Improved Matchmaking Algorithm for Semantic Web Services Based on Bipartite Graph Matching

Umesh Bellur, Roshan Kulkarni
Kanwal Rekhi School of Information Technology, IIT Bombay
umesh@it.iitb.ac.in, roshan@it.iitb.ac.in

Abstract

The ability to dynamically discover and invoke a Web Service is a critical aspect of Service Oriented Architectures. An important component of the discovery process is the matchmaking algorithm itself. In order to overcome the limitations of a syntax-based search, matchmaking algorithms based on semantic techniques have been proposed. Most of them are based on an algorithm originally proposed by M. Paolucci, et al. [21].

In this paper, we analyze this original algorithm and identify some correctness issues with it. We illustrate how these issues are an outcome of the greedy approach adopted by the algorithm. We propose a more exhaustive matchmaking algorithm, based on the concept of matching bipartite graphs, to overcome the problems faced with the original algorithm. We analyze the complexity of both the algorithms and present performance results which show that our algorithm performs as well as the original.

1 Introduction

Loose Coupling is an important principle underlying Service Oriented Architectures. One aspect of loose coupling is the ability to invoke a service provider with little (or no) prior knowledge about it. The *publish-find-bind* architecture is intended to facilitate this process. Service providers create WSDL [9] descriptions and publish them to UDDI [8] registries. Clients search the registry to locate providers of the desired service. Today, in most cases, the WSDL is compiled into client-stubs and the service is invoked. This approach, however, has several limitations.

The WSDL is a specification of the messaging syntax between the client and the provider. It is necessary for a human to interpret the WSDL and then invoke the client-stub with the correct parameters.

The search capabilities of UDDI are limited to a syntax-based search. A client can search the registry for a *string* in the service description or it can search the service classification hierarchy (like NAICS [3]) in the TModel. Neither of these techniques are sufficient, for a client, to be able to autonomously choose a service provider and invoke it without human intervention.

In order to overcome these limitations, techniques for semantic description and matchmaking of services have been proposed in recent literature. These techniques use semantic concepts from *Ontologies* to describe the Inputs, Outputs, Pre-conditions and Effects (IOPE) of a service. The discovery process involves the matchmaking of the semantic descriptions offered by the client and the provider.

In this paper we analyze the semantic matchmaking algorithm proposed by Paolucci, et al. [21]. We have considerable interest in this algorithm because it has been cited extensively in recent literature and several subsequent proposals ([13], [22], [16], [14]) are based on it.

The outline of the paper is as follows: First, we present the algorithm by Paolucci [21]. We then present counter-examples where this algorithm does not generate correct outcomes. We describe our own matchmaking algorithm which overcomes these correctness issues. Finally, we analyze the complexity of the two algorithms and present some experimental results in order to compare their performance.

2 Background and Related Work

Ontologies are used in order to incorporate semantics in web service descriptions. An *Ontology* models domain knowledge in terms of *Concepts* and *Relationships* between them. OWL [12] [11] has evolved as a standard for representation of ontologies on the Web.

OWL-S [18] [11], formerly called DAML-S [10], defines an ontology for semantic web services. OWL-S describes a service in terms of its *Service Profile*, *Process Model* and *Grounding*. The *Service Profile* models the Inputs, Outputs, Pre-conditions and Effects (IOPE) of the service. The Inputs and Outputs in the *Service Profile* refer to concepts in ontologies published on the Web. Service *Advertisements* and search *Queries* are both expressed in terms of OWL-S descriptions.

An ontology *reasoner* is an important component in the process of semantic matchmaking. A reasoner can infer additional information which has not been explicitly stated in an ontology. Subsumption, concept satisfiability, equivalence and disjointness are some examples of reasoning operations. Many of these operations are used in the semantic matchmaking process. DAML-S is based on a logic formalism called *Description Logics* (DL). Description Logics and its reasoning are explored in detail by [15] and [19]. Racer [7] and Pellet [5] [23] are some implementations of DL-Reasoners.

The *Service Profile* contains enough information for a matchmaker to determine if a service satisfies the requirements of a client. In fact, several matchmaking algorithms rely only on the matching of Inputs and Outputs of the *Service Profiles*. One such algorithm has been proposed by M. Paolucci, et al., in [21]. Several extensions to this algorithm have been subsequently proposed by [13] [22] [16] and [14].

Phatak [22] adds *ontology mappings* and QoS constraints to the algorithm from [21]. Choi [13] expands the search scope of [21] by the use of analogous terms from an ontology server. It also makes use of a rule-based search in order to apply user restrictions and to rank search results. It computes fine-grained rankings by the use of *concept similarity* (horizontal and vertical closeness between concepts). Jaeger [16] extends the work from [21] by using matching over the *properties* and over the *Service Profile* hierarchy. It offers a better (fine-grained) ranking scheme as compared to [21].

2.1 Semantic Matchmaking Algorithm

This section describes the algorithm by Paolucci [21]. The algorithm takes a OWL-S *Query* from the client as input and iterates over every OWL-S *Advertisement* in its repository in order to determine a match. An *Advertisement* and a *Query* match if their Outputs and Inputs, both, match. The algorithm returns a set of matching advertisements sorted according to the degree of match.

Let $Query_{out}$ and $Advt_{out}$ represent the *list of output concepts* of the *Query* and *Advertisement* respectively. Matching the outputs requires the matching between two *concept-lists*, $Query_{out}$ and $Advt_{out}$, as follows:

$$\forall c \in Query_{out}, \exists d \in Advt_{out}, \\ \text{s.t. } match(c, d) \neq Fail$$

Let $Query_{in}$ and $Advt_{in}$ represent the list of input *Concepts* of the *Query* and *Advertisement* respectively. Matching the inputs requires:

$$\forall c \in Advt_{in}, \exists d \in Query_{in}, \\ \text{s.t. } match(c, d) \neq Fail.$$

Note that the order of *Query* and *Advt* has been transposed between the two expressions above. Suppose $outQ \in Query_{out}$ and $outA \in Advt_{out}$ are two concepts. In case of output matching, the $match(outQ, outA)$ function accepts $outQ$ and $outA$ as inputs and returns the degree of match between them. Four degrees of match are defined between a them:

- ***Exact***: If $outA$ is an equivalent concept to $outQ$ or $outA$ is a superclass of $outQ$. In case of a superclass relationship, it is assumed that the service provider has agreed to support *every* possible subclass of $outA$.
- ***Plugin***: If $outA$ Subsumes $outQ$. The relation between $outA$ and $outR$ is weaker as compared to the previous case since subsumption is indirectly inferred by the reasoner. It is assumed that the provider has agreed to support *some* sub-concepts of $outA$. We hence infer that $outA$ can be *plugged in place of* the required $outR$.
- ***Subsume***: If $outQ$ Subsumes $outA$. The set of individuals defined by the concept, $outA$, is a subset of the set of individuals defined by the concept $outR$.
- ***Fail***: If none of the above conditions are satisfied.

These four degrees as ranked as: *Exact* $>$ *Plugin* $>$ *Subsumes* $>$ *Fail*. Here, $x > y$ indicates that x is ranked higher (is a more desirable match) than y.

Greedy Approach: The algorithm adopts a greedy approach towards matching the concept-lists. For example, in the case of output matching, for each concept in the $Query_{out}$, it determines a corresponding concept in $Advt_{out}$ to which it has a *maximum* degree of match. Once all such max-matchings are computed, the minimum match amongst them is the *overall degree of match* between the *Query* and the *Advertisement*.

3 Analysis

In this section we analyze the algorithm [21] from the perspective of correctness. We present counter-examples where the algorithm does not generate correct outcomes.

3.1 Degree of Match

[21] assumes that if an advertisement claims to output a certain concept, it commits itself to output every *SubClass* of that concept. We believe that such an assumption is detrimental to the effectiveness of the matchmaker because of the following reasons:

- In a real-world scenario, a provider for, say *Vehicle*, is likely to sell *some* types of *Vehicle*, but not *every* type of vehicle.
- This assumption encourages the advertisers to advertise more generic concepts. For instance, an advertiser claiming to output *Everything* – e.g. *owl:Thing* – will have a *Plugin* match with every Query. This is undesired since a malicious advertiser can poison the search results. The genuine advertisements will be overwhelmed by the large number of such malicious advertisements.
- In the present architecture, the semantic notions exist only in the matchmaking layer. Subsequent stages, like grounding or service invocation, deal with syntax.

  Consider an advertisement A, which claims to output *Vehicle*. Assume that this provider can indeed output every type of *Vehicle*.

  A query Q is searching for a service which offers a *StationWagon*. Let us assume that the ontology defines *StationWagon* as a subclass of *Vehicle*. The algorithm returns *'A'* as an Exact match to *'Q'*, using the rules presented earlier.

  Now, the service provider $Grounds$ the concept, $Vehicle$, to a concrete XML message. However, there does not exist any invocation mechanism by which the client can automatically express to the provider that it wants a *Station Wagon* instead of a generic *Vehicle*.

Due to the arguments presented above, we subscribe to an alternative procedure for computing the *match()* as shown in Algorithm-1. This algorithm inverts the concepts of Plugin and Subsumes.

Algorithm 1 PROCEDURE match(outA, outQ)

```
 1: if outA = outQ then
 2:     return Exact
 3: else if outQ superclass of outA then
 4:     return Plugin
 5: else if outQ subsumes outA then
 6:     return Plugin
 7: else if outA subsumes outQ then
 8:     return Subsumes
 9: else
10:     return Fail
11: end if
```

A similar approach has also been proposed in [13]. In this section, we have offered stronger arguments in favour of this approach.

3.2 False Positives and False Negatives

In this section, we present some counter-examples where the results from the matchmaking algorithm of [21] are incorrect. In the description below, we have considered the process of matching *Query* outputs. Similar arguments can also be given for the process of matching the inputs.

The algorithm from [21] iterates over the list of output concepts of the *Query* and it tries to find a max-match to an output concept in the *Advertisement*. Initially, every output concept of the *Advertisement* is a candidate for such a match. We call this set of output concepts of the *Advertisement* as a *candidate list*.

The original algorithm does not specify whether a concept from the *candidate list* is removed once it has been matched. We analyse both the scenarios – with and without the removal of concepts. Both of them yield some incorrect results.

3.2.1 False Positives

Suppose a concept from the *Advertisement* is not removed from the candidate list after it has been matched.

Consider an *Advertisement* for a travel-agent who books *Accomodation* for their customers in the specified *Destination*. The *Advertisement* has the following Inputs and Outputs:

Inputs:	$Destination$
Outputs:	$Accommodation, Cost$

The concepts *Destination, Accommodation, Cost,* are defined in a travel ontology. Some parts of the ontology are illustrated in Fig-1. Consider a *Query* from a client who is planning a vacation. The client wants to make reservations for a $Hotel$ and a $Campground$ at the specified destination. $Hotel$ and $Campground$, both, are subclasses of the concept $Accomodation$. The $Query$ has the following Inputs and Outputs:

Inputs:	$Destination$
Outputs:	$Hotel, Campground$

- The initial candidate list is $\{Accommodation, Cost\}$.
- The algorithm tries to compute a max-match for $Hotel$. Using the rule - $outA$ Superclass $outQ$ - this will be flagged as an Exact match with $Accommodation$.
- The algorithm tries to compute a max-match for $Campground$. Using the same rule, this will be flagged as an Exact match with $Accommodation$

In reality, the $Advertisement$ indeed does not satisfy the $Query$. Such a match is a false positive result.

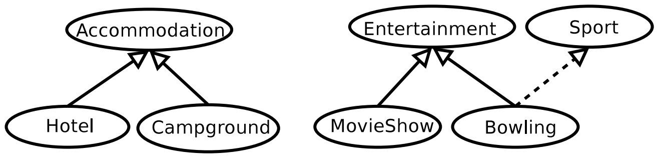

Figure 1. Travel Ontology

Such false positive outcomes can be expected whenever two or more concepts from the $Query$ match a single concept in the $Advertisement$.

3.2.2 False Negatives

We now consider a scenario where a concept is removed from the candidate list after it has been matched with a concept from the $Query$.

Consider an $Advertisement$ for a travel-agent who reserves tickets for two kinds of activities at a holiday destination. The inputs and outputs of the $Advertisement$ are given below:

Inputs:	$Destination$
Outputs:	$Entertainment, Sport$

A client is planning a vacation and desires to make reservations for the following two activities: (i) $Bowling$ (ii) $MovieShow$. The inputs and outputs of the $Query$ are given below:

Inputs:	$Destination$
Outputs:	$Bowling, MovieShow$

The concepts used above are defined in the travel ontology shown in Fig-1. The solid lines indicate the explicitly asserted relationships. The dotted lines indicate the relationships inferred by the reasoner.

- The algorithm will first attempt to compute a max-match for $Bowling$. The candidate list of the $Advertisement$ outputs is: $\{Entertainment, Sport\}$. The following matches are inferred:

$$Bowling \text{ subclass } Entertainment \Rightarrow \text{Exact}$$
$$Bowling \text{ subsumed by } Sport \Rightarrow \text{Plugin}$$

- $Bowling$ has a max-match with $Entertainment$. Hence $Entertainment$ is removed from the candidate list.
- The algorithm now attempts to match the next concept: $MovieShow$. Since $MovieShow \cap Sport = \emptyset$, the match fails.

We now transpose the order of concepts in the output of the *Query* and analyse the behaviour of the algorithm. Consider an alternative $Query$ as:

Inputs:	$Destination$
Outputs:	$MovieShow, Bowling$

- The algorithm first computes a max-match for $MovieShow$. The initial candidate list is: $\{Entertainment, Sport\}$

$$MovieShow \text{ subclass } Entertainment \Rightarrow \text{Exact}$$
$$\{MovieShow \cap Sport = \emptyset\} \Rightarrow \text{Fail}$$

- $MovieShow$ has a max-match with $Entertainment$. Hence $Entertainment$ is removed from the candidate list.
- The algorithm now attempts a match for $Bowling$. Since $Bowling$ is subsumed by $Sport$, it is a Plugin match. The final outcome is thus a Plugin match.

In this scenario we see that the outcome of the matchmaker depends on the order of the concepts in the *Query*. Semantic matchmaking should be agnostic of the syntactic ordering of the concepts in the $Advertisements$ and $Queries$. We therefore believe that a more exhaustive matchmaking process is desired, instead of the greedy approach adopted by this algorithm.

4 Proposed Algorithm

In this section, we propose our matchmaking algorithm based on the notion of matching bipartite graphs. We present some basic concepts on bipartite graphs and then look at the proposed algorithm. Finally, we offer a complexity analysis of the algorithm.

4.1 Bipartite Graphs and Matching

- **Bipartite Graph:** A *Bipartite Graph* is a graph $G = (V, E)$ in which the vertex set can be partitioned into two disjoint sets, $V = V_0 \cup V_1$,such that every edge $e \in E$ has one vertex in V_0 and another in V_1. Fig-2 shows a *weighted* bipartite graph G.

- **Matching:** A *matching* of a bipartite graph $G = (V, E)$ is subgraph $G' = (V, E')$, $E' \subseteq E$, such that no two edges $e_1, e_2 \in E'$ share the same vertex. We say that a vertex v is *matched* if it is incident to an edge in the matching. Fig-2 also shows one such matching G' for the graph G.

  Given a bipartite graph $G = (V_0 + V_1, E)$ and its matching G', the matching is *complete* if and only if, all vertices in V_0 are matched. The matching G' in Fig-2 is not a compete matching since vertex x is not matched.

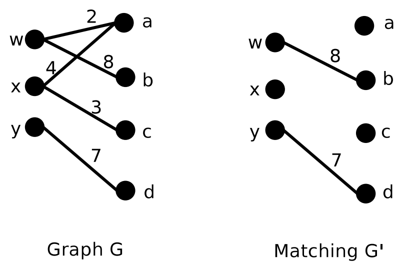

Figure 2. Bipartite Graph and its Matching

4.2 Modelling Semantic Matchmaking as Bipartite Matching

Consider a *Query* Q and *Advertisement* A. We model the problem of matching their outputs as a problem of matching over a bipartite graph. This involves two steps:

- **Constructing a Bipartite Graph:** Let Q_{out} and A_{out} be the set of output concepts in Q and A respectively. These constitute the two vertex sets of our bipartite graph. Construct graph $G = (V_0 + V_1, E)$, where, $V_0 = Q_{out}$ and $V_1 = A_{out}$.

  Consider two concepts $a \in V_0$ and $b \in V_1$. Let R be the degree of match (Exact, Plugin, Subsume, Fail) between concepts a and b. If $R \neq Fail$, we define an edge (a, b) in the graph and label this edge as R.

- **Defining a Matching Criteria:** We compute a *complete matching* of this bipartite graph. A complete matching will ensure that every concept in the output of the *Query* is matched to some concept in the output of the *Advertisement*. We consider the following cases:
  - *Complete matching does not exist:* We infer that the match between the *Advertisement* and the *Query* has failed.
  - *Multiple complete matchings exist:* We should choose a complete matching which is *optimal*. So far we have not defined any optimality criteria from the perspective of a semantic match. We shall define this below.

We assign a numerical weight to every edge in the bipartite graph. The weight of an edge, $e = (a, b)$, is a function of the degree of match between concepts a and b.

Degree of Match	Weight of edge
Exact	w_1
Plugin	w_2
Subsumes	w_3

Also, $w_1 < w_2 < w_3$

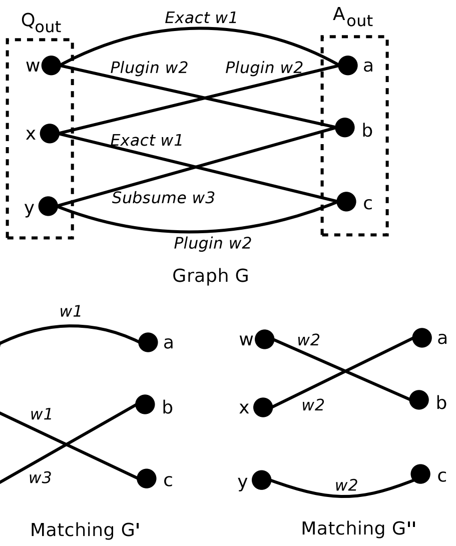

Figure 3. Bipartite Graph of Output Concepts

Fig-3 illustrates a bipartite graph G constructed from Q_{out} and A_{out} using the procedure described earlier. G' is a complete matching of G.

Let $max(w_i)$ denote the maximum weighted edge in G'. The maximum weighted edge represents the worst degree of match between the two vertex sets in G'. This is similar to the notion of *global degree of match* defined in [21]. We therefore say that $max(w_i)$ denotes the *overall degree of match* for G'.

Consider a scenario in which several different matchings exist for the given bipartite graph. An *optimal matching*, from the perspective of semantic matchmaking, is a complete matching in which $max(w_i)$ is minimized.

For example, in Fig-3, G' and G'' are two complete matchings of G as shown in the figure. We can now infer the following:

Matching	max(w_i)	Overall Match
G'	$w_3 \Rightarrow$	Subsume
G''	$w_2 \Rightarrow$	Plugin

Our algorithm chooses the matching in which $max(w_i)$ is minimized. Since $w_2 < w_3$, G'' (Plugin) is chosen over G' (Subsume) as the match.

Our discussion so far has only considered the matching of output concepts. The matching of input concepts is a similar process. In case of outputs, every concept in Q_{out} needs to be matched. Whereas in case of inputs, every concept in A_{in} needs to be matched. Hence we construct a bipartite graph where $V_0 = A_{in}$ and $V_1 = Q_{in}$.

So far we have constructed the graph and defined the matching criteria. In the next section, we shall see how the matching is actually computed.

4.3 Computing the Optimal Matching

The *Hungarian algorithm* ([17], [20]) computes a complete matching of the bipartite graph such that the sum of weights of the edges in the matching, Σw_i, is minimized. The use of Hungarian algorithm for matching bipartite graphs is desired due to its strong polynomial time bound. If $|V|$ is the number of vertices in the graph, the time complexity of the Hungarian algorithm is $O(|V|^3)$. This is more efficient than the combinatorial complexity of any brute-force algorithm.

In our current problem, we wish to compute a matching such that the $max(w_i)$ is minimized. This optimization criteria is different than the one assumed by the hungarian algorithm.

This difference is illustrated in the example from Fig-3. Consider the assignment of weights as: $w_1 = 1$, $w_2 = 2$, $w_3 = 3$. G' and G'' are the two matchings of the graph. We can now compute:

Matching	**max(w_i)**	Σw_i
G'	3 (Subsume)	5
G''	2 (Plugin)	6

Our optimization criteria would choose G'', whereas the hungarian algorithm would choose G' as the optimal match. As a result, the hungarian algorithm cannot be directly used to compute the matching that we desire.

We hence propose a different technique for the assignment of edge weights such that the following *lemma* holds true:

Lemma: A matching in which Σw_i is minimized, is equivalent to a matching in which $max(w_i)$ is minimized

If the above *lemma* holds true, we can indeed use the hungarian algorithm to compute the matching that we desire. We first look at the technique for assignment of edge weights. We then prove that the above *lemma* holds true for the proposed assignment.

In $G = (V_0 + V_1, E)$, the values of the edge weights are computed as follows:

Degree of Match	**Weight**
Exact $\Rightarrow$	$w_1 = 1$
Plugin $\Rightarrow$	$w_2 = (w_1 * \|V_0\|) + 1$
Subsume $\Rightarrow$	$w_3 = (w_2 * \|V_0\|) + 1$

$|V_0|$ = Cardinality of set V_0

We note the following properties, which will be used in the subsequent proof:

- The maximum number of edges in any complete matching of the graph G will be equal to $|V_0|$
- The following relation holds true: $w_1 < w_2 < w_3$
- The above computation of weights enforces that a single edge of a higher weight will be greater than a set of $|V_0|$ edges of lower weights taken together:

$$w_i \geq w_j \times |V_0|, \forall j < i \qquad (1)$$

Proof: We use a proof-by-contradiction method to prove the *lemma* stated earlier.

- Given a graph G, let M be a complete matching in which Σw_i is minimized. Let $(d_1, d_2, d_3, ...)$ denote the set of edges in M.
- Let M' be a complete matching in which $max(w_i)$ is minimized. Let $(e_1, e_2, e_3, ...)$ denote the set of edges in M' and let e_{max} denote the maximum weight edge in this set.
- Assume that the lemma is untrue. Hence, $M \neq M'$.
- Now, there will be at least one edge, $d_M \in M$, such that $w(d_M) > w(e_{max})$. This is because:
  - $M \neq M'$
  - M' is a matching in which $max(w_i)$ is minimized. e_{max} is the maximum weight edge in M'
- $w(d_M) > w(e_{max}) \Rightarrow w(d_M) > w(e_i), \forall e_i \in M'$
- The maximum number of edges in M' is bounded by $|V_0|$. Using previous results and Equation-1 from the previous section:

$$\begin{gathered} w(d_M) > w(e_i), \forall e_i \in M' \\ \Rightarrow w(d_M) > \Sigma w(e_i) \\ \Rightarrow \Sigma w(d_j) > \Sigma w(e_i) \end{gathered}$$

  Here $\Sigma w(e_i)$ and $\Sigma w(d_j)$ denote the sum of weights of all edges in M and M' respectively.
- $\Sigma w(d_i) > \Sigma w(e_i)$ contradicts our assumption that M is a matching having the minimal sum of weights. The contradiction holds as long as we assume that $M \neq M'$. We can hence infer that both M and M' are equivalent.

4.4 Our Algorithm

Algorithm-2 defines the $search()$ procedure. It accepts a $Query$ as input and tries to match it with each advertisement in the repository. A match is computed for both, output and input concepts. If the match is not a *Fail*, it appends the advertisement to the result set. Finally the sorted result set is returned to the client.

The $match()$ procedure in Algorithm-3 accepts two concept-lists as inputs and constructs a bipartite graph using them. It then invokes a hungarian algorithm to compute a *complete matching* on the graph. The $match()$ procedure is invoked twice in $search()$. The order of $Query$ and $Advertisement$ in each call is however swapped.

The $computeWeights()$ function computes the values of w_1, w_2, w_3, depending on the number of concepts in V_0. It uses the formulae presented in the section *"Computation of Edge Weights"* to compute the values. The $degreeOfMatch()$ function is a call to the reasoner in order to determine the relationship between the two concepts a and b.

4.5 Complexity Analysis

Let N denote the number of advertisements in the repository. The average number of input and output concepts in the *Query* are denoted by $|Q_{in}|$ and $|Q_{out}|$ respectively. Similarly, the average number of input and output concepts in the *Advertisement* are denoted by $|A_{in}|$ and $|A_{out}|$ respectively. The complexity of the algorithm is analyzed as follows:

- Search involves iteration over each *Advertisement* in the repository.
- Weights w_0, w_1, w_2 are computed based on $|V_0|$. This is an $O(1)$ operation.
- The graph is constructed by comparing every pair of concepts $(a, b), a \in Q_{out}, b \in A_{out}$. This operation has a complexity of $O(|Q_{out}| \times |A_{out}|)$.
- The time complexity of hungarian algorithm is bounded by $|Q_{out}|^3$

The above matching is done twice: Once for outputs, once for inputs. We can thus compute the time complexity of a search as:

$$N \times \left\{ (|Q_{out}| \times |A_{out}| + |Q_{out}|^3) + (|A_{in}| \times |Q_{in}| + |A_{in}|^3) \right\}$$

We can now approximate, $|Q_{out}| = |A_{out}| = |Q_{in}| = |A_{in}| = m$. Here, m is independent of the number of advertisements in the repository and is likely to take small integer values (usually 1 to 15). We can hence consider m to be a constant. The time complexity of search is simplified:

$$O\big(N \times 2 \times \{m^2 + m^3\}\big) = O\big(N\big) \tag{2}$$

Although the time complexity expressed here is $O(N)$, the constant factors involved are quite high. For $m = 10$, for instance, $m^3 = 1000$.

Algorithm 2 search($Query$)

```
$Result$ = Empty List

for each $Advt$ in Repository do
    $outMatch$ = match($Query_{out}$, $Advt_{out}$)
    if ($outMatch$ = Fail) then
        Skip $Advt$. Take next $Advt$.
    end if

    $inMatch$ = match($Advt_{in}$, $Query_{in}$)
    if ($inMatch$ = Fail) then
        Skip $Advt$. Take next $Advt$.
    end if

    $Result$.append($Advt$, $outMatch$, $inMatch$)
end for

return sort($Result$)
```

Algorithm 3 match($List_1$, $List_2$)

```
Graph G = Empty Graph $(V_0 + V_1, E)$
$V_0 \leftarrow List_1$
$V_1 \leftarrow List_2$
($w_1$, $w_2$, $w_3$) $\leftarrow$ computeWeights($|V_0|$)

for each concept $a$ in $V_0$ do
    for each concept $b$ in $V_1$ do
        $degree$ = degreeOfMatch($a$, $b$)
        if $degree \neq$ Fail then
            Add edge $(a, b)$ to $G$
            if ($degree$ = Exact) then $w(a, b) = w_1$
            if ($degree$ = Plugin) then $w(a, b) = w_2$
            if ($degree$ = Subsume) then $w(a, b) = w_3$
        end if
    end for
end for

Graph $M$ = hungarianMatch($G$)
if ($M$ = null) then
    No complete matching exists
    return Fail
end if

Let $(a, b)$ denote Max-Weight Edge in $G$
$degree \leftarrow$ degreeOfMatch($a$, $b$)
return $degree$
```

4.5.1 Complexity Comparison

The algorithm from [21] iterates over all the advertisements in the repository and performs matching over both, inputs and outputs. If we assume that concepts are not removed from the *candidate-list* after a match, the time complexity of the algorithm can be expressed as:

$$N \times \big\{(|Q_{out}| \times |A_{out}|) + (|A_{in}| \times |Q_{in}|)\big\}$$

Using simplifications similar to the above, we get:

$$O\big(N \times 2 \times \{m^2\}\big) = O\big(N\big) \tag{3}$$

Comparing (2) and (3) we conclude that both (the original and the proposed algorithms) have linear time complexity. However the constants involved in our proposed algorithm are larger.

5 Implementation

The following algorithms were implemented in Java in order to compare their correctness and performance:

1. Our *Bipartite Matching* algorithm
2. *Greedy matchmaking algorithm* by Paolucci [21]
3. *Brute-Force* matching algorithm

The *Brute-Force* algorithm *exhaustively* compares every possible matching between the concept lists. It then chooses a matching with the best overall degree of match. The *Brute-Force* algorithm, due to its exhaustive nature, will serve as a reference model to compare the correctness of the *Greedy* and the *Bipartite* algorithms.

Our implementation is illustrated in Fig-4. The Protege editor [6] is used to browse and edit OWL ontologies. We load the OWL ontologies into the *KnowledgeBase* defined by the Mindswap OWL-S API [2]. This API is also used to parse the OWL-S *Queries* and *Advertisements*. We use the *Pellet* reasoner [5] [23] to *classify* the loaded ontologies.

The Jena API [1] is used to query the reasoner for concept relationships. In order to compute matchings for bipartite graphs, we use an implementation of the Munkres-Kuhn (Hungarian) algorithm by [20].

6 Correctness and Performance Comparison

The OWLS-TC (service retrieval test collection from SemWebCentral) [4] is used to compare the algorithms. We use 7 ontologies (2449 concepts) from OWLS-TC in our *KnowledgeBase*. About 350 advertisements from OWLS-TC were loaded into our advertisement repository.

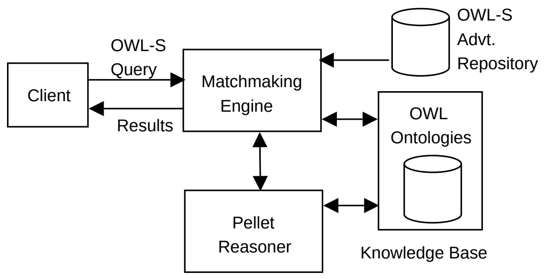

Figure 4. Implementation

6.1 Correctness

False Positives: We first test the occurence of false positives. In this case, we use a greedy algorithm which does not remove concepts from the *candidate list*. The following *Query* from OWLS-TC is matched against the advertisement repository:

Inputs:	Book
Outputs:	TaxedPrice, Price

The number of matches flagged by the three algorithms is shown below:

-	Exact	Plugin	Subs.	Fail	Total
Greedy	1	0	5	344	350
Brute F.	1	0	0	349	350
Bipartite	1	0	0	349	350

The results of the *Bipartite* and the *Brute-Force* algorithm are identical. The *Greedy* algorithm has flagged 5 subsume matches. These matches are the false positive outcomes. These matches have conditions identical to those illustrated in section 3.2.1 earlier.

False Negatives: We now test the occurence of false negatives. For this purpose, we use a greedy algorithm which removes concepts from the *candidate list*. OWLS-TC did not have any *Queries* which flagged false negatives. We could however construct a few such *Queries* for the purpose of illustration. At first, 3 *Queries* were constructed using the ontologies in OWLS-TC. Then, 3 additional *Queries* were constructed by merely swapping the order of concepts in the first 3 *Queries*.

Since we search for 6 Queries over 350 advertisements, there would be a total of 6 x 350 = 2100 matchings. Ideally, we expect all the 6 queries to match their corresponding advertisements. The actual results are shown below:

-	Exact	Plugin	Subs.	Fail	Total
Greedy	0	0	3	2097	2100
Brute F.	0	0	6	2094	2100
Bipartite	0	0	6	2094	2100

As seen in the results above, the *Bipartite* algorithm matches all 6 *Queries*. The *Greedy algorithm* however generates 3 false negatives.

We have thus tested that the *Greedy* algorithm indeed generates false positive and negative outcomes. On the other hand, the outcomes of the *Bipartite matching* are identical to that of the *Brute Force* reference model.

6.2 Performance

We determine the search-time of the three algorithms with respect to the number of advertisments in the repository. This is presented in Fig-5 below.

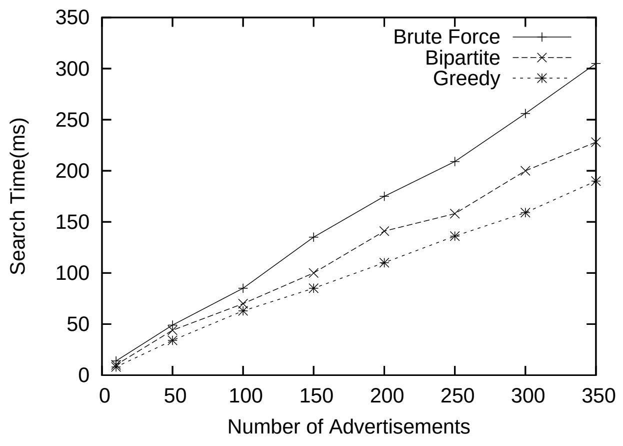

Figure 5. Query Search Time

We observe that the search time of the *Bipartite matching* algorithm is higher than that of the *Greedy algorithm* but less than that of the *Brute force* algorithm. The search time is linear w.r.t. the number of advertisements in the repository. Moreover, the slope of the graph for *Bipartite matching* is slightly higher than that for the *Greedy algorithm*. This is because the *O(N)* expression, in the complexity analysis, has a higher multiplying constant for the *Bipartite matching* algorithm. These results ascertain the claims made in the complexity analysis section earlier.

7 Conclusion

In this paper we have identified the problems with the matchmaking algorithm from [21] and offered an alternative algorithm to resolve these problems. We have also tested the correctness of our proposed algorithm as compared to [21]. Moreover, the *Bipartite matching* algorithm offers a much better performance as compared to that of the *Brute-Force* technique.

Our future work is focused on improving the efficiency of this algorithm. In particular, we would like to reduce the time complexity of the algorithm by reducing the time required for construction of the bipartite graphs.

References

[1] JENA: Java Framework for Building Semantic Web Applications. *http://jena.sourceforge.net/*.

[2] MINDSWAP: Maryland Information and Network Dynamics Lab Semantic Web Agents Project, OWL-S API. *http://www.mindswap.org/2004/owl-s/api/*.

[3] North American Industry Classification System. *http://www.naics.com/*.

[4] OWL-S Service Retrieval Test Collection. Version 2.1. *http://projects.semwebcentral.org/projects/owls-tc/*.

[5] Pellet: An OWL DL Reasoner. *http://pellet.owldl.com/*.

[6] Protege: Ontology Editor and Knowledge-base framework. *http://protege.stanford.edu/*.

[7] RacerPro: OWL Reasoner and Inference Server for the Semantic Web. *http://www.racer-systems.com/*.

[8] Universal Description Discovery and Integration (UDDI). *http://uddi.org/*.

[9] Web Services Description Language (WSDL). *http://www.w3.org/TR/wsdl*.

[10] A. Ankolekar et al. DAML-S Coalition. DAML-S: Web Service Description for the Semantic Web. *ISWC*, 2002.

[11] G. Antoniou et al. Web Ontology Language: OWL. *Handbook on Ontologies in Information Systems*, 2003.

[12] S. Bechhofer et al. OWL Web Ontology Language Reference. *W3C Recommendation: http://www.w3.org/TR/owl-ref/*, 2004.

[13] O. Choi et al. Extended Semantic Web Services Model for Automatic Integrated Framework. *NWESP*, 2005.

[14] R. Guo et al. Capability Matching of Web Services Based on OWL-S. *Proceedings of 16th International Workshop on Database and Expert Systems Applications*, 2005.

[15] I. Horrocks. Reasoning with Expressive Description Logics: Theory and Practice. *18th International Conference on Automated Deduction*, 2002.

[16] M. Jaeger et al. Ranked Matching for Service Descriptions using DAML-S. *Proceedings of CAiSE'04 Workshops*, 2004.

[17] H. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistic Quarterly*, 1955.

[18] D. Martin et al. OWL-S: Semantic Markup for Web Services. *Technical Report, Member Submission, W3C http://www.w3.org/Submission/2004/07/*, 2004.

[19] D. McGuinness et al. The Description Logic Handbook: Theory, implementation and applications. *Cambridge University Press*, 2003.

[20] K. Nedas. Implementation of Munkres-Kuhn (Hungarian) Algorithm. *http://www.spatial.maine.edu/ kostas*, 2005.

[21] M. Paolucci et al. Semantic Matching of Web Service Capabilities. *Springer Verlag, LNCS, International Semantic Web Conference*, 2002.

[22] J. Phatak et al. A Framework for Semantic Web Services Discovery. *WIDM*, 2005.

[23] E. Sirin et al. Pellet: An OWL DL Reasoner. *Journal of Web Semantics*, 2005.