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. [19].

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 based on our implementation of both these algorithms. We show that the complexity of our algorithm is equivalent to that of the original algorithm in spite of the improvements we have made to address the correctness issues.

1. Introduction

Today's implementations of the *Publish-Find-Bind* paradigm that underlies Service Oriented Architectures are centered around syntactic descriptions of service access protocols. Service providers create WSDL [7] descriptions and publish them to UDDI [6] registries. The WSDL is a specification of the messaging syntax between the client and the provider.

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 perform a search using a service classification hierarchy (like NAICS [3]) defined in the TModel. The WSDL is compiled into client-stubs and the service is invoked. The issues apparent in this approach arise from the use of syntax in descriptions as well as in the matchmaking since syntactic approaches limits the scope of the search to an exact match of the strings that make up the client query. Another important downside is the tight coupling between the invoker and provider of a service that this results in since the client is usually only prepared to invoke a service for which it has a precompiled stub. This approach precludes the client from dynamically invoking an equivalent service but whose signature is now slightly different than what it is prepared for.

A solution to this involves upgrading syntactic descriptions to semantic ones and using *Ontologies* rather than strings as the basis for search and matchmaking. Many techniques for semantic description and matchmaking of services have been proposed in recent l iterature. In this paper we analyze the semantic matchmaking algorithm proposed by Paolucci, et al. [19]. We have considerable interest in this algorithm because it has been cited extensively in recent literature and several subsequent proposals ([10], [20], [14], [11]) are based on it. We show that the matchmaking algorithm suffers from some shortcomings and we propose an improved version of it.

The rest of the paper is laid out as follows: First, we discuss the many efforts at semantic matchmaking that have been published and present the algorithm by Paolucci [19] in some detail that is necessary for our analysis. 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

An *Ontology* models domain knowledge in terms of *Concepts* and *Relationships* between them. OWL [9] has evolved as a standard for representation of ontologies on the Web. OWL-S [16], formerly DAML-S [8], defines an ontology for semantic web services.

Both, *Advertisements* and search *Queries* are expressed in terms of OWL-S descriptions. The OWL-S *Service Profile* defines a service in terms of its *Inputs, Outputs, Pre-conditions* and *Effects* (IOPE). The Inputs and Outputs in this tuple contain references to concepts in ontologies published on the Web.

The semantic matchmaking process makes use of several reasoning operations provided by an *Ontology Reasoner*. A reasoner can infer additional information that has not been explicitly asserted in an ontology and can support reasoning operations like equivalence, disjointness, subsumption, concept satisfiability etc. Ontology reasoners and DAML-S are based on a logic formalism called *Description Logics* (DL) [13] and [17]. Racer [5] and Pellet [21] are some implementations of DL-Reasoners.

Several semantic matchmaking algorithms are based on the matching of Inputs and Outputs of the *Service Profiles*. One such algorithm has been proposed by M. Paolucci, et al., in [19]. Various extensions to this algorithm have been subsequently proposed by [10] [20] [14] and [11].

Phatak [20] adds *ontology mappings* and QoS constraints to the algorithm from [19]. Choi [10] expands the search scope of [19] 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 [14] extends the work from [19] by using matching over the *properties* and over the *Service Profile* hierarchy. It offers a better (fine-grained) ranking scheme as compared to [19].

2.1. Semantic matchmaking algorithm

This section briefly describes the matchmaking algorithm by Paolucci [19]. The input to the algorithm is a OWL-S *Query* from the client and the output is a set of matching OWL-S *Advertisements* sorted according to the *degree of match*. The algorithm iterates over every OWL-S *Advertisement* in its repository in order to determine a match for the given *Query*. An *Advertisement* and a *Query* match if their Outputs and Inputs, both, match.

Let $Query_{out}$ and $Advt_{out}$ represent the *list of output concepts* of the *Query* and an *Advertisement* respectively. Matching of outputs is defined as:

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

Let $Query_{in}$ and $Advt_{in}$ represent the *list of input concepts* of the *Query* and *Advertisement* respectively. Matching of inputs is defined as:

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

The $match(c,d)$ function returns the degree of match between the two concepts. For concepts $outQ \in Query_{out}$ and $outA \in Advt_{out}$ the $match(outQ, outA)$ function is defined as:

Table-1

Condition	**match(outQ, outA)**
$outA$ Equivalent to $outQ$	Exact
$outA$ SuperClass of $outQ$	Exact
$outA$ Subsumes $outQ$	Plugin
$outQ$ Subsumes $outA$	Subsume
None of the above	Fail

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

The algorithm adopts a **greedy approach** for matching the concept-lists. For example, in the case of output matching, for each concept $c \in Query_{out}$, it determines a corresponding concept $d \in Advt_{out}$ 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 [19] from the perspective of correctness and present counter-examples where the algorithm does not generate correct outcomes.

3.1. Degree of match

Algorithm [19] assumes that if an advertisement claims to output a certain concept, it commits itself to output every *SubClass* of that concept. This is manifested by the condition: { *outA SuperClass of outQ* ⇒ *Exact* } in Table-1 above. 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* (*owl:Thing*) will have a *Plugin* match with every Query. A malicious advertiser can exploit this fact to poison the search results. The genuine advertisements will be overwhelmed by the large number of such malicious advertisements.
- In the present architecture, 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 a *Vehicle* and 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* and 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 an invocation mechanism by which the client can automatically express to the provider that it wants a *Station Wagon* instead of a generic *Vehicle* in the output.

To overcome the above limitations, we subscribe to an alternative Algorithm-1 for *match()* which inverts the rules for Plugin and Subsume degree of match. A similar approach has also been proposed in [10]. In this section, we have offered stronger arguments in favour of this approach.

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
```

3.2. False positives and false negatives

The algorithm from [19] iterates over the list of output concepts of the *Query* and 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 consider both the scenarios – with and without the removal of concepts – and illustrate counter-examples where the algorithm [19] yields incorrect results. These examples use the original rules (Table-1) to define the degree of match between concepts.

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 its customers at the specified travel destination. The *Advertisement* has the following Output concepts: $\{Accommodation,\ Cost\}$. Fig-1 illustrates a part of the travel ontology which defines these concepts.

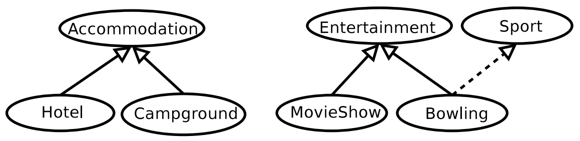

Figure 1. Travel Ontology

Consider a *Query* from a client who wants to make reservations for a $Hotel$ **and** a $Campground$ at the specified destination. The client $Query$ has the following Outputs: $\{Hotel,\ Campground\}$, where $Hotel$ and $Campground$, both, are subclasses of $Accomodation$. The matchmaking algorithm behaves as follows:

- The initial *candidate list* from the Advertisement outputs is: $\{Accommodation,\ Cost\}$ and the list of Query output concepts is: $\{Hotel,\ Campground\}$.
- The algorithm tries to compute a max-match for $Hotel$. Using the rule "$outA$ Superclass of $outQ$" from Table-1 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 $Accommodation$

$Accommodation$ is matched with two concepts from the $Query$: $Hotel$ and $Campground$. The client expects reservation for both – $Hotel$ **and** $Campground$ – whereas the $Advertisement$ offers only a single reservation for an $Accommodation$. This match is a false positive result since the cardinality of the client's request is not being honoured. Guo [12] also asserts that an Input or Output parameter can be used at most once in the matching.

This problem is partially resolved if we adopt the alternative *match()* procedure from Algorithm-1. In that case, the outcome is a Subsume match. A Subsume match, while ranked lower than an Exact or Plugin match, indicates that the provider may help the client achieve its goal. We still consider this to be a false positive outcome.

False positive outcomes, like the one illustrated in this example, can be expected whenever two or more concepts from the $Query$ match a single concept in the $Advertisement$.

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 Outputs of the $Advertisement$ are: $\{Entertainment,\ Sport\}$.

A client who is planning a vacation desires to make reservations for two activities - $Bowling$ and $MovieShow$. The Outputs of the client $Query$ are: $\{Bowling,\ MovieShow\}$.

The concepts used above are defined in the travel ontology (Fig-1). The solid lines indicate the explicitly asserted relationships (SubClass). The dotted lines indicate the relationships inferred by the reasoner (Subsume). Now,

$$Advt_{out} = \{Entertainment, Sport\}$$
$$Query_{out} = \{Bowling, MovieShow\}$$

- The algorithm will first attempt to compute a max-match for $Bowling$. The following matches are inferred:

$$Entertainment \text{ SuperClass of } Bowling \Rightarrow \text{Exact}$$
$$Sport \text{ Subsumes } Bowling \Rightarrow \text{Plugin}$$

- $Bowling$ has a max-match with $Entertainment$. $Entertainment$ is removed from the candidate list.
- The algorithm now attempts to match the next concept: $MovieShow$. Since *match(MovieShow, Sport) = Fail*, *the final outcome is a $Fail$ match.*

We now transpose the order of concepts in $Query_{out}$ and analyse the behaviour of the algorithm. Consider,

$$Advt_{out} = \{Entertainment, Sport\}$$
$$Query_{out} = \{MovieShow, Bowling\}$$

- The algorithm first computes a max-match for $MovieShow$.

$$Entertainment \text{ SuperClass } MovieShow \Rightarrow \text{Exact}$$
$$\textit{match(MovieShow, Sport) = Fail}$$

- $MovieShow$ is matched with $Entertainment$ and $Entertainment$ is removed from the candidate list.
- The algorithm now attempts a match for $Bowling$. Since $Sport$ Subsumes $Bowling$, it is a Plugin match. *The final outcome is thus a Plugin match.*

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 OWL-S $Advertisements$ 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.

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 the other in V_1. Fig-2 shows a *weighted* bipartite graph G.

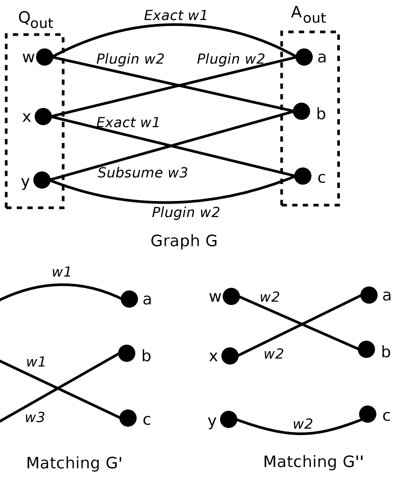

Figure 2. Bipartite Graph of Output Concepts

- **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.

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. 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 them - computed using Algorithm-1. If $R \neq Fail$, we define an edge (a, b) in the graph and label it 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 two cases:
  - *Complete matching does not exist* $\Rightarrow$ *Query* and *Advertisement* do not match.
  - *Multiple complete matchings exist* $\Rightarrow$ We should choose a complete matching which is *optimal*.

Optimal matching: We now need to define an optimality criteria from the perspective of a semantic match. We first assign a numerical weight, w_i, 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

$$w_1 < w_2 < w_3$$

Fig-2 illustrates a bipartite graph G and its complete matching 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'. Similar to the notion of *global degree of match* in [19], we say that $max(w_i)$ denotes the *overall degree of match* for G'. If several different matchings exist for the given bipartite graph, an *optimal matching is a complete matching in which $max(w_i)$ is minimized.* For example, in Fig-2, G' and G'' are two complete matchings of G. We can now infer the following:

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

Since $w_2 < w_3$, G'' (Plugin) is chosen over G' (Subsume) as the optimal match.

The process of matching input concepts is similar to the process of matching output concepts. Since every concept in the input of the advertisement needs to be matched, we construct a bipartite graph where $V_0 = A_{in}$ and $V_1 = Q_{in}$. Here, A_{in} is the set of input concepts in the *Advertisement* and Q_{in} is the set of input concepts in the *Query*.

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* ([15], [18]) 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 compared to the combinatorial complexity of a brute-force algorithm. If $|V|$ is the number of vertices in the graph, the time complexity of the Hungarian algorithm is $O(|V|^3)$.

In our current problem, we wish to compute a matching such that $max(w_i)$ is minimized. This optimization criteria is different from that of the hungarian algorithm. This difference is illustrated in the example from Fig-2. 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 the following:

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. 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 use the hungarian algorithm to compute the desired optimal matching. We first look at the technique for assignment of edge weights and then prove that the above *lemma* holds true for the proposed assignment.

In $G = (V_0 + V_1, E)$, the edge weights are computed as shown in the table below:

Table-2

Degree of Match: match(a,b)	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 take note of 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:

$$w_i > w_j \times |V_0|, \forall i > j \tag{1}$$

Proof of lemma: (Proof-by-contradiction)

- 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 $(e_1, e_2, e_3, ...)$ be the set of edges in M' and e_{max} be the maximum weight edge in this set.
- Assume that the lemma is untrue and hence $M \neq M'$. Since M is not a matching in which $max(w_i)$ is minimized, there will be at least one edge, $d_M \in M$, such that $w(d_M) > w(e_{max})$. Now,

$$w(d_M) > w(e_{max}) \Rightarrow w(d_M) > w(e_i), \forall e_i \in M'$$

- The maximum number of edges in M' is bounded by $|V_0|$. Using previous results and Equation-(1):

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

  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 M and M' will be equivalent if weights are assigned as given in Table-2.

4.4. Our algorithm

The $search()$ procedure in Algorithm-2 accepts a $Query$ as input and tries to match it with each Advertisement in the repository. 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 $matchLists()$ procedure in Algorithm-3 accepts two concept-lists and constructs a bipartite graph using them. It then invokes a hungarian algorithm to compute a *complete matching* on the graph. The $matchLists()$ procedure is invoked twice in $search()$. The order of $Query$ and $Advertisement$ in each call is however swapped.

The $computeWeights(|V_0|)$ function computes the values of w_1, w_2, w_3 as illustrated in Table-2 of the previous section. The $match()$ function computes the degree of match between two concepts as defined in Algorithm-1.

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 $|Q_i|$ and $|Q_o|$ respectively. The average number of input and output concepts in the *Advertisement* are denoted by $|A_i|$ and $|A_o|$ respectively. The complexity analysis follows:

- Search iterates over N *Advertisements*.
- 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 $(a, b), a \in Q_o, b \in A_o$. This operation has a complexity of $O(|Q_o| \times |A_o|)$. The time complexity of hungarian algorithm is bounded by $|Q_o|^3$

The above steps are executed twice - once for output and once for input - of each *Advertisement*. Hence the time complexity of the search is:

$$N \times \left\{ (|Q_o| \times |A_o| + |Q_o|^3) + (|A_i| \times |Q_i| + |A_i|^3) \right\}$$

Algorithm 2 search($Query$)

```
1:  $Result$ = Empty List
2:  for each $Advt$ in Repository do
3:      $outMatch$ = matchLists($Query_{out}$, $Advt_{out}$)
4:      $inMatch$ = matchLists($Advt_{in}$, $Query_{in}$)
5:      if ($outMatch$ = Fail OR $inMatch$ = Fail) then
6:          Skip $Advt$. Take next $Advt$.
7:      else
8:          $Result$.append($Advt$, $outMatch$, $inMatch$)
9:      end if
10: end for
11: return sort($Result$)
```

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

```
1:  Graph G = Empty Graph $(V_0 + V_1, E)$
2:  $V_0 \leftarrow List_1$, $V_1 \leftarrow List_2$
3:  $(w_1, w_2, w_3) \leftarrow$ computeWeights($|V_0|$)
4:
5:  for each concept $a$ in $V_0$ do
6:      for each concept $b$ in $V_1$ do
7:          $degree$ = match($a$, $b$)
8:          if $degree \neq$ Fail then
9:              Add edge $(a, b)$ to $G$
10:             if ($degree$ = Exact) then $w(a, b) = w_1$
11:             if ($degree$ = Plugin) then $w(a, b) = w_2$
12:             if ($degree$ = Subsume) then $w(a, b) = w_3$
13:         end if
14:     end for
15: end for
16:
17: Graph $M$ = hungarianMatch($G$)
18: if ($M$ = null) then
19:     No complete matching exists. return Fail.
20: end if
21:
22: Let $(a, b)$ denote Max-Weight Edge in $G$
23: $degree \leftarrow$ match($a$, $b$)
24: return $degree$
```

We approximate, $|Q_o| = |A_o| = |Q_i| = |A_i| = m$. 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 and the time complexity of search is simplified:

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

The algorithm from [19] 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:

$$N \times \big\{(|Q_o| \times |A_o|) + (|A_i| \times |Q_i|)\big\}$$

Using simplifications similar to the above, we get:

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

We also consider a *Brute-Force* algorithm which *exhaustively* computes every possible matching of the bipartite graph and chooses an optimal matching amongst them. The Brute-Force algorithm has a combinatorial growth and its worst-case time complexity is:

$$O\big(N \times 2 \times m!\big) = O\big(N\big) \qquad (4)$$

It is important to note that although the asymptotic complexity of (2), (3) and (4) are identical, the multiplying constants for the Brute-Force algorithm can be quite higher as $(m! \gg m^3)$ for $m > 6$.

5. Implementation

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

- Our *Bipartite Matching* algorithm
- *Greedy matchmaking algorithm* by Paolucci [19]
- A *Brute-Force* matching algorithm

The *Brute-Force* algorithm is exhaustive nature. It was implemented in order to serve as a reference model to compare the correctness of the *Greedy* and the *Bipartite* algorithms. This implementation of the *Brute-Force* algorithm removes concepts from the candidate-list after a match.

Our implementation is illustrated in Fig-3. 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 [21] 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 [18].

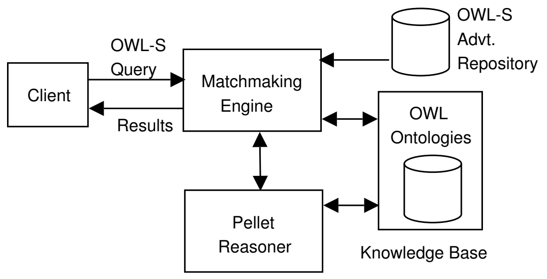

Figure 3. Implementation

6. Correctness and performance comparison

We load 7 ontologies (2449 concepts) and about 350 advertisements from the OWLS-TC (service retrieval test collection from SemWebCentral) [4] in our test setup. The three matchmaking algorithms that are compared here use *match()* from Algorithm-1 to define the degree of match.

6.1. Correctness

False Positives: We use a greedy algorithm which does not remove concepts from the *candidate list*. A *Query* from OWLS-TC is matched against the advertisement repository. This *Query* defines the concept $Book$ as Input and the following concepts as Output: $\{TaxedPrice, Price\}$. The number of matches flagged by the algorithms are:

-	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 and they have conditions identical to those illustrated in section 3.2 earlier.

False Negatives: Here, we use a greedy algorithm which removes concepts from the *candidate list*. First, we construct 3 *Queries* using the ontologies in OWLS-TC. Then, an additional 3 *Queries* were constructed by merely swapping the order of output 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. As seen in the actual results below, the *Bipartite* algorithm matches all 6 *Queries*. The *Greedy algorithm* however generates 3 false negatives.

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

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

6.2. Performance

Fig-4 shows the search-time of the three algorithms w.r.t. the number of advertisments in the repository. The search time of *Bipartite matching* is higher than that of the *Greedy algorithm* but lower than that of the *Brute force* algorithm. The search time is linear w.r.t. the number of advertisements in the repository. This observation is consistent with the complexity analysis presented earlier. In our test data there were a maximum of four concepts in the input or output of any OWL-S advertisement. Thus m was bounded to four. However, in real-world repositories this number is expected to be much higher. The performance of the *Brute force* algorithm would be hence much worse than the one observed here.

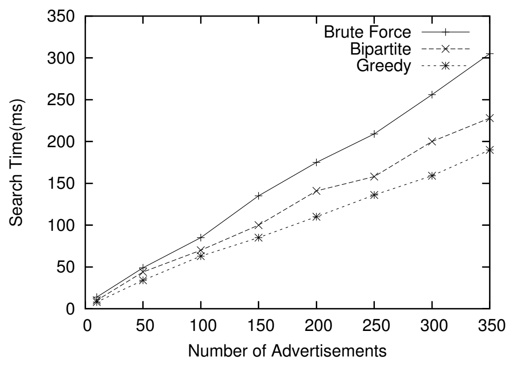

Figure 4. Query Search Time

7 Conclusion

In this paper we identified some problems with the matchmaking algorithm from [19] and offered an alternative algorithm to resolve these problems. Our algorithm offers a correct outcome - equivalent to that of the *Brute force* reference model. Moreover, the time-complexity of our algorithm is equivalent to that of the *Greedy* technique.

Guo [12] has also proposed a matchmaking algorithm based on bipartite graph matching. Their proposal however uses a continuous-valued $similarity$ function to define the edge-weight between two concepts. Our proposal flags discrete degrees of matches using formal logic concepts of equivalence, subclass etc. and thus lends itself to an automated invocation of the discovered web service. We argue that a match flagged by [12], due to its use of continuous-valued $similarity$, cannot be used in formal logic and automated invocation.

Our future work is focused on improving the efficiency of our algorithm by reducing the time required for construction of *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 (NAICS). *http://www.naics.com/*.
[4] OWL-S service retrieval test collection. version 2.1. *http://projects.semwebcentral.org/projects/owls-tc/*.
[5] RacerPro: OWL reasoner and inference server for the semantic web. *http://www.racer-systems.com/*.
[6] Universal Description Discovery and Integration (UDDI). *http://uddi.org/*.
[7] Web Services Description Language (WSDL). *http://www.w3.org/TR/wsdl*.
[8] A. Ankolekar et al. DAML-S Coalition. DAML-S: Web service description for the semantic web. *ISWC*, 2002.
[9] S. Bechhofer et al. OWL Web Ontology Language reference. *W3C Recommendation: http://www.w3.org/TR/owl-ref/*, 2004.
[10] O. Choi et al. Extended semantic web services model for automatic integrated framework. *NWESP*, 2005.
[11] 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.
[12] R. Guo et al. Matching semantic web services across heterogeneous ontologies. *International Conference on Computer and Information Technology*, 2005.
[13] I. Horrocks. Reasoning with expressive Description Logics: Theory and practice. *18th International Conference on Automated Deduction*, 2002.
[14] M. Jaeger et al. Ranked matching for service descriptions using DAML-S. *Proceedings of CAiSE'04 Workshops*, 2004.
[15] H. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 1955.
[16] D. Martin et al. OWL-S: Semantic markup for web services. *Technical Report, Member Submission, W3C http://www.w3.org/Submission/2004/07/*, 2004.
[17] D. McGuinness et al. The Description Logic handbook: Theory, implementation and applications. *Cambridge University Press*, 2003.
[18] K. Nedas. Implementation of Munkres-Kuhn (Hungarian) algorithm. *http://www.spatial.maine.edu/ kostas*, 2005.
[19] M. Paolucci et al. Semantic matching of web service capabilities. *Springer Verlag, LNCS, International Semantic Web Conference*, 2002.
[20] J. Phatak et al. A framework for semantic web services discovery. *WIDM*, 2005.
[21] E. Sirin et al. Pellet: An OWL DL reasoner. *Journal of Web Semantics, http://pellet.owldl.com/*, 2005.