[3] of the two vectors. It is a dot product[4] between two normalised vectors.

Figure 6 illustrates context vectors similarity in 2 dimensional space.

The pseudo code of context vector semantic similarity computation algorithm is as follows:

```
1 float match( Synset synset1, Synset synset2 )
2   int glossVec1[] = loadGlossVector( synset1 );
3   int glossVec2[] = loadGlossVector( synset2 );
4   float normVec1[] = normalizeVec( glossVec1 );
5   float normVec1[] = normalizeVec( glossVec1 );
6   return dotProduct( normVec1, normVec2 );
```

[2]tf-idf is a weight used to evaluate how important a word is to a document (or gloss in our case). The formula we used is $tfidf = tf \cdot \ln(idf)$; where tf is the frequency of occurrence of the word and $idf = \frac{nr.documents}{docFrequency}$. For our experiments *nr.documents* is the number of glosses and *docFrequency* is the number of glosses in which our word appears.

[3]the magnitude of vector $\vec{v}$ is equal to $\sqrt{\sum_{i=1}^{n} v_i^2}$

[4]the dot product between vector $\vec{v}$ and vector $\vec{w}$ is $\vec{v} \cdot \vec{w} = \sum_{i=1}^{n} v_i w_i$

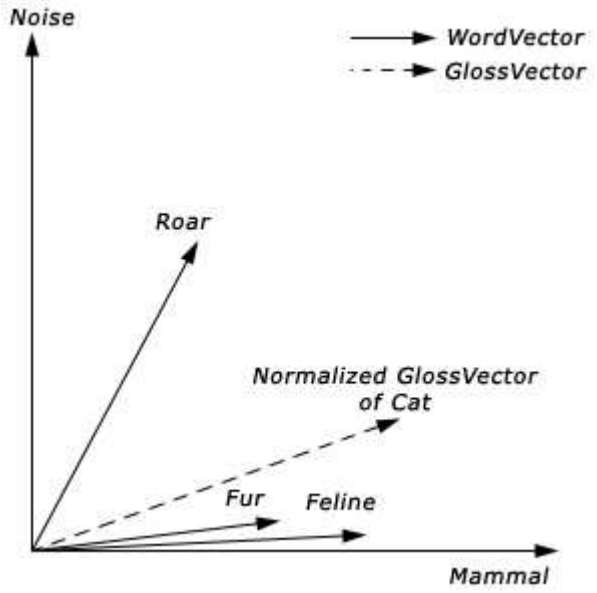


Figure 6: An example of a 2-dimensional space for Word and Gloss vectors for "cat - feline mammal usually having thick soft fur and being unable to roar" synset

loadGlossVector (line 2-3) loads, from database, the precomputed gloss vector for the given synset.
normalizeVec (line 4-5) calculates the normalised form of the given vector.
dotProduct (line 6) return the dot product between two given vectors.

The measure has a lower bound of 0 and an upper bound of 1.

Table 14 illustrates context vectors matcher results with 0.3 threshold.

Table 14: Semantic relations produced by context vectors matcher

Source synset	Target synset	Semantic relation
autograph	signature	=
actor	actress	=
robot	android	=
fruit	glass	Idk

8.12 WordNet gloss

WordNet gloss is a gloss based matcher. It compares the labels of the first input sense with the WordNet gloss of the second. First, it extracts the labels of the first input sense from WordNet. Then, it computes the number of their occurrences in the second gloss. If this number exceeds a given threshold, $\sqsubseteq$ is returned. Otherwise, Idk is produced.

The reason why the less general relation is returned comes from the lexical structure of the WordNet gloss. Very often the meaning of the index words is explained through a specification of the more general concept. In the following example, *hound* (*any of several breeds of dog used for hunting, typically having large drooping ears*) is described through

the specification of the more general concept *dog*. In this example *hound* is a *dog* with special properties (*large drooping ears*, *used for hunting*).

Counting the label occurrences in the gloss does not give a strong evidence of what relation holds between concepts. For example, WordNet gloss returns the less general relation for *hound* and *ear* in the above mentioned example, which is clearly wrong.

Table 15 illustrates WordNet gloss matcher results.

Table 15: Semantic relations produced by WordNet gloss matcher

Source synset	Target synset	Semantic relation
hound	dog	$\sqsubseteq$
hound	ear	$\sqsubseteq$
dog	car	Idk

8.13 WordNet extended gloss

WordNet extended gloss is a gloss based matcher. It compares the labels of the first input sense with the extended gloss of the second. This extended gloss is obtained from the input sense descendants (ancestors) descriptions in the is-a (part-of) WordNet hierarchy. A given threshold determines the maximum allowed distance between these descriptions and the input sense in the WordNet hierarchy. By default, only direct descendants (ancestors) are considered. The idea of using extended gloss originates from [2]. Unlike [2], we do not calculate the extended gloss overlaps measure, but count the number of first input sense labels occurrences in the extended gloss of the second input sense. If this number exceeds a given threshold, a semantic relation is produced. Otherwise, Idk is returned. The type of relation produced depends on the glosses we use to build the extended gloss. If the extended gloss is built from descendant (ancestor) glosses, then the $\sqsupseteq$ ($\sqsubseteq$) relation is produced. For example, the relation holding between the words *dog* and *breed* can be easily found by this matcher. These concepts are not related in WordNet, but the word *breed* occurs very often in the *dog* descendant glosses.

Table 16 illustrates WordNet extended gloss matcher results.

Table 16: Semantic relations produced by WordNet extended gloss matcher

Source synset	Target synset	Semantic relation
dog	breed	$\sqsupseteq$
wheel	machinery	$\sqsubseteq$
dog	cat	Idk

8.14 Gloss comparison

Gloss comparison is a gloss based matcher. Within the matcher the number of the same words occurring in the two input glosses increases the similarity value. The equivalence

relation is returned if the resulting similarity value exceeds a given threshold. *Idk* is produced otherwise.

Let us try to find the relation holding, for example, between *Afghan hound* and *Maltese dog* using gloss comparison strategy. These two concepts are breeds of *dog*, but unfortunately WordNet does not have explicit relation between them. However, the glosses of both concepts are very similar. Let us compare:

Maltese dog is a breed of toy dogs having a long straight silky white coat.

And:

Afghan hound is a tall graceful breed of hound with a long silky coat; native to the Near East.

There are 4 shared words in both glosses (*breed*, *long*, *silky*, *coat*). Hence, the two concepts are taken to be equivalent. Table 17 illustrates gloss comparison matcher results. Several modifications of this matcher exist. One can assign a higher weight to the phrases

Table 17: Semantic relations produced by gloss comparison matcher

Source synset	Target synset	Semantic relation
Afghan hound	Maltese dog	=
dog	cat	Idk

or particular parts of speech than single words [18]. In the current implementation we have exploited the approach used in [18], but changed the output to be a semantic relation.

8.15 Extended Gloss comparison

Extended gloss comparison is a gloss based matcher. It compares two extended glosses built from the input senses. Thus, if the first gloss has a lot of words in common with descendant glosses of the second then the first sense is more general than the second and vice versa. If the corpuses (extended glosses) formed from descendant (ancestor) glosses of both labels have a lot of words in common (this value is controlled by a given threshold) then the equivalence relation is returned. For example, *dog* and *cat* are not connected by any relation in WordNet. Comparing the corpuses obtained from descendants glosses of both concepts we can find a lot of words in common (*breed*, *coat*, etc). Thus, we can infer that *dog* and *cat* are related (they are both pets), and return the equivalence relation. The relations produced by the matcher are summarised in Table 18.

Table 18: Semantic relations produced by extended gloss comparison matcher

Source synset	Target synset	Semantic relation
house	animal	Idk
dog	cat	=

8.16 Semantic Gloss comparison

Semantic Gloss comparison is a gloss based matcher. The key idea is to maintain statistics not only for the same words in the input senses glosses (like in Gloss comparison) but

also for words which are connected through is-a (part-of) relationships in WordNet. This can help finding the gloss relevance not only at the syntactic but also at the semantic level. In Semantic Gloss Comparison we consider synonyms, less general and more general concepts which (hopefully) lead to better results.

In the first step the glosses of both senses are obtained. Then, they are compared by checking which relations hold in WordNet between the words of both glosses. If there is a sufficient number of synonyms (in the current implementation this value is controlled by a threshold), the equivalence relation is returned. In the case of a large amount of more (less) general words, the output is $\sqsupseteq$ ($\sqsubseteq$) correspondingly. *Idk* is returned if we have a nearly equal amount of more and less general words in the glosses or there are no relations between words in glosses. Table 19 contains the results produced by semantic gloss comparison matcher.

Table 19: Semantic relations produced by extended gloss comparison matcher

Source synset	Target synset	Semantic relation
dog	breed	$\sqsupseteq$
dog	cat	Idk
wheel	machinery	$\sqsubseteq$

8.17 MatchMiner

It is an approximated knowledge-based element level matcher which uses the growing amount of online available semantic data which makes up the Semantic Web as a source of background knowledge in ontology mapping. The core idea of our method is that, given two terms, an appropriate mapping will be discovered by inspecting how these terms are related in online available ontologies (the concepts in online ontologies that are equivalent to the terms to be mapped are referred to as "anchor terms" following the terminology introduced by [1]). The mappings that are obtained can be considered approximate in the following ways:

Choose approximate anchor terms. When finding anchor terms, it is often difficult to find online ontologies that define exactly the same concepts as those that we wish to map. In case of failure to find exact anchor terms, we try to find terms that are syntactic or semantic approximations of the terms to be mapped (as described in Section 8.17.1).

Discover both exact and semantically approximate mappings. Our technique can discover both equivalence relations between the two terms and a variety of semantic approximations ($\sqsubseteq, \sqsupseteq, \perp$). We present this mechanism in Sections 8.17.2 and 8.17.3.

Approximate the correct mapping depending on available resources. In the context of our technique, one can argue that the correct mapping between two terms has been found only at a point when all online ontologies have been inspected and a conclusion has been drawn by considering all existing evidence. This, however, is often unfeasible as the response time of a mapping algorithm is a resource that should

be optimised. We are working on phrasing our algorithms as anytime algorithms which can produce some useful results quickly by inspecting only a few ontologies and then provide increasingly correct approximations of the truth as more time is available.

Implementation Details. We explore our idea by implementing different mapping strategies on top of the Swoogle'05 ontology search engine [4]. Swoogle crawls and indexes a large amount of semantic metadata available online and as such allows access to a large part of the Semantic Web.

Notations. Each strategy takes two candidate concept names (`A` and `B`) as an input and returns the discovered mapping between them. The corresponding concepts in the selected ontology are `A'` and `B'` ("anchor terms"). We rely on the description logic syntax for semantic relations occurring between concepts in an ontology,e.g., `A'` $\sqsubseteq$ `B'` means that `A'` is a sub-concept of `B'` in a selected ontology. The returned mappings are expressed using C-OWL [3] like notations, e.g., `A` $\xrightarrow{\sqsubseteq}$ `B` means that `A` is a sub-concept of `B`.

8.17.1 Choosing approximate anchor terms

In order to discover more ontologies that cover the candidate concepts, the process of finding anchor terms must be more flexible. This flexibility can be achieved by considering the following techniques:

A. String normalisation. Differences between concept names can be based on simple differences in naming conventions (e.g., $TURKEY\_BREAST$ and $TurkeyBreast$). Most mapping mechanisms use *string normalisation* techniques [5], that consist in transforming strings into a standard form (low case letter, using a particular character for spaces and separators) before comparison. Our ontology selection relies on such mechanisms as well.

B. Dealing with compound names. Compound names are particularly difficult to match as they are likely to appear under slightly different forms. Several mapping techniques suggest to be more flexible when searching for compound terms and to allow for:

Different order of the constituents. For example, the term $TurkeyRoast$ does not appear in Swoogle, but $RoastTurkey$ does.

Additional constituents. For example, $TurkeyBreast$ is not covered but $TurkeyMeatBreast$ (which additionally contains $Meat$) is.

Less constituents. Some compound terms are only partially covered. For example, $MeatProduct$ does not exist in Swoogle, but $Meat$ does.

Such a flexible matching is also used when discovering anchor terms in the work of Aleksovski et al. [1]. However, while the examples given above are semantically equivalent, automatically identifying lexically different but semantically equivalent compound terms is a difficult task.

B. Exploiting semantic relations between terms. Semantic relations such as synonymy can be used to replace terms with their semantic equivalents. A good source for

(a) (b)

Figure 7: Using ontologies as background knowledge for semantic mapping: (a) discovering mappings within one ontology (b) discovering cross-ontology mappings.

synonymy information is WordNet, especially for simple terms. However, the drawbacks of WordNet are that it is difficult to get relevant synonyms unless the sense of the term is known a priory and that compound terms are weakly covered.

8.17.2 Mappings Based on One Ontology

Our simplest strategy consists in using Swoogle to find ontologies containing concepts equivalent to the candidate concepts and to derive mappings from their relationship in the selected ontologies. Figure 7(b) illustrates this strategy with an example where three ontologies are discovered containing the concepts `A'` and `B'` with the same names as `A` and `B`. The first ontology contains no relation between the anchor concepts, while the other two ontologies contain a subsumption relation.

The concrete steps of this strategy are:

1. Select ontologies containing concepts `A'` and `B'` corresponding to `A` and `B`;
2. For each resulting ontology:
   - if `A'` $\equiv$ `B'` then derive `A` $\xrightarrow{\equiv}$ `B`;
   - if `A'` $\sqsubseteq$ `B'` then derive `A` $\xrightarrow{\sqsubseteq}$ `B`;
   - if `A'` $\sqsupseteq$ `B'` then derive `A` $\xrightarrow{\sqsupseteq}$ `B`;
   - if `A'` $\perp$ `B'` then derive `A` $\xrightarrow{\perp}$ `B`;
3. If no ontology is found, no mapping is derived;

Even if this strategy seams simple, it leads to several implementation choices, depending on the relative importance given to time performance and accuracy of the mapping mechanism:

Stop when the first mapping is found. In its simplest version, the algorithm would stop as soon as a mapping is discovered. This is the easiest way to deal with the multiple returned ontologies but it assumes that the first discovered relation can be trusted and there is no need to inspect the other ontologies.

Dealing with contradictions. Instead of relying on the information provided by only one ontology as before, we can envisage to combine the results obtained using all the selected ontologies. Mappings resulting from different sources can be different (e.g., `A` $\xrightarrow{\sqsubseteq}$ `B` and `A` $\xrightarrow{\sqsupseteq}$ `B`), or, in the worst case, inconsistent (e.g., `A` $\xrightarrow{\sqsubseteq}$ `B` and `A` $\xrightarrow{\perp}$ `B`). Several ways of dealing with these contradictions can be considered: we can keep all the mappings (favouring recall), only keep mappings without contradiction (favouring precision), keep the mappings that are derived from most of the ontologies, or try to combine the results (e.g., by deriving `A` $\xrightarrow{\equiv}$ `B` from `A` $\xrightarrow{\sqsubseteq}$ `B` and `A` $\xrightarrow{\sqsupseteq}$ `B`). In any case,

combining the results from several ontologies is more time consuming (but more reliable) than deriving it from a single ontology.

Considering a particular level of inferences. In the simplest implementation, we can rely on *direct* and *declared* relations between `A'` and `B'` in the selected ontology. But, for better results, *indirect* and *inferred* relations should also be exploited (e.g., if `A'` $\sqsubseteq$ `C` and `C` $\perp$ `B'`, then `A'` $\perp$ `B'`). Different levels of inferences can be considered (no inference, basic transitivity, DL reasoning), each of them representing a particular compromise between the performance of mapping and the completeness of the result.

8.17.3 Cross-Ontology Mapping Discovery

The previous strategy assumes that a semantic relation between the candidate concepts can be discovered in a single ontology. However, some relations could be distributed over several ontologies. Therefore, if no ontology is found that relates both candidate concepts, then the mappings should be derived from two (or more) ontologies. In this strategy, mapping is a recursive task where two concepts can be mapped because the concepts they relate in some ontologies are themselves mapped (Figure 7(c)):

1. If no ontologies are found that contain both `A` and `B` then select all ontologies containing a concept `A'` corresponding to `A`;
2. For each of the resulting ontologies:
   (a) for each `C` such that `A'` $\sqsubseteq$ `C`, search for mappings between `C` and `B`;
   (b) for each `C` such that `A'` $\sqsupseteq$ `C`, search for mappings between `C` and `B`;
   (c) derive mappings using the following rules:
   - (r1) if `A'` $\sqsubseteq$ `C` and `C` $\xrightarrow{\sqsubseteq}$ `B` then `A` $\xrightarrow{\sqsubseteq}$ `B`
   - (r2) if `A'` $\sqsubseteq$ `C` and `C` $\xrightarrow{\equiv}$ `B` then `A` $\xrightarrow{\sqsubseteq}$ `B`
   - (r3) if `A'` $\sqsubseteq$ `C` and `C` $\xrightarrow{\perp}$ `B` then `A` $\xrightarrow{\perp}$ `B`
   - (r4) if `A'` $\sqsupseteq$ `C` and `C` $\xrightarrow{\sqsupseteq}$ `B` then `A` $\xrightarrow{\sqsupseteq}$ `B`
   - (r5) if `A'` $\sqsupseteq$ `C` and `C` $\xrightarrow{\equiv}$ `B` then `A` $\xrightarrow{\sqsupseteq}$ `B`

In this strategy, steps (a) and (b) can be run in parallel and stopped when one of them is able to establish a mapping. These two steps correspond to the recursive part of the algorithm. The task of *searching for mappings between `C` and `B`* can be realised using either of our two strategies.

8.18 PowerMap

PowerMap [15] is a knowledge based semantic matcher which uses not only WordNet as an oracle but the ontologies on the SW as a background knowledge. The PowerMap algorithm is used for discovering and performing approximate mappings between multiple available online heterogeneous ontologies on the Web, with no pre-determined assumption about the source and the ontological structure of these data. The main differentiating

features from other methods where mappings are required between two ontologies is that in PowerMap the **mapping process is driven by the task** that has to be performed, more concretely by the input query asked by the user. Indeed, in this method not all the concepts of the ontology need to be interpreted; only ontology entities that seem similar to the input query should be interpreted.Therefore, we are likely to discover several candidate mappings drawn from different ontologies that may only have very few concepts in common.

The mappings considered by PowerMap are exact and approximate mappings so a non-relevant ontology with a potential relevant mapping is left out. PowerMap is a hybrid matching algorithm comprising *terminological/element and structural schema matching* techniques with the assistance of large scale ontological or lexical resources. At element level an input query is represented by a keyword or set of keywords and syntactic and semantic techniques can be used to obtain the ontology mappings. At structural level the input query is represented by a triple or set of triples that indicated how the words are related (in fact, better results are expected considering the triples than by only considering isolated words), at structural level the meaningful mappings that better represents the query domain are filtered by determining those ontologies that better cover the input query triples and by studying the ontology relatedness to determine the valid semantic interpretation (relation mapping techniques to match between the predicates of the triples and relations in the identified ontologies are used)

PowerMap consists of three main phases. In order to optimise performance, the complexity of these phases increases, hence the most time-consuming techniques are executed last, when the search has been narrowed down to a smaller set of ontologies. However, in the context of the Open Knowledge project PowerMap is used as at element level matcher and therefore only the phase I and phase II (element level) are considered here. After the execution of these two phases, each input keyword has associated a set of ontology mappings (classes, instances, properties, or literals), where each mapping has a WordNet synset, a semantic relation (exact, synonimia, hypernimia and hyponimia) and a numeric score. Given this ontology background information for each keyword the semantic relation that hold between each pair of input keywords (if any) can be obtained as seen in the example subsection.

Phase I: Syntactic Mapping. The role of this phase is to identify candidate mappings and semantic relations for all query terms in different online ontologies (therefore identify potentially relevant ontologies for that particular query). Recall is important so that no relevant ontologies with potential entities to map are left out. To bridge the gap between user and ontology terminology, partial or fuzzy mappings are needed to find relevant hits and ontologies with potential mapping for any of the keywords in the input query. This is the simplest phase as it only considers concept labels and local names (i.e., ignores the structure of ontologies). Approximate mappings rely on simple, string-based comparison methods (e.g., edit distance metrics) and WordNet to look up lexically related words (synonyms, hypernyms and hyponyms), in case of nominal compounds, like "educational organisation", the main lemma of the compound is obtained as substituted by a WN lexical related word, i.e. "educational system". However, not only WordNet is used to find lexically related words for an input keyword, also information about the superclasses and subclasses of the obtained ontology entity mappings is obtained (ontol-

ogy background) to perform a second iteration of this phase to look up for hypernyms and hyponyms and find new potentially relevant hits (relevant mappings) in other ontologies. Each mapping has associated information about it, like the semantic relation (exact, synonym, hypernym, hyponym) and the score. To finalise, we envision a scenario with thousand of KBs corresponding to hundred of ontologies. Therefore to efficiently perform approximate searches on large-scale ontology repositories, Lucene is used to create inverted indexes for the ontology elements (classes, instances, relations and literals), and is used as our fast search engine, which also supports fuzzy searches based on the Lavenshtein Distance, or Edit Distance algorithm.

Phase II: Semantic Mapping. This phase operates on the reduced set of ontologies identified in the previous phase. The goal of this phase is to verify the syntactic mappings identified previously and exclude those that do not make sense from a semantic perspective (e.g., the intended meaning of the query term differs from the intended meaning of the concept that was proposed as a candidate match). PowerMap relies on WordNet information and on the meaning of the mapped concepts in their hierarchy to verify that the proposed mappings are also semantically sound. For example, if the term "capital" is matched to concepts with identical labels in a geographical ontology and a financial ontology, these two meanings are not semantically equivalent. This phase relies on more complex methods. First, it exploits the hierarchical structure of the candidate ontologies to elicit the sense of the candidate concepts (in particular, the sense of an ontology class is determined by the sense of its ascendant/descendant in the ontology). Second, it uses WordNet based methods to compute the semantic similarity between the query terms and the ontology classes, and between the candidate ontology classes themselves. Similarity between them is measured by the distance (depth) and common subsumers between the two concept/senses in the WordNet "IS-A" taxonomy (as in hierarchy distance based matchers). Additionally the WN glosses are also used.

Example in the context of the OK scenario -element level matcher.

Consider the following example, given two input keywords, e.g. "academics" and "research fellow". By using PowerMap techniques, the two inputs can separately map to the following ontology entities with similar synsets:

Ontology mappings for "academics":

1. "academic-staff-member" in ontology 1 (fuzzy/approximate searches)
2. "professor-in-academia" in ontology 3 ("professor" is a WN hyponym)
3. "researcher" in ontology 4 ("researcher" is a subclass of "academic-staff-member" in ontology 1)

Ontology mappings for "research fellow":

1. "research-fellow" in ontology 4 (fuzzy/approximate searches)
2. "post-doc" in ontology 1 ("post-doc" is a WN synonym)

Looking at the mappings for both keywords and at the ontology information, we can infer that "academics" has a semantic relation with "research fellow" (`research fellow` $\sqsubseteq$ `academics`)because:

1. In the ontology 1 "post-doc" (the ontology mapping for "research fellow") is a subclass of "academic-staff-member" (the ontology mapping for "academics")

2. In the ontology 4 "research fellow" (the ontology mapping for "research fellow") is a subclass of "researcher" (the ontology mapping for "academics")

The applied rules to obtain the semantic relations are the same that in the previous algorithm based on Swoogle (section 1.3)

8.19 Community driven and Interaction specific Element Level Matching

8.19.1 Introduction

In the previous sections *interaction models* are already introduced. these IMs are process flow descriptions between roles. In Open-Knowledge, these roles are implemented by services and run on hosts connected to the OK-network. Clearly, the research on web-services is very related to this. In this section we come up with an approach where mappings are added inside IMs, because it allows one to drop some assumptions that are made in the traditional (semantic) web service domain.

Web-services are web-accessible software components, which are advertised and invoked via web-interfaces. Until now, invocation of these services is a time-intensive effort because human programmers still need to understand what a webservice does before they can use it. In other words, they need to know what the intended meaning is of the provided functionality offered by a service. Furthermore, they need to know the details of the operational interface of the service before they can integrate the service with other software.

The promise of the Semantic Web Service community is to reduce the human effort that is involved in locating, combining and deploying web-services. The general idea is to *annotate* with knowledge such that tasks like discovering, selecting and composing web-services can be done automatically.

Two languages have been proposed for enriching web-service descriptions with such semantic annotations: OWL-S and WSDL-S. An OWL-S description is divided into three parts, specifying what a service does (the "profile", used for advertising), how the service works internally (the "process model"), and how to interoperate with the service via messages (the "grounding"). WSDL-S adds semantics to non-semantic WSDL descriptions of services by allowing the annotation of WSDL data-types with pointers to a domain ontology, and by annotating operations with preconditions and effects.

Semantic annotations of web-services are crucial to the Semantic Web services vision. However, all these approaches make rather strong assumptions on the presence and nature of these annotations:

- **services are annoted with terms from an ontology:** Currently, manual effort is the only known method for providing semantic annotations of web-services. NLP techniques, which have proved quite effective for automatic annotation of web-pages are not capable of doing the same job for web-services given the small size of text-fragments available for web-service descriptions. Some early research work is being

done on automatically annotating web-services ([8, 21], but this technology is far from deployable. Consequently, providing semantic annotations for web-services will remain a costly effort for the foreseeable future. Hence, it is not obvious how realistic it is to simply assume the existence of such annotations.

- **these annotations cover the I/O types and the functionality**

  Even if these annotations are available, the Semantic Web Service vision assumes that they adequately describe both the I/O types of the service and the functionality of the service (= the relation between input- and output-parameters, plus possible side-effects of executing the service). It is well known from software engineering that in particular this final type of annotation (the functionality) is very hard to capture, both informally and formally (as would be required for automated processing of the service annotations).

- **this ontology is shared between the communicating services (or: between query and services)** Even when annotations of I/O types and functionality have been provided (possibly at great cost), the current approaches to locating services require that these annotations have been done using a single semantic vocabulary, in other words that they are taken from the same ontology. For example, the growing body of work on composition-as-planning assumes the description of web-services as plan-like operators, on which a planning algorithm can be run. But these algorithms assume that the output-state of one operator can be linked to the input-state of another. In practice, different providers of web-services will use different (and sometimes even proprietary) ontologies for annotating their services. This requires solving the ontology-mapping problem as part of the semantic web-service paradigm. (This problem is widely recognised in the Semantic Web community at large, with much work being done on ontology-mapping, but is not often acknowledged in work on Semantic Web Services).

  The only Semantic Web Services approach that explicitly acknowledges this problem is WSMO, which introduces the notion of mediators (or bridges) between service and requester (or between services). Such mediators are used to transform the goals of the requester to the capabilities of the provider, and if needed, the mediators map between different ontologies.

- **this ontology is semantically rich enough to do inference** Even when sufficiently precise annotations have been provided using a uniform vocabulary (ie assuming all the previous 3 issues have been solved), typical approaches to locating semantic web services rely on subsumption reasoning for web-service location or composition. However, it is well known from practical experience in the Semantic Web area that most realistic ontologies are semantically very "lightweight", and do not provide a strict subsumption hierarchy that would support rich DL-style reasoning.

In our work, we will provide an approach to locating web-services which will relax at least the first 3 of the above 4 assumptions:

- Web services only need to be described in non-semantic WSDL. We will use cheap "anchoring"-techniques to link these services with available ontologies.
- The minimal assumption will be that I/O types are available as keywords (which is trivially true through the availability of the non-semantic service description). Optionally, the functionality-description may be available as free-text, from which further anchorings may be computed.
- We will use ontology-mapping techniques to bridge between the different ontologies that provide the anchors for the service descriptions.
- If a rich ontology is available, we will deploy subsumption reasoning for locating services. In the (more likely) case that only very lightweight ontologies are available, locating services will be limited to keyword-based retrieval in the extreme case.

In the next subsection, we give an informal description of an approach that should be able to work with the relaxed assumptions that we mentioned before. After that we give a short overview of related work on reasoning with background knowledge.

8.19.2 Learning mappings by example

In the introduction, we mentioned four unrealistic assumptions that are often, if not always, made within the semantic web service domain. In our envisioned scenario, all services are described as follows:

- **Services are only described with strings ("terms")** Instead that the designer of a webservice needs to express the functionality by finding or developing an appropriate ontology, only a list of 'keywords' needs to be given, to express the functionality (i.e. 'what' it does and which input and output types it handles). In this way we drop the assumption that the descriptions are annotations with terms from an ontology
- **I/O types are described by primitive datatypes and keywords without instance enumeration** As already pointed in the previous bullet, the description of the webservice is only by strings that are not bounded to any formal and/or shared meaning. More specifically, the I/O types are described by keywords and the instances of the types are not enumerated. In this paper we distinct two different I/O types, namely *contenttypes* and *datatypes*. The first one gives a hint about which thing the input or output represents like a 'cityname' or 'processortype'. The second type, the datatype, is for the machine that sends or receives this content. Some examples are arrays, strings, numbers, dates, lists of strings etc. Given that the amount of datatypes is reasonably small and that computers need to know exactly what type of data is on the stream before they can parse it, we assume that this set is fixed. The instances of the contenttypes need not to be enumerated (like enumerating all possible cities for the type 'cityname'. In the next paragraph we give an example on how we deal with unknown instances provided at runtime.)

- **Only primitive datatypes are shared, not the content types or instances** We assume that the primitive datatypes are known by all webservices, but not the content types and instances. As we will show in the example, our approach is based on coincidental occurrences of the terms via literal string matching with terms appearing in ontologies. This means that we drop the assumption that if two or more webservices communicate, they need to be annotated with terms from a shared ontology.

- **The terms in a description are not embedded in a structure on which to do inference**— Since the contenttypes are not annotated in any formal structure, no automated reasoning is possible. For example, if one would know that a webservice could find pictures of type ANIMAL, and the system 'knows' that a DOG is an ANIMAL, queries about dogs can be appropriately directed to the animal picture service. As an alternative to this predetermined way of reasoning we will propose a technique that discovers these kind of mappings via user-feedback.

To explain our approach, we first start with an example that exactly follows it. Imagine one web-service that compiles a C source[5] file into executable code for a specific CPU type, and another web-service that executes code for a given CPU type:

CCompilerService

INPUT		
	DESCRIPTION	compiles a C source file into a CPU dependent binary
	CONTENTTYPE	SOURCE_CODE, CPU_TYPE
	DATATYPE	FILE, STRING
OUTPUT		
	DESCRIPTION	executable code together with the CPU type
	CONTENTTYPE	EXECUTABLE_CODE, CPU_TYPE
	DATATYPE	FILE, STRING

SoftwareExecutionService

INPUT		
	DESCRIPTION	executable code together with the CPU type
	CONTENTTYPE	EXECUTABLE_CODE, CPU_TYPE
	DATATYPE	FILE, STRING
OUTPUT		
	DESCRIPTION	status of execution
	CONTENTTYPE	LOG_FILE
	DATATYPE	FILE

We can construct a simple process flow between two services, where the output of the first is the input of the second. We can instantiate it by giving a piece of C source code a 80386 processor type. The output of the first service will be the executable code in the 80386 instruction set. We assume that the second webservice can only execute code for

[5]C is a programming language

the Pentium CPU types. Upon receipt of the request, service 2 has to decide whether and how it can execute the code provided by service 1 for the given CPU type. Note that service 2 does initially not know what a 80386 processor type is.

Exploiting the vast amount of information on the Internet, a mapping service perhaps could find out an ontology containing a statement that relates the two instances 80386 and Pentium:

isBackwardsCompatibleWith(Pentium, 80386)[6]

Now, the user can be presented with the above statement and is asked to verify whether '80386' can be replaced by 'Pentium'. If so, the system can subsume that the 'isBackwardsCompatibleWith' relation in that ontology is applicable as a mapping rule for the CPU_TYPE field in this interaction. Note that the rule has no formal meaning to the system, except that any X linked via the relation 'isBackwardsCompatibleWith' Y in this ontology can be replaced by this Y. This rule does not apply for all contexts, for example imagine a buying scenario, where you normally don't want a 80386 when you order a Pentium. Therefore, this rule should be associated only with the specific ontology and process flow.

After this example, we give an informal description of the process of *finding mapping rules by example*.

First we have to make the assumption that we have access to a large set of ontologies that connect terms via a path of relations and terms that lie between them. We need these ontologies to find for two terms a path which then will be added to an interaction description between the services for which a mapping is needed (which we call a mapping rule). Mappings are needed when the content type or instances of these types for the output of one web-service, is not literally identical to the expected input of another web-service. More precisely, we identify two situations when mappings are necessary:

- **Content type matching at design time** Imagine that a developer has a task in mind that involves a process flow between several web-services. Assume that this process-flow is written down in a machine executable form, which we call an *interaction description*. At the moment when an interaction description is constructed between two web-services, the content-type of the output of service 1 may not be identical to the input of a service 2.

- **Content type instance matching at runtime** During execution of the web-service composition which is described by an interaction description, it may happen that although for two web-service the content-types match but one service gets an instance that it does not know of. For example, a service may have as "CITY" as input content type, and gets from another service "Newyork" as an instance. Given that we do not assume that we always have enumerations of instances in the interaction description, this instance matching problem can occur at runtime.

For both the above described situations, we will include user feedback to retrieve the content type matching and instance matching. This works as follows:

[6]the Pentium generation of CPUs is backward compatible with the 80386 instruction set, which means that all code compiled for the 80386 can be executed on any Pentium processor.

1. First we need a matching problem which, as said, can occur at design- or runtime. A matching problem in our case means that a web-service receives a content-type or an instance that it does not know of literally or via already existing matching rules. Thus, imagine that x is such a content-type or instance.

2. Second, it makes a set of combinations $C(\{x, y_1\}, \{x, y_2\}, ...\{x, y_n\})$ where the set $\{y_1, y_2, ..., y_n\}$ are the content types or instances that it normally would expect for the given function call.

3. Third, for each pair (x, y) from the above step a search is applied on the ontologies that are known to find a path $path(x, y)$ between x and y. Later in this document we formally describe the several possible path types that may be found and the restrictions that we will make during the experiments. For now, we simply assume that we found a direct relation r between x and y: $r(x, y)$. For example, imagine that $x = 80836$ and $y = pentium$ that both occur in some ontology O connected via the direct relation with the label 'isBackwardsCompatibleWith, $isBackwardsCompatibleWith(80836, pentium)$.

4. Fourth, some human in the process, for example the developer of the interaction description or the invoker of the interaction receives the question:

   "Would it be possible in this situation to replace '80836' by 'Pentium'? The reason for this question is that I found a relation 'isBackwardsCompatibleWith' between them." When the user approves, the relation will be added as a matching rule in the interaction description $O : isBackwardsCompatibleWith(80836, pentium)$.

5. Fifth, a confidence function that could be based on expiration dates and majority votes, determines if in the next time the same question will be posted to a user or if it assumes the mapping rule to be correct. One can decide that instead of only replacing x and y by the mapping rule **any** term connected via the relation is a correct replacement. In other words, the mapping rule can be generalized. For example, image that the example ontology besides

   $isBackwardsCompatibleWith(80836, pentium)$

   also contains the relation

   $isBackwardsCompatibleWith(amd\_duron, amd\_athlon)$.

   The mapping rule would, after generalizing over the parameters of the relation, apply on mapping 'amd_duron' with 'amd_athlon'.

Currently, we are working on formalizing these steps and come up with a realistic set of interaction models to test the approach.

9 Further Work and Conclusions

9.1 Local Good Enough Answers

This document describes how semantic matching at the element level can be composed to allow semantic matching at node and then tree level, and how these matchings can be

enhanced to allow not merely exact matching but also approximate matching. However, Section 6 illustrates that an approximate semantic tree matching can exist between trees that are extremely dissimilar. In order for this approximate semantic tree matching to be useful to us, we need to determine a notion of the quality of the matchings, which will then allow us to determine which, of a set of possible matchings, is the 'best' match (according to criteria which have to be defined) and whether this best match can be considered 'good enough'. These calculations produce what we call *local good enough answers*: the score that is assigned by a peer to its map to a particular constraint. In order to determine which peer should play a role in an IM, many of these local GEAs will usually be necessary. Every peer that may potentially play the role must find and score the best map to each constraint in the role, and these scores will be used to judge which peer should be chosen for each role.

The next work to be done is therefore to determine how such scores can be calculated and the criteria against which they are to be judged. The algorithms for approximate semantic tree matching must be developed so that they are able to find scores for the quality of matching at each node and then compose these scores in a manner that reflects the structure of the tree. The score for each node is therefore a combination of the score for the semantic match at that node (e.g., is it equivalence? is it subsumption - in which case in what direction and to what degree?) with a weight reflecting the importance of the node. The importance of the node is determined primarily by its position in the tree but this value may be overridden by a user preference weighting: perhaps the position of the node would indicate that it is not particularly important to the matching but it represents an attribute that is particularly important to the user.

A key research issue for the next deliverable is how these weights will be determined: to what extent do we penalise imperfect matches and how do we factor in the position of the nodes in the tree. We believe that nodes at a higher level have a larger bearing on the quality of the match than those at a lower level: for example, if the predicate names have no match, the match should be judged to be of very low quality, since it is unlikely that the predicates are referring to the same thing; however, if the predicate names have a perfect match but one child has no match, this probably indicates a good match but with a missing attribute. Therefore, the weight assigned to a node should depend on the weight of its parent. However, it must be determined how these weights degrade: linearly; quadratically; exponentially.

9.2 Global Good Enough Answers

Before an interaction can commence, an IM that matches the description provided by the user must be chosen and the best peers must be chosen to play each of the roles, ensuring that these best meet some 'good enough' standard. The process by which this is done is referred to as *global good enough answers*. Potentially suitable IMs are chosen through matching their descriptions (in natural language or keywords) against the description entered by the user. We can then determine whether it is sensible to initiate them by determining whether there are reliable peers who are able to perform the roles and, if so, assigning the roles to them. There are two factors in deciding how well a peer can perform a role. The first is the peer's own feedback about how well it can satisfy the constraints

of the role, which is a combination of the local GEA matching that peer has performed on the role description. Secondly, since it is impossible to verify that the peer has given this score honestly as the score can only be derived through access to the peer's personal ontology, and since we cannot be sure that the peer will honour the commitment and not leave the interaction halfway through, or behave in some other irresponsible manner, we must factor in information about the trust and reputation of the peer. This can be gleaned from the history of interaction of the peer and will be discussed further in other deliverables. These scores will be combined to give an overall idea of how likely the peer is to perform the role well: for example, a peer with an excellent matching score but a very low reputation is not likely to be believed; a peer with a lower score but a good reputation would be preferred. If there are any roles in the interaction for which peers with a high enough combined local GEA and reputation score cannot be found, the interaction cannot commence.

Note that this matching process includes any peers that are acting on behalf of the user: for example, a *buying* peer when the user wishes to buy something. The user's peer(s) must also perform local GEA matching to determine how well adapted the IM is to their knowledge and of course a low score here will be likely to lead to the rejection of the IM. If the user wishes to have some roles performed by its own peers, this will affect the global GEA process. Other peers will not be considered for that role (for example, it makes no sense for someone else's buying peer to buy items even if that peer is able and willing to do so: the items must be bought for the user). Additionally, trust and reputation may not be a factor for a user's peer: it is to be assumed that the user's peer will behave in a responsible manner in this situation.

9.3 Conclusions

This document forms the basis for the matching aspect of the OpenKnowledge system. This process is of limited practical value until it has been enhanced by the good enough answers aspect of the system. Ideas as to how this will be done have been presented in this document, and further details of the process, together with preliminary results, will be presented in deliverable 4.4.

References

[1] Z. Aleksovski, M. Klein, W. ten Katen, and F. van Harmelen. Matching Unstructured Vocabularies using a Background Ontology. In S. Staab and V. Svatek, editors, *Proc. of EKAW*, LNAI. Springer-Verlag, 2006.

[2] S. Banerjee and T. Pedersen. Extended gloss overlaps as a measure of semantic relatedness. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 267–270, 2003.

[3] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. Contextualizing ontologies. *Journal of Web Semantics*, 1(4):24, 2004.

[4] L. Ding, R. Pan, T. Finin, A. Joshi, Y. Peng, and P. Kolari. Finding and Ranking Knowledge on the Semantic Web. In *Proc. of ISWC*, 2005.

[5] J. Euzenat (Coordinator). State of the art on ontology alignment. Knowledge Web deliverable, D2.2.3, 2004.

[6] F. Giunchiglia and P. Shvaiko. Semantic matching. *The Knowledge Engineering Review Journal*, (18(3)):265–280, 2003.

[7] Fausto Giunchiglia, Maurizio Marchese, and Ilya Zaihrayeu. Encoding Classifications into Lightweight Ontologies. In *Proceedings of the European Semantic Web Conference (ESWC)*, pages 448–453, 2006.

[8] Andreas Heß, Eddie Johnston, and Nicholas Kushmerick. ASSAM: A tool for semi-automatically annotating semantic web services. In *3rd International Semantic Web Conference (ISWC2004)*, Hiroshima,Japan, 2004.

[9] H.H.Do and E. Rahm. COMA - a system for flexible combination of schema matching approaches. In *Proceedings of Very Large Data Bases Conference (VLDB)*, pages 610–621, 2001.

[10] G. Hirst and D. St-Onge. Lexical chains as representation of context for the detection and correction malapropisms. In *C. Fellbaum, editor, WordNet: An electronic lexical database and some of its applications. Cambrige, MA: The MIT Press.*, 1997.

[11] J.J. Jiang and D.W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. In *Proceedings of the International Conference on Research in Computational Linguistics*, 1998.

[12] Henry Kucera and W Nelson Francis. *Computational analysis of present-day American English*. Providence, Brown University Press, 1967. http://CEUR-WS.org/Vol-128/.

[13] Claudia Leacock, Martin Chodorow, and George A. Miller. Using corpus statistics and wordnet relations for sense identification. *Computational Linguistics*, 24(1):147–165, 1998.

[14] Dekang Lin. An information-theoretic definition of similarity. In *ICML*, pages 296–304, 1998.

[15] V. Lopez, M. Sabou, and E. Motta. Mapping the Real Semantic Web on the Fly. In *Proceedings of the International Semantic Web Conference (ISWC)*, 2006.

[16] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. *The Very Large Databases (VLDB) Journal*, pages 49–58, 2001.

[17] George A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[18] S. Patwardhan, S. Banerjee, and T. Pedersen. Using measures of semantic relatedness for word sense disambiguation. 2003.

[19] Siddharth Patwardhan and Ted Pedersen. Using wordnet based context vectors to estimate the semantic relatedness of concepts. In *Proceedings of the EACL 2006 Workshop Making Sense of Sense - Bringing Computational Linguistics and Psycholinguistics Together*, pages 1–8, 2006.

[20] Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. In *IJCAI*, pages 448–453, 1995.

[21] M. Sabou, C. Wroe, C. Goble, and H. Stuckenschmidt. Learning domain ontologies for semantic web service descriptions. *Journal of Web Semantics 3(4)*.

[22] Hinrich Schütze. Dimensions of meaning. In *SC*, pages 787–796, 1992.