Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 126 (2018) 116–125

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Knowledge Based and Intelligent Information and Engineering Systems, KES2018, 3-5 September 2018, Belgrade, Serbia

Semantic Web Service Composition based on Graph Search

Liana Ţucăr[a], Paul Diac[a,*]

[a]*Faculty of Computer Sicence, Alexandru Ioan Cuza University, General Berthelot, 16, Iasi, 700483, Romania*

Abstract

In Web Service Composition, services or APIs from multiple and independent providers are combined to create new functionality. Web Service Challenges have formalized the problem at a series of events where the community competed with various solutions. The composition problem has been incrementally extended adding more expressiveness and since 2006 the semantic aspect was addressed by the use of ontologies for service parameter description. In this paper, we propose a polynomial-time algorithm based on graph search, that solves the 2008 challenge. The algorithm uses a heuristic to address the `NP-Hard` problem of optimizing the number of services selected. Experimental results show that our solution is several times faster and generates shorter compositions on all evaluation tests compared with all winning solutions of the competition. Also, it is up to 50 times faster and very close to generate shortest compositions on the tests, compared with the solution published in 2011, that generates optimal compositions.

Keywords: Service Composition; Semantic Services; Ontologies; Polynomial Time; Graph; Heuristic; Dynamic Scoring

1. Introduction

Web Services are the core component of Service-Oriented Architecture (SOA), that is gaining increased popularity over the last years. The benefit of using web services traditionally can be extended further by **automated** discovery, composition and execution of aggregate services. Several languages, standards and frameworks have been developed to empower this, such as OWL-S, SAWSDL, WS-BPEL[8], BPEL4SWS, WSMO[9], EFlow[11], SPSC[10], Mad Swan[12] and many others. Also, much research has been devoted to extend the borders of what is theoretically possible but also what is already practically feasible in combining the existent web services, which provide great potential.

The number of publicly available web services is very large and increasing, accelerated recently by growing popularity of microservices and IoT. Moreover, services state might be dynamic. One core principle of SOA is that a service should handle a very limited and specific functionality. The provider of the service should deal more with the availability, scalability, effectiveness of a service, than on allowing a larger functionality of a single service. And while a single entity (company, server, institution) can provide more than a single service, services of the same provider usually share their domain, area or context. Therefore, even just slightly more complex application must use a combination of services. This motivates for the composition of web services.

Web Service Composition is possible if multiple services are already known, for example have been previously discovered and stored in a service repository. Ideally, from composition perspective, these have to adhere to the same

* Corresponding author. Tel.: +04-075-215-3555;
E-mail address: paul.diac@info.uaic.ro

standards, terminology or share the same characteristics. Semantic Web Services for example, describe their definition not only syntactically, but also adding markup for machine-readable semantic meaning of their operation. This can be important and beneficial for automating service composition. Besides the repository, there is also a user query defined by a set of known and a set of required elements. For the user, the elements are atomic pieces of information, that can play the role of service parameters. The user query has the same structure of a web service, that is *missing* from the repository, but there is interest in it. The known elements, or parameters, instantiate service inputs, and some output parameters of services can match the user required elements, based on how the **matching** is defined. In the simplest form, the matching is conditioned by parameter names matching as identical strings, but this is a big limitation. More expressive, parameters can be defined by the use of an ontology. Then matching is defined as subsumation, meaning that a subconcept can replace any of its more generic concepts. Composition requires finding a sequence of services that can naturally respond to the user request. Services within a composition interact with each other and their order is relevant, as parameters are passed from a service to another based on the same matching criteria.

Between 2005 and 2010[2,3,4,5] a special challenge has been organized annually for gathering community ideas, solutions and algorithms, and evaluating them on multiple aspects. Each year the problem was extended to consider more elements, enriching the expressiveness, gradually adding semantics, Quality of Service and using newer standards. Our contribution is a new proposed algorithm for the 2008 version of the problem, which included semantics and parallel service execution. The approach uses a graph-search based polynomial time algorithm, combined with a heuristic to reduce the number of services included in the composition. It is a semantics-aware extension of a previous algorithm[6] that provided excellent results for the 2005 Web Service Composition Challenge[3]. We compared our solution with the results published at the competition in 2008, and also solutions presented later in different publications and the proposed algorithm is much more efficient, running up to 50 times faster on the original benchmark.

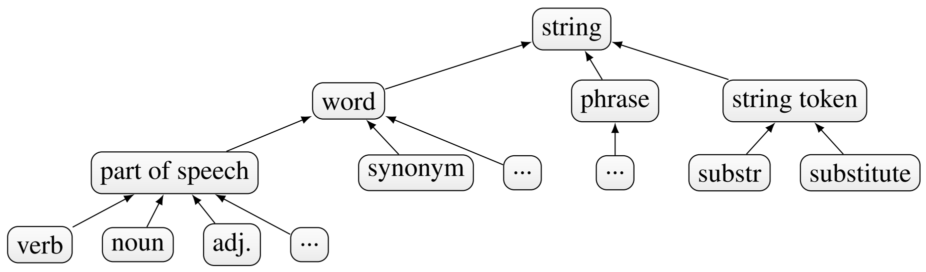
The rest of the paper is organized as follows: Section 2 formally defines the problem starting with a motivating example of this version of semantic composition. The proposed algorithm is presented in detail in Section 3 and evaluated in Section 4. Possible further improvements and extensions are presented in Sections 5 that also concludes the paper.

2. Problem Definition

2.1. Motivating Example

One example motivating semantic service composition, in the domain of Natural Language Processing is described below. Suppose that in some application one needs to replace a phrase's main verb with a synonym of that verb if it exists. The ontology contains concepts in Figure 1, obviously among many others; with the services defining their parameters as concepts within this ontology.

Fig. 1: NLP concepts organized hierarchically. Parent nodes are generalization concepts, substitutable by child subconcepts.



$$extractMainVerb \begin{array}{l} input = \{phrase\} \\ output = \{verb\} \end{array} \qquad getSynonym \begin{array}{l} input = \{word\} \\ output = \{synonym\} \end{array}$$

$$stringReplace \begin{array}{l} input = \{string, substr, substitute\} \\ output = \{string\} \end{array}$$

Starting with the initial *phrase* a call is made to *extractMainVerb* service that returns the *verb* in the sentence. Since a *verb* is a *part of speech* that is, more generally, a *word*, this enables a *getSynonym* service call. The final service call is correct if the parameters are used in the right order: *stringReplace(phrase, verb, synonim)* that would result in the desired modified sentence. Finally, a valid solution of the example is composition: *extractMainVerb*, *getSynonym*, *stringReplace* in order.

2.2. Semantic Composition Problem

The problem is described by paper[2] that presents the 2008 competition. We are going to describe all elements of the same version of the problem, but avoiding less relevant parts such as syntax, file formats and benchmark structure.

Web Service. A web service is defined by two parameter sets: S_{in} - set of input parameters and S_{out} - set of output parameters.

In this version, calling a web service only offers information, without changing the state of the system, itself, or state of the world, as the vast majority of composition problems deal with the stateless services. Even if this is a simplification, the associated optimization problem is hard enough as it is shown later in Subsection 2.3.

Web Service S can be called if all its input parameters S_{in} have been previously obtained and it provides with its output parameters S_{out}. By *calling* a web service, we understand adding it to the composition that is being constructed. Relative to previous steps, output parameters are *learned*, becoming accessible, so later they can eventually be used as input parameters of different services. A repository is a set $\{S_1, S_2, .., S_n\}$ of web services.

Taxonomy. The semantic WSC problem models instances and concepts, organized in a taxonomy. Each concept can have more particular concepts named subconcepts, and instances that are materializations of that concept, for example *23* is an instance of concept *number*. All concepts are direct or indirect subconcepts of a most general concept, e.g. *thing / object*. There are no duplicates in the taxonomy: no two concepts or instances have the same name. Any instance appears only once as being part of a single concept.
The hierarchy of concepts can be represented using a tree, having the concepts as vertices and the children of a vertex are its direct subconcepts. The root of the tree is the most general concept.

Instance Parent Function. Let $parent(instance)$ be a function that for any instance returns the node in the tree corresponding to the most specific concept that contains that instance. By most specific, we understand lowest in the tree or most particularized.

Subsumes. Between taxonomy instances we define the *subsumes* function:

$$subsumes : Instances \times Instances \rightarrow \{false, true\}, where :$$
$$subsumes(a, b) \text{ is } true \iff \text{instance } b \text{ is more specific than } a\text{, or } a.$$
$$\text{or formally:} \iff parent(b) \in \text{subtree of } parent(a) \ or \ \ a = b.$$

More precisely, the vertex associated with $parent(b)$ is a direct or indirect successor of the vertex associated with $parent(a)$ in the tree of concepts determined by the taxonomy or the instances are equal.

Web Service parameter sets, S_{in} and S_{out} are defined over the set of all instances. Parameters are matched with respect to the *subsumes* function. This means that after an instance *b* is generated by the output of a called service, *b* itself and all instances *a* for which *subsumes(a,b)* is *true* can be used as inputs of future services.

Composition. Generally, a *composition* is a structured set of web services that can be validly called starting from a set of initially known instances. All service input parameters have to be known at the time of that service invocation. This means that the set of outputs of previous services in the composition, together with the initially known instances, have to be subsumed by all input instances of the current service. Particularly, the input of the first service in the composition has to be completely subsumed by the initially known instances.

Composition *generated* instances. For one composition *comp*, let $comp_{out}$ be the set of output instances that are obtained after iterating the services in the composition, expanding generalizations.

Let *subsumeSet* be a generalization of the *subsumes* function over sets of instances: $subsumeSet : \mathcal{P}(Instances) \times \mathcal{P}(Instances) \rightarrow \{0, 1\}$, where:

$$subsumeSet(A, B) = \begin{cases} 1, & \text{if } \forall a \in A,\ \exists\, b \in B \mid subsumes(a, b) = 1 \\ 0 & \text{otherwise} \end{cases}$$

A *composition* and its *provided* instances are recursively defined:

1. If service $S = (S_{in}, S_{out}) \in repository$ and $subsumeSet(S_{in}, provided)$ is true, then $Comp = S$ is a composition that has the set of provided instances *provided* and $Comp_{out} = S_{out}$
2. If $C_1, C_2, ..., C_n$ are compositions: for any $1 \leq i \leq n$, C_i starts from $\left(\bigcup_{j \in \{1,...,i-1\}} (C_j)_{out}\right) \bigcup provided$, then $sequential(C_1, C_2, ..., C_n)$ is a composition that starts from *provided* and $Comp_{out} = \cup_{i \in \{1,...,n\}} (C_i)_{out}$
3. If $C_1, C_2, ..., C_n$ are compositions: for any $1 \leq i \leq n$, C_i starts from *provided*, then $parallel(C_1, C_2, ..., C_n)$ is a composition that starts from *provided* and $Comp_{out} = \bigcup_{i \in \{1,...,n\}} (C_i)_{out}$

Client Request. Besides the repository of services, the problem input includes a client request. Syntactically it is similar to a service and we write it (C_{in}, C_{out}). The user or client holds a set of known instances C_{in} and has to obtain a set of desired instances C_{out}.

Satisfying Composition. Composition *comp* satisfies the user request (C_{in}, C_{out}) if and only if the composition starts from the instance set C_{in} and $subsumeSet(C_{out}, comp_{out})$ is true.

Composition size: execution path, number of services.

One valid composition is evaluated based on two parameters:

1. length - the total number of services in the composition.
2. efficiency - the total number of execution steps if parallelism is exploited.

The 1. *numberOfServices*, and 2. *executionPath* of a composition are defined as follows:

1. If $Comp = S$, then
   $numberOfServices(Comp) = executionPath(Comp) = 1$
2. If $Comp = sequential(C_1, C_2, ..., C_n)$, then
   $numberOfServices(Comp) = \sum_{i \in \{1,...,n\}} numberOfServices(C_i)$ and
   $executionPath(Comp) = \sum_{i \in \{1,...,n\}} executionPath(C_i)$
3. If $Comp = parallel(C_1, C_2, ..., C_n)$, then
   $numberOfServices(Comp) = \sum_{i \in \{1,...,n\}} numberOfServices(C_i)$ and
   $executionPath(Comp) = \underset{i \in \{1,...,n\}}{MAX}\, executionPath(C_i)$

SEM-WSC. The Semantic Web Service Composition problem named `SEM-WSC` requires finding a valid composition for input *taxonomy*, *repository* and user request (C_{in}, C_{out}). The corresponding minimization problem requires such a composition with reduced *numberOfServices* and *executionPath*. There is no strict priority between these two, both being evaluated with the same importance in the challenge.

2.3. Complexity

Let `MIN-WSC` be `SEM-WSC` minimization problem prioritizing *numberOfServices*. `MIN-WSC` is known to be `NP-Hard`. The proof is shortly presented further.

`Set-Cover` can be reduced in polynomial time complexity to `MIN-WSC`. Let $\mathcal{F}$ be a function that transforms an instance of the `SET-COVER` problem into an instance of `MIN-WSC`.

Let $\mathcal{S} = \{S_1, S_2, .., S_n\}$ be the collection of sets, and $\bigcup_{S_i \in \mathcal{S}} = \mathcal{U}$ be the universe.

$\mathcal{F}(\mathcal{U}, \mathcal{S}) = ((C_{in}, C_{out}), repository) = ((\emptyset, \mathcal{U}), \mathcal{R})$,

where $\mathcal{R} = \{serv_1, serv_2, .., serv_n | serv_i = (\emptyset, S_i), i \in \{1, 2, ..., n\}\}$

This construction guarantees that if a composition with optimal number of services is found, then `SET-COVER` has optimal solution too. Also, the constructed taxonomy has to contain all instances, S_i in this case, under a single concept.

Tough finding minimum *numberOfServices* is `NP-Hard`, finding shortest *executionPath* is solvable in polynomial time, for example by algorithm 2.

3. Graph Search based Algorithm

Basic idea. The algorithm runs two processes in a loop. The first step, *"construction phase"*, consists in constructing a valid composition, attempting to use as few services as possible. The second step, *"reduction phase"*, removes useless services in the current composition based on some specific criteria. The loop ends when the second step provides no further improvement. At any time, the algorithm maintains a dynamic scoring of the services. Each service score estimates the amount of new information that can be obtained after adding the service to the composition. This score is used to minimize the size of the composition and is computed on the fly.

After selecting a service, its output parameters together with all instances that are more generic than these are learned.

ConstructionPhase. The composition is built layer by layer. Each layer is composed of all the services satisfying the following properties:

1. can be validly added to the composition using the instances provided by the client who makes the request together with all the instances obtained from calling the services from previous layers;
2. provides new information: at least one of its output instances is not provided by another service already in the composition or by the client as known instances.

This construction is performed until all the requested instances from the C_{out} set are obtained or until no service can be added at the composition. If the construction stopped in the first case, then a valid composition is obtained that satisfies the request with a minimum number of execution steps. If the construction stopped in the second case, it means that no valid composition can satisfy the client's request.

Formally, each layer is a set of services with the following property:

$$\underset{i \geq 1}{layer_i} = \left\{ serv \in repository \;\middle|\; \begin{array}{c} subsumeSet(serv_{in}, owned) \text{ is true} \\ \text{and } serv_{out} \nsubseteq owned \end{array} \right\}, \text{ where}$$

$$owned = \left\{ inst \in Instances \;\middle|\; \begin{array}{c} \exists\, outI \in \left(\bigcup_{serv' \in layer_j}^{j=\{1,\ldots,i-1\}} serv'_{out} \cup C_{in} \right) \\ \text{and } subsumes(inst, outI) \text{ is true} \end{array} \right\}$$

Reduction Phase. A service is considered useful in a composition for a request if its output parameters contain at least one useful instance.

An output instance I returned by a service on layer i is considered useful in composition for a request (C_{in}, C_{out}) if there is no instance in C_{in} that is equal or more particular than the instance I and at least one of the following conditions:

1. I is equal or more particular than an instance $inst \in C_{out}$ and there is no service on any next layer $k > i$ that outputs the instance *inst* or a particularization of it.
2. I is equal or more particular than an instance *inst* that is input parameter or a particularization of an input parameter of an useful service from any next layer $k, i < k$ and there is no service on a intermediate layer $j, i < j < k$ that returns as output parameter the instance *instances* or a particularization of it.

In this step, all services that are not useful in the composition for the request are removed from the composition. Also, for every service we keep as output parameters only the instances that are useful.

3.1. Implementation

In the following algorithm descriptions, we consider that the taxonomy, the repository of services and the client request are known. The algorithm is split into several methods or parts, to allow the necessary degree of relevant detail, while avoiding larger sections.

First, an important preprocessing phase is presented, that is particular to our solution and that is crucial to the improved complexity if used together with several other suitable data structures, that are more common. This optimization can often be used when working with tree data structures for querying ancestor-descendant relations in constant time.

3.1.1. *Parsing taxonomy to reduce the time complexity of* subsume *queries to* O(1)

The following algorithm 1 describes the method that computes the "*entry time*" and "*exit time*" of each node of a concept in the taxonomy, performing a linearization of the taxonomy tree. The *time* variable is a global variable initialized with 0 that will keep the position in a modified Euler tour of the tree.

Algorithm 1. Compute *entry time* and *exit time* of each concept node

```
   Input: concept
1: Method linearization(concept):
2:     $time \leftarrow time + 1$;
3:     $entryTime[concept] \leftarrow time$;
4:     foreach $subconcept \in subconcepts(concept)$ do
5:         $linearization(subconcept)$;
6:     end
7:     $time \leftarrow time + 1$;
8:     $exitTime[concept] \leftarrow time$;
```

The above algorithm builds an **Euler type tour** of the tree. It is useful based on the following key observation: node b is a successor of node a iff $entryTime[a] < entryTime[b] < exitTime[a]$, thus the reduction to $O(1)$. This method is further used within the subsumes function throughout the algorithm.

3.1.2. *Main algorithm loop.*

The following function 2 describes the main repetitive loop that alternates the two steps: *building* and *reduction* phase. The first step is performed in the $findSolution$ function, where a valid composition is constructed. The second step takes the resulting composition and analyzes it layer by layer in reverse order to determine and eliminate useless instances and services. This reduction is described in $removeUselessInstancesAndServices$ function. To speed up the process, for the next iterations only the services that appear in the composition generated by the *building* step are further stored in the repository.

Algorithm 2. Main algorithm loop

```
    Input: $taxonomy, repository, userQuery$
    Output: $solution$ // valid composition with minimal execution path
1:  $previousSol \leftarrow \emptyset$; $solution \leftarrow \emptyset$;
2:  $time \leftarrow 0$;
3:  $linearization(mostGeneralConcept)$;
4:  do
5:      $previousSol \leftarrow solution$;
6:      $solution \leftarrow findSolution()$;
7:      $repository \leftarrow \bigcup_{layer \in result} layer$;
8:      $removeUselessInstancesAndServices(solution)$;
9:  while $previousSol \neq solution$;
10: return $solution$;
```

3.1.3. *Construction phase*

$findSolution$ method 3 builds the composition layer by layer, starting with the instances C_{in} provided by the client who makes the request, named *providedInstances* in the algorithm. The construction stops when all the requested instances (from the C_{out} set, named *wantedInstances* in the algorithm) are obtained or there is no service that can be added at the composition. A service is added to a layer only if it provides as output parameters new instances that are not provided by other services in the composition or by the client. The services that can be added to the current layer are analyzed in descending order of their score values, to add fewer services to the composition. Adding a service on a layer is performed by $addServiceToLayer$ method.

Inside $addServiceToLayer$ method 4 the service received as input parameter is added to the current layer. When adding a new service to the composition it is necessary to determine which new instances are in the output param-

Algorithm 3. Construction phase

```
Input: $repository$, $providedInstances$, $wantedInstances$
Output: $result$ // valid composition
1:  Function findASolution():
2:      $auxWantInstances \leftarrow wantedInstances$; $result \leftarrow \emptyset$;
3:      $servicesReadyToCall \leftarrow \emptyset$;
4:      $ownedInstances \leftarrow \{inst \mid \exists\, provInstance \in providedInstances$ and $subsumes(inst, provInstance)\}$;
5:      $mark(ownedInstances, null)$;
6:      while $wantedInstances \neq \emptyset$ and $servicesReadyToCall \neq \emptyset$ do
7:          $layer \leftarrow \emptyset$;
8:          $servicesToCall \leftarrow servicesReadyToCall$;
9:          $servicesReadyToCall \leftarrow \emptyset$;
10:         while $servicesToCall \neq \emptyset$ do
11:             $bestService \leftarrow \max\{servicesToCall\}$;
12:             $servicesToCall \leftarrow servicesToCall \setminus \{bestService\}$;
13:             if $\neg subsumes(newOuts[bestService], ownedInstances)$ then
14:                 $addServiceToLayer(bestService, layer)$;
15:             end
16:         end
17:         if $layer \neq \emptyset$ then
18:             $result \leftarrow result.\,layer$;
19:         end
20:     end
21:     $wantedInstances \leftarrow auxWantInstances$;
22:     foreach $service \in repository$ do
23:         $neededInps\,[service] \leftarrow inps\,[service]$;
24:         $newOuts\,[service] \leftarrow outs\,[service]$;
25:     end
26:     return $result$;
```

eters of the service. Because the taxonomy can be of considerable size, it is important to determine these instances efficiently. We used a hash table to track and update the known instances. To discover new instances provided by a newly added service, all its output parameters are iterated. For each, we advance higher in the taxonomy, starting with the node corresponding to direct parent node of the instance, and stopping at the first node that has all its instances in *ownedInstances* set. There is no point to continue going up in the taxonomy after this node, because all the instances on the path between this node and the root are already in *ownedInstances* set, being added at least when we added an instance of the concept we stopped at. With this optimization, each node in the taxonomy is visited at most once per construction, resulting in linear *amortized* time complexity for performing all the updating found generalizations.

Algorithm 4. Adding a service to a layer

```
1:  Method addServiceToLayer($layer, service$):
2:      $layer \leftarrow layer \cup \{service\}$;
3:      foreach $inst \in newOuts[service]$ do
4:          $instancesObtained \leftarrow \{gInst \mid subsumes\,(gInst, inst)\,\} \setminus ownedInstances$;
5:          $ownedInstances \leftarrow ownedInstances \bigcup instancesObtained$;
6:          $mark(instancesObtained, service)$
7:      end
```

The missing input instances that are required to call a specific service are stored separately in order to determine when the service can be added to the composition. The service can only be called when the number of missing input instances reaches 0. This process is applied to each candidate service, using a mapping from input instances to the services that require them to improve performance.

After selecting the services that can be called, the service that best improves the known instances according to the dynamic-scoring function is selected to be added to the next layer of the composition. After this service is added its output parameters are then used to update the known input instances and the scores of other candidate services. This process is also optimized using a mapping of the output parameters to the services that provide them.

Algorithm 5. Method used for updating the services with input or output parameters in *instancesObtained* set

```
1:  Function mark($instancesObtained, exceptService$):
2:      foreach $service \in repository$ do
3:          if $neededInstances[service] \bigcap instancesObtained \neq \emptyset$ then
4:              $neededInstances[service] \leftarrow neededInstances[service] \setminus instancesObtained$;
5:              if $neededInstances[service] = \emptyset$ then
6:                  $servicesReadyToCall \leftarrow servicesReadyToCall \bigcup \{service\}$;
7:              end
8:          end
9:          if $service \neq exceptService$ then
10:             $newOuts[service] \leftarrow newOuts[service] \setminus instancesObtained$;
11:         end
12:     end
```

3.1.4. Reduction phase

In algorithm 6 layers are iterated from the last one to the first one to find out what is "*useless*" on each layer and to discard such services. At the beginning all instances that are required by the client, C_{out} named *wantetInstances* in the algorithm, are considered useful.
All the useful instances are stored in a hash table called *usefulInstances*. While iterating the layers, all the instances that are useful for the last iterated layer, named *usefulForLayer*, are added to *usefulInstances* set.
Another hash table, *reqInstances* is used for keeping the set of instances from iterated layers for which an output instance that is more particular than it isn't yet found. Output instance *out* is considered useful if it is a particularization of any other instance *inst* in the *reqInstances* set and there is no another instance *pr* in the set C_{in} that is more particular than *out*. If a service does not provide at least one useful instance, then it is considered "*useless*" and it will be removed from the composition. Otherwise, all its input instances that are specializations of at least one instance from the *provided* set will be added to the set *usefulForLayer*. For a useful service it is also needed to identify all the instances from *reqInstances* set that are more generic than at least one output parameter of the service and to remove those instances from the *reqInstances*. This is motivated by the fact that when adding the service to the composition we can use its output parameters as input parameters for services on following layers. After iterating all the services on the layer, the "*useless*" services are removed and the instances from the current layer proven to be useful are added to *reqInstances* and *usefulInstances* sets.

4. Evaluation

The algorithm described in Section 3 was first validated for correctness with a specially created checker. Then output parameters and execution times were compared with the results from the WSC-08 competition[2], and with a later solution published in 2011[1]. Original tests from the competition have been used for both that were generated to simulate realistic, real-world scenarios. The algorithm performance would rank first in the competition based on the same scoring rules, and runs much faster than the algorithm in[1]. The implementation was done in `Java` and ran on a `Intel(R) Core(TM) i5-3210M CPU @250 GHz` with 8 `GB RAM`.

4.1. Web Services Challenge 2008 Results

The solutions have been evaluated over 3 tests, and on each of them, one solution could gain at most 18 points. 6 points were granted if the provided solution contains the minimum number of services, 6 for minimum execution

Algorithm 6. Reduction phase

```
1: Function removeUselessInstancesAndServices($solution$):
2:     $usefulInstances \leftarrow wantedInstances$;    $toFindInstances \leftarrow wantedInstances$;
3:     $reqInstances \leftarrow reqInstances \setminus \{inst \in reqInstances \mid \exists\, pr \in providedInstances, subsumes(inst, pr)\}$;
4:     for $i \leftarrow |solution| - 1$ to 0 do
5:         $layer \leftarrow solution[i]$;    $servsToRemove \leftarrow \emptyset$;    $usefulForLayer \leftarrow \emptyset$;
6:         foreach $serv \in layer$ do
7:             $instancesToRemove \leftarrow \{inst \mid \exists\, out \in outs[serv], subsumes(inst, out)\}$;
8:             if $instancesToRemove = \emptyset$ then
9:                 $servsToRemove \leftarrow servsToRemove \cup \{serv\}$;
10:            end
11:            else
12:                $usefulForLayer \leftarrow usefulForLayer \cup \{inp \in inps[serv] | \nexists pr \in$
                     $providedInstances, subsumes(inp, pr)\}$;
13:            end
14:            $reqInstances \leftarrow reqInstances \setminus instancesToRemove$;
15:        end
16:        $solution[i] \leftarrow layer \setminus servicesToRemove$;
17:        $reqInstances \leftarrow reqInstances \cup usefulForLayer$;
18:        $usefulInstances \leftarrow usefulInstances \cup usefulForLayer$;
19:    end
20:    $instances \leftarrow usefulInstances$;
```

Table 1: Results relative to WSC 08 challenge solutions

		Tsinghua University		University of Groningen		Pennsylvania University		**Proposed algorithm 3**	
		result	points	result	points	result	points	result	points
Test 4	min. services	10	+6	10	+6	10	+6	**10**	**+6**
	min. path	5	+6	5	+6	5	+6	**5**	**+6**
	time (ms)	312	+4	219	+6	28078		**34**	**+6**
Test 5	min. services	20	+6	20	+6	20	+6	**20**	**+6**
	min. path	8	+6	10		8	+6	**8**	**+6**
	time(ms)	250	+6	14734	+4	726078		**87**	**+6**
Test 6	min. services	46		37	+6			**45**	
	min. path	7	+6	17		no result		**7**	**+6**
	time(ms)	406	+6	241672	+4			**132**	**+6**
Score		46 Points		38 Points		24 Points		**48 Points**	

path. The first, second and third fastest running times would gain another 6, 4 and respectively 2 points, under the condition that they provide valid minimum path or number of services. Eight universities submitted solutions to the challenge, and the top three results are listed together with our proposed solution in Table 1. The points for top three fastest solutions are not updated for the solutions at the competition, even if our solution had the fastest running time on all tests.

4.2. Results Relative to the Optimum Solution Search Algorithm

The algorithm proposed in[1] guarantees finding the optimum solution, with minimum number of services. It uses a A* graph-search based algorithm heuristic to reduce its running time, but does not run as fast as the solutions winning the competition. Our algorithm: is clearly faster than all other algorithms, provides solutions with close to minimum number of services (on the evaluated tests) and always generates solutions with minimum execution path. As seen in Table 2, only on test 6 the solution contains 3 extra services, out of all eight tests in the benchmark (out of which, three were used in competition evaluation).

Table 2: Results compared with shortest solution algorithm[1]

Test	Algorithm in[1]			**Proposed algorithm** [3]		
	number of services	execution path	time (ms)	number of services	execution path	time (ms)
1	10	3	91	**10**	**3**	**22**
2	5	3	123	**5**	**3**	**27**
3	40	23	1929	**40**	**23**	**76**
4	10	5	314	**10**	**5**	**34**
5	20	8	6356	**20**	**8**	**87**
6	**42**	7	777	**45**	**7**	**132**
7	20	12	9835	**20**	**12**	**95**
8	30	20	6398	**30**	**20**	**109**

5. Conclusion and Future Work

The proposed algorithm shows that at least relative to the benchmark of the 2008 challenge, there are still a lot of improvements that should be addressed for efficient Web Service Composition. Related to the **semantic** extension by use of parameters as concepts, it proves that the problem complexity is not increased if the parameters are matched by *subsumation*. This is relative to the simplest, string equality match used earlier, for example in the 2005 edition of the challenge. In practice, run times are similar with run times of the algorithm in[6] that solves compositions with simple match, if the ontology is preprocessed efficiently, for example as in Subsection 3.1.1. Furthermore, we compared with the latest published solution in[1], that had the opposite approach on `NP-Hard` optimization problems: worst-case exponential running time, with guaranteed optimal solution. Our algorithm has polynomial time complexity and approximates on the length of the generated solution, that is observed to be very close to optimal on the benchmark.

For future work we want first to add parallelism to the algorithm, speeding up the runtime even more and to adapt it to consider more aspects of service composition, like Quality of Service (QoS); dynamic service state changes, simultaneous construction of alternate compositions, etc. Also, a different but important direction we consider for future is to extend the expressibility of modeled semantics, motivated by the lack of means of modeling real-world examples such the one presented in Section 2.1.

Acknowledgements

The dissemination of this work is partly funded from the *European Union's Horizon 2020 research and innovation programme* under grant agreement No 692178. This work was also partly supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI - UEFISCDI, project number 10/2016, within PNCDI III.

References

1. Rodríguez-Mier, Pablo, Manuel Mucientes, and Manuel Lama. "Automatic web service composition with a heuristic-based search algorithm." Web Services (ICWS), 2011 IEEE International Conference on. IEEE, 2011
2. Bansal, Ajay, et al. "WSC-08: continuing the web services challenge." IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008.
3. Blake, M Brian and Tsui, Kwok Ching and Wombacher, Andreas. "The EEE-05 challenge: A new web service discovery and composition competition", In e-Technology, e-Commerce and e-Service, EEE 2005.
4. Kona, Srividya, et al. "WSC-2009: a quality of service-oriented web services challenge." Commerce and Enterprise Computing, 2009. CEC'09. IEEE Conference on. IEEE, 2009.
5. Blake, M. Brian, Thomas Weise, and Steffen Bleul. "WSC-2010: Web services composition and evaluation." Service-Oriented Computing and Applications (SOCA), 2010 IEEE International Conference on. IEEE, 2010.
6. Diac, Paul. "Engineering Polynomial-Time Solutions for Automatic Web Service Composition." Procedia Computer Science 112 (2017).
7. McIlraith, Sheila A., Tran Cao Son, and Honglei Zeng. "Semantic web services." IEEE intelligent systems 16.2 (2001): 46-53
8. Weerawarana, Sanjiva and Curbera, Francisco and Leymann, Frank and Storey, Tony and Ferguson, Donald F. "Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more". Prentice Hall PTR, 2005.
9. Lara, Rubén; Roman, Dumitru; Polleres, Axel; Fensel, Dieter. "A conceptual comparison of WSMO and OWL-S", Web services (2004).
10. Cao, Xiaoqi and Kapahnke, Patrick and Klusch, Matthias, "SPSC: Efficient composition of semantic services in unstructured P2P networks", European Semantic Web Conference (2015): 455-470;
11. Casati, Fabio and Ilnicki, Ski and Jin, Li-Jie and Krishnamoorthy, Vasudev and Shan, Ming-Chien. "eFlow: a platform for developing and managing composite e-services." Academia / Industry Working Conference on Research Challenges (AIWoRC), 2000
12. Markou, George; Refanidis, Ioannis. "MAD SWAN: A Semantic Web Service Composition System", ESWC (2013).