2011 IEEE International Conference on Web Services

Automatic web service composition with a heuristic-based search algorithm

Pablo Rodriguez-Mier*, Manuel Mucientes*, Manuel Lama*
*Centro de Investigación en Tecnologías de la Información (CITIUS)
Universidad de Santiago de Compostela, E-15782 Spain
{pablo.rodriguez.mier,manuel.mucientes,manuel.lama}@usc.es

Abstract—**Service Oriented Architectures and web service technology are becoming popular in recent years. As more web services can be used over the Internet, the need to find efficient algorithms for web services composition that can deal with large amounts of services becomes important. These algorithms must deal with different issues like performance, semantics or user restrictions. In this paper we present an A* algorithm which solves the problem of semantic input-output message structure matching for web service composition. Given a request, a service dependency graph with a subset of the original services from an external repository is dynamically generated. Then, the A* search algorithm is used to find a minimal composition that satisfies the user request. Moreover, in order to improve the performance, a set of dynamic optimization techniques has been implemented over the search process. A full experimental validation with eight different public repositories has been done showing a good performance as in all tests as the algorithm finds a valid solution with minimal number of services and execution path.**

Keywords-**Heuristic search; A* algorithm; Web services composition;**

I. INTRODUCTION

Web Services are self-contained modular applications described by a collection of operations that are network-accessible through standardized web protocols, and whose features are defined using a standard XML-based language [1], [2]. Among the characteristics of the Web Services, there are functional features that indicate the inputs and outputs required to invoke the execution of a web service; non-functional features such as cost, reliability, robustness, etc.; interaction features that describe how a client dialogs with the service in order to consume its functionality; and structural features that model how the internal components of the service are combined to execute it.

One of the advantages of the web services is to enable greater and easier integration and interoperability among systems and applications. This advantage is partly given by the ability of web services to communicate their data efficiently and effectively over the network. Hence, if this communication fails or there is no way to respond a user request with a single web service (i.e., there is no service with the required inputs and outputs), there should be the possibility to combine existing services in order to fulfill the request. This combination consists of a set of services that are executed in a sequence or in a set of workflow-like structures that control the execution of the services (specified through web services composition languages as OWL-S [3] or BPEL4WS [4]).

In the last years several papers have dealt with composition of web services. Some approaches, such as [5]–[10] treat the service composition as a planning problem, where a sequence of actions lead from a initial state (inputs and preconditions) to a goal state (required outputs). However, most of these proposals have some drawbacks: high complexity, high computational cost and inability to maximize the parallel execution of web services.

Other approaches, such as [11]–[17], consider the problem as a graph/tree search problem, where a search algorithm is applied over a web service dependency graph in order to find a minimal composition. These proposals are simpler than the AI planners and also many of them can exploit parallel execution of web services. However, most of these approaches rely on very complex dependency graphs that have not been optimized to reduce data redundancy (equivalent services and equivalent combination of services). Therefore, the scalability of these algorithms may also be adversely affected when the interaction among services and data is huge due to the redundancy of the repository.

This paper addresses the problem of the web service composition as a graph search problem without consideration of non-functional properties. The novelties of our proposal are:

1) We describe some optimization techniques to reduce the graph size by eliminating redundancy.
2) We present a heuristic search algorithm based on the well-known A* which finds an optimal composition with a minimal number of services and execution path (i.e., maximizes the parallel execution of services).
3) We define a method to reduce dynamically the possible paths to explore during the search by filtering equivalent compositions.

Furthermore, a full validation has been done in eight public repositories proposed for the 2008 Web Service Challenge of the IEEE conference [18]. The behavior of the algorithm shows a great performance, as in all the cases the best composition was found.

The rest of the paper is organized as follows: Section II describes the different approaches that have already been proposed. Section III introduces the basis of web service composition. Section IV illustrates the proposed A* algorithm for web service composition. Section V presents some optimization techniques to improve the performance of the algorithm. Section VI analyzes the algorithm with eight different repositories. Section VII concludes the paper.

II. Related Work

The use of graph-based and tree-based search algorithms to solve the composition problem has been studied before [16]. Although there are similarities among all proposals, they differ in many concepts, such as performance, information handling, graph/tree encoding, solution quality, etc. In this section, a brief analysis of some approaches is presented.

Shiaa et al. [11] present an approach to automatic service composition with semantic matching. Given a request (goals, inputs and outputs), a set of matching services are discovered from the repository, applying semantic matching between service properties and the composition request. Then, a graph is created dynamically by connecting semantically similar nodes (single services) to each other. Once the graph is created, a search over it is performed building acyclic tree structures from goal nodes to start nodes. One major drawback of this proposal is that it does not take into account the use of heuristics in order to speedup the search, so searching for an optimal composition in large repositories may be infeasible. Moreover, there are no experimental results to validate the model.

Kona et al. [19] propose a simple but effective approach for semantic web service composition. In this work, a composition is generated as a directed acyclic graph from a user request. The graph (divided in a set of layers) is calculated iteratively, starting with the input parameters provided by the requester. In each step, all possible services from the repository that can be invoked are added to the current layer. Although the useless services are filtered, the algorithm cannot find an optimal composition. A heuristic search over the graph is required in order to minimize the number of services in the composition.

Yan et al. [15] present an automatic service composition algorithm using AND/OR graph. In this proposal, an AND/OR graph is created from a request, connecting services by their inputs and outputs. Then, a search over the graph is performed using the AO* search algorithm. Although this proposal shows a great performance over large repositories, the autors have not implemented optimization techniques in order to improve the scalability of the algorithm.

Oh et al. [17] propose a Web-Service Planner using the A* search algorithm (WSPR*), an improvement of the WSPR planner. In this approach, the use of the A* algorithm allows finding an optimal composition based on some heuristic costs. The heuristic function is defined as the set of required parameters found by the algorithm. This heuristic function has an important drawback: it is not able to guide the search when only the last services of a composition produce all the required parameters. On the other hand, the transition function only allows the addition of a single service in each step. In contrast, our approach uses an optimistic heuristic based on the distance from the current composition to the initial node. Furthermore, the search is performed backwards, handling more than a single web service in each step, which allows to exploit parallelism during the search.

We can conclude that the main differences between our proposal and other approaches are:

- The construction of a non-redundant service dependency graph at the first stage by removing unused services and combining the equivalent ones. Other approaches use simple filtering techniques that do not remove all data redundancy.
- The use of the A* algorithm backwards, handling multiple services in each step in order to maximize the execution in parallel of the web services.
- The use of dynamic optimization during the search, that reduces the number of possible paths to explore by combining equivalent combination of services.

In the following sections we describe in detail the composition problem and how can it be solved with our proposal.

III. Web services composition

In order to compose web services, we must define the relationship among services. From a simplistic point of view, a web service is a software component that receives a set of inputs and generates a set of outputs after the execution. Thus, a web service w can be described by a set of inputs $W_{in} = \{I_1, I_2, ...\}$ and a set of outputs $W_{out} = \{O_1, O_2, ...\}$. Outputs from a service can be provided as inputs to other service only if there is a semantic relationship between them. In our approach, we have modeled this restriction as a hierarchical class/subclass relationship between concepts, so we consider that an output of a service O_{so} matches the input of other service I_{si} when O_{so} is a subclass of I_{si}. In general, when a concept C_i is a subclass of a concept C_j ($C_i \subseteq C_j$), then there is a semantic matching between C_i and C_j.

Another important concept is a web service request. A request R is composed by a set of inputs ($R_{in} = \{I^1_{in}, I^2_{in}, ...\}$) provided by the requester, and a set of outputs ($R_{out} = \{O^1_{out}, O^2_{out}, ...\}$) that the requester expects to obtain. Given a request $R_{user} = \{R_{in}, R_{out}\}$, where $R_{in} = \{I^1_R, I^2_R, ...\}$ and $R_{out} = \{O^1_R, O^2_R, ...\}$, and given a web service $S = \{S_{in}, S_{out}\}$ where $S_{in} = \{I^1_S, I^2_S, ...\}$ and $S_{out} = \{O^1_S, O^2_S, ...\}$, the web service S can be invoked only if $R_{in} \supseteq S_{in}$, i.e., for each input $I_S \in S_{in}$ there exists an input $I_R \in R_{in}$ such that I_R is equal or subclass of I_S ($I_R \subseteq I_S$). Also, R_{out} will be satisfied only if $R_{out} \subseteq S_{out}$, i.e., for each output $O_R \in R_{out}$ there exists an output $O_S \in S_{out}$ such that O_S is equal or subclass of O_R ($O_S \subseteq O_R$).

Considering this description for web services, the composition problem can be formulated as the automatic construc-

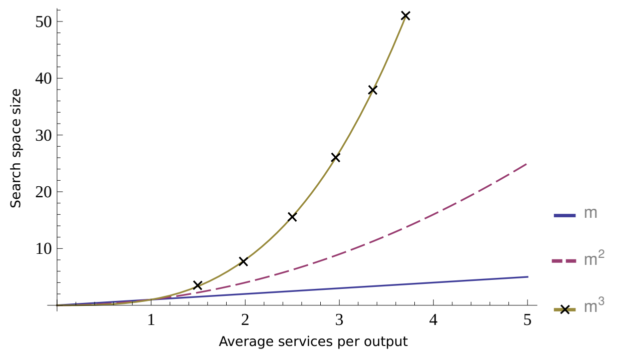

Figure 1. Search space size for n=1, n=2 and n=3 (1, 2 and 3 inputs per service) in the first expansion (d=1).

tion of a workflow that coordinates the execution of a set of services that interact among them through their inputs and outputs (applying the semantic matching). This workflow, therefore, has services and a set of control structures that define both the behaviour of the execution flow and the inputs/outputs of the services related to those structures. Despite the amount of different control structures defined in composition languages like BPEL4WS, we take into account only two of the most important ones: sequence and split. These structures allow to build most of the possible compositions and they work as follows:

- Sequence structure: the output of a service is the input of one of the following services of the sequence. This is the basic control structure of the workflow languages.
- Parallel (split): two or more services are executed in parallel and, as result, produce several and different ouputs.

Regarding to the complexity analysis of the search space, the number of combinations to be analyzed using a brute-force algorithm grows very fast. To demonstrate this, we can assume that, given a service, each of its inputs is provided by a different service (worst case). The complexity in this scenario is $O(m^{nd})$, where m is the average number of services in the repository that generate the same output, n is the average number of inputs from web services and d is the depth at which all inputs are resolved. Since there are m services that provide each required input, the number of possible choices in order to resolve all inputs from a service is m^n. Each of these combinations represent a set of services executed in parallel, that can be expanded again. Fig. 1 shows the size of the search space for $d = 1$, with $n = 2$ and $n = 3$ (two and three inputs respectively for each service in repository).

As can be seen, this kind of compositions have an exponential growth of paths to explore. The search space size in the case of a repository of services with three inputs on average (n) and four possible choices to provide an input to a service (m), where the solution has a runpath of 10 (i.e, d=10 splits connected in sequence), reaches the value of $4^{3\cdot 10} = 1.1529 \times 10^{18}$ possible paths to explore. Given the large number of combinations, the problem of searching an optimal execution path is not trivial, and it is therefore necessary to reduce the number of combinations. In order to reduce the search space size, our algorithm includes some optimization techniques, which are described in Sec. V.

IV. A* ALGORITHM FOR WEB SERVICES COMPOSITION

As previously discussed, given the large number of possible paths to explore, a fast algorithm is required in order to find an optimal solution in a reasonable period of time. Although the high space complexity makes the use of traditional search algorithms unpractical for large repositories, the problem can be solved by using a good heuristic in the search and applying some optimization techniques and data preprocessing.

The A* Algorithm, developed by Hart et al. [20], is one of the most popular pathfinding algorithms. This algorithm uses a heuristic function $h(n)$ to estimate the cost from the current node to a goal node, and a function $g(n)$ to calculate the cost from the starting node to the current node. Therefore, the search cost is defined as $f(n) = g(n)+h(n)$. Choosing a good h function has an important impact on the search process. The better this function is, the faster the solution will become. However, there is a restriction on it: h cannot overestimate the cost to reach the goal, otherwise, the algorithm could find a solution with higher cost than the optimal one.

Our proposal, based on A* algorithm, follows the next steps: first, a web service dependency graph is computed, based on the method described in [19]. Then, a reduction on the number of services is performed by eliminating unused services and combining equivalent services. Finally, the A* search is applied over the reduced graph, which finds an optimal service composition, with minimal number of services and execution path. These steps will be described in the following sections.

A. Web service dependency graph

Web services composition requires the combination of many atomic services that can be executed in sequence or in parallel as previously mentioned. Given a service request, a service dependency graph with a subset of the original services from an external repository is dynamically generated. This subset contains the solutions that meet the request and consists of a set of layered services (splits) connected in sequence. Each layer contains all services from the repository that can be executed with the outputs of the previous one. The general expression for a layer can be defined as follows:

$$L_i = \{S_i : S_i \notin L_j(j < i) \wedge I_{Si} \cap O_{i-1} \neq \emptyset \wedge I_{Si} \subseteq I_R \cup O_0 \cup \ldots \cup O_{i-1}\}$$

where, for each layer L_i:

- S_i is a service on the i_{th} layer.
- O_i is the set of outputs generated in the i_{th} layer.
- I_{Si} is the set of inputs required for the execution of service S_i.
- I_R is the set of inputs provided by the requester.

The construction of the graph can be done in a simple manner, as can be seen in Alg. 1. First, a subset of services from the repository is selected. This subset consists of all the services such that all their inputs are provided by the requester. Then, the new outputs generated by the selected subset are combined with the inputs provided by the requester. This combination defines a new set, I_a, which will be the available inputs for the next layer. These steps are repeated iteratively until the outputs of all layers contain the requested output set.

Algorithm 1 Service dependency graph algorithm

```
1: $I_a := \emptyset$
2: $i := 0$
3: $O_{total} := I_R$
4: $Layers := \emptyset$
5: repeat
6:   $L_i := \emptyset$
7:   $I_a := I_a \cup O_{total}$
8:   for Service $S_i \in$ Repository do
9:     $O_s := Outputs\{S_i\}$
10:    if $S_i \in L_i$ then
11:      $Li := Li \cup S_i$
12:      $O_{total} := O_{total} \cup O_s$
13:    end if
14:  end for
15:  $Layers := Layers \cup L_i$
16:  $i = i + 1$
17: until $O_R \subseteq O_{total} \vee L_i = \emptyset$
```

In order to speed up the calculation of the graph, we used a pre-computed table that maps each input to the services that use it. Thus, for each output generated in a layer, we can obtain all possible services for the next layer very quickly. In Figure 2, an example of a service dependency graph with i layers is showed. Web services denoted as R_i and R_o are dummy services. The outputs of R_i are the inputs provided by the requester. Since these outputs are provided in the first layer, they will be propagated through all layers from 1 to i, so the layer L_1 will have all services from the repository such that all their inputs are a subset of R_i. Moreover, the inputs of R_o are the outputs wanted by the requester.

B. A* algorithm description

Once the graph is calculated, a search over it must be performed. The search algorithm will traverse the graph backwards, from the solution (the service whose inputs are the outputs wanted by the requester), to the initial node (the service whose outputs are the provided inputs). As mentioned before, our heuristic algorithm is based on an implementation of the A* pathfinder. There are three principal concepts in this type of algorithms: the neighborhood function, the cost function and the heuristic function. These concepts will be explained below.

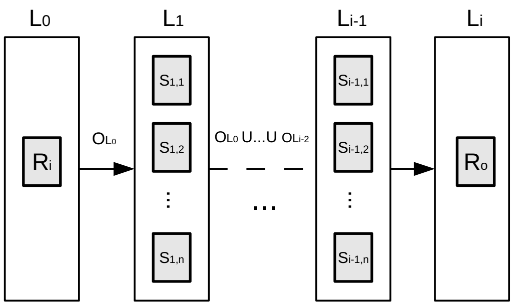

Figure 2. Example of i layers, with n services per layer

In order to perform the searching process, the search space must be divided into nodes. Each node will contain a set of services from a graph layer that can be executed in parallel. Thus, a path will be composed of a list of neighbor nodes, which represents the sequential execution path. Thus, the starting node will only contain the service labeled as R_0 in Fig. 2. This service represents the outputs wanted by the requester, as their inputs match with them. To generate all possible neighbors from a node, the following steps are performed:

1) Calculate, for each input of a node, a list of services from the previous layer that provide it. If there are no services in the previous layer for that input, a dummy service that generates this input and receives the same input is created. This dummy holds the dependency so it can be resolved later.
2) Make all combinations between services from each list. These combinations will generate all possible neighbors from the current node.
3) Remove all equivalent neighbors. This process will be described in Sec. V.

For example: Given a node N with a service S in the layer L_i, with $I_s = \{a, b\}$ and a set of services X, Y, Z in the layer L_{i-1} where $O_x = \{a\}$, $O_y = \{b\}$ and $O_z = \{a, b\}$, we construct a list of services for each input of S:

- $Set(a) = \{X, Z\}$
- $Set(b) = \{Y, Z\}$

Then, we generate all combinations. Each combination will constitute a neighbor node from N. The possible combinations are: (X,Y), (X,Z), (Y,Z), (Z). All these nodes generate all the required inputs for node N (a, b).

On the other hand, the behaviour of the A* algorithm depends on two functions: $g(N)$, the cost, and $h(N)$, the

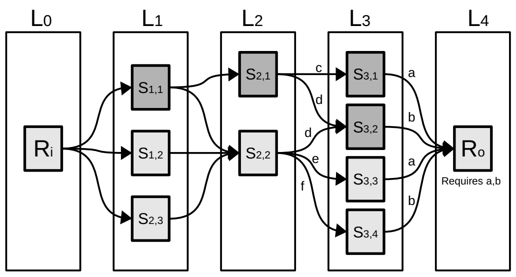

Figure 3. Example of the minimum composition over the graph. Dark grey services correspond with the services selected by the algorithm. This composition can be written as: $sequence(S_{1,1}, S_{2,1}, split(S_{3,1}, S_{3,2}))$

heuristic. N is a composite service obtained as a path over a set of nodes (N_i), where N_i is the set of services in layer L_i. One of the goals is to minimize the number of web services in a composition, therefore, the function cost should calculate the lenght of a composition based on the number of services. On this basis, we define a function $g(N)$ as showed in 1 :

$$g(N) = \sum_{i=L_N}^{\#L} cost(N_i) \tag{1}$$

where L_N is the first layer of the current composition service, $\#L$ is the first layer and $cost$ is a function that retrieves the number of services from node N_i. The dummy services in a node will not be computed.

The other function is the heuristic. This function should estimate the cost to the solution. A good choice is to use, as heuristic, the layer in which the node is located. The layer number indicates the distance from the initial node. Thus, a service in layer 3 means that the algorithm needs three more steps in order to reach the start node. The heuristic function is defined as:

$$h(N) = distance(N_i) \tag{2}$$

Putting (1) and (2) together, function $f(n)$ is defined as (3):

$$f(N) = \sum cost(N_i) + distance(N_i) \tag{3}$$

Figure 3 shows an example of a minimum composition path detected with this algorithm. In the next section, a set of optimization techniques are explained.

V. Optimization techniques

In order to achieve a significant performance improvement over the search process, we designed two techniques that reduce the number of possible paths to explore: *Offline Service Compression* and *Online Node Reduction*.

Table I
Characteristics of the Web Service Challenge repositories.

Test	#S.	#I.	#O.	I./S.	O./S.	S./I.
WSC'01	158	735	778	3.53	5.25	2.69
WSC'02	558	2972	2890	3.79	3.92	8.97
WSC'03	604	3254	3129	4.07	6.46	7.01
WSC'04	1041	5781	5611	4.23	5.47	16.10
WSC'05	1090	5816	5953	3.36	4.26	9.24
WSC'06	2198	12218	11831	6.00	4.31	16.42
WSC'07	4113	22324	22392	7.37	7.21	41.14
WSC'08	8119	44569	44628	5.44	6.54	20.11

A. *Offline Service Compression*

The essence of this technique is to combine equivalent services from each layer in the graph, which implies a lower number of paths to explore during the search. This process is subdivided into two steps: remove unused services and detect equivalent services. These steps are described below:

- Remove unused services:
  1) Create an empty list M. This list will contain all the required inputs to reach the solution.
  2) Create an empty list U. This list will contains all unused services.
  3) Traverse backwards the graph, starting from the final layer.
  4) For each layer L in the graph:
     a) Create an empty list R. This list will contain all the required inputs for this layer.
     b) For each service S in the current layer:
        i) Check if $O_s \subseteq M$, where O_s are the service outputs. If M is empty, skip this step.
        ii) If S meets the condition or M is empty, add all inputs from S to the list R.
        iii) In other case, add S to the list U
     c) Add all inputs from R to the list M.
  5) Finally, remove from the graph, each service in U
- Detect and combine equivalent services:
  1) For each layer in the graph:
     a) Group services by the equivalence of their inputs. Two services have equivalent inputs if the services from the graph that provide their inputs are the same.
     b) For each group:
        i) If the services have quality parameters, then check the dominance between them.
        ii) For each service S_i that dominates other S_j from the group ($S_i \succeq S_j$), check if $O_{S_i} \supseteq O_{S_j} \wedge O_{S_i} \succeq O_{S_j}$. This condition is described below.
        iii) If S_i meets the previous restrictions, then combine S_j with S_i. S_j must be deleted.

One service S_i with parameters $P_{S_i} = \{P^1_{S_i}, P^2_{S_i}, ..., P^n_{S_i}\}$ dominates other service S_j ($S_i \succeq S_j$) with parameters $P_{S_j} = \{P^1_{S_j}, P^2_{S_j}, ..., P^n_{S_j}\}$ if:

$$\forall\, k \in \{1, \ldots, n\}\ P^k_{S_i} \geq P^k_{S_j} \wedge \exists\, k \in \{1, \ldots, n\}, P^k_{S_i} > P^k_{S_j}$$

Moreover, in order to check the restriction $O_{S_i} \succeq O_{S_j}$, the following steps must be performed:

1) Set $List_i$ as the list of services from the graph such that their inputs are a subset of O_{S_i}.
2) Set $List_j$ as the list of services from the graph such that their inputs are a subset of O_{S_j}.
3) Compare both lists. If $List_i \supseteq List_j$ then go to the next step. Else, the restriction is not met and therefore S_i and S_j cannot be combined.
4) Check if O_{S_i} resolves the same or more inputs from each common service than O_{S_j}. For example, if $O_{S_i} = \{a, b\}$ and $O_{S_j} = \{a, c\}$, and $List_i = List_j = X(a, b, c), Y(a, c)$, where $X(a, b, c)$ and $Y(a, c)$ are services that receive as inputs (a, b, c) and (a, c) respectively, we must verify which inputs are resolved with O_{S_i} and O_{S_j}. So, in this example, O_{S_i} resolves input a, b from X and a from Y, and O_{S_j} resolves a from X and a, c from Y. Therefore, $O_{S_i} \not\succeq O_{S_j}$.

B. Online Node Reduction

This technique consists in the combination of equivalent neighbors during the A* search process. Given that a nodes can generate equivalent neighbors (different combination of services that together are equivalent), a mechanism to delete this type of redundancy must be implemented. Two nodes are equivalent if they meet two conditions:

1) Neighbors from the node must have the same $f(n)$ value for all their objectives.
2) Services from graph that provide the inputs required for each neighbor must be the same.

The first condition is obvious: two neighbors cannot be reduced if the $f(n)$ value is different, as they will generate different paths to the solution. The second condition refers to the equivalence of the inputs. As before, a list of services that provides the required input for each neighbor must be calculated and then compared. Only nodes with same lists of services and $f(n)$ value can be combined. This technique is performed while the neighbors are being generated.

VI. Experiments

Our analysis consists in two parts: (1) we validate the algorithm with eight different repositories from Web Service Challenge 2008 and (2) we measure the speed up obtained with the optimization techniques.

A. Web Service Challenge 2008 Datasets

In order to prove the validity and efficiency of our algorithm in different situations, we carried out some experiments[1] using eight public repositories from Web Service Challenge 2008[2]. These repositories contain from 158 to 8119 services defined using WSDL. Also, inputs and outputs are semantically described in a XML file. The best solutions for each dataset are showed in Table II.

Table I shows in detail the characteristics of each dataset. Column *I./S.* indicates the average number of inputs per service, *O./S.* indicates the average number of outputs per service, and *S./I.* indicates the average number of services that provides each input. The first three columns indicate the number of services in the repository (#S), the total inputs (#I.) and the total outputs (#O). As can be seen, the complexity of the selected datasets is enough for a complete validation of our proposal.

Table II
Web Service Challenge: Solutions provided by the WSC'08

Test	min services	min exec. path
WSC'01	10	3
WSC'02	5	3
WSC'03	40	23
WSC'04	10	5
WSC'05	20	8
WSC'06	40	9
WSC'07	20	12
WSC'08	30	20

Table III
Algorithm results for the eight datasets

Test	Gr.s.	iter.	time(ms)	#serv.	ex.path
WSC'01	17	37	91	10	3
WSC'02	19	29	123	5	3
WSC'03	60	856	1929	40	23
WSC'04	31	18	314	10	5
WSC'05	62	1823	6356	20	8
WSC'06	95	13	777	42	7
WSC'07	89	332	9835	20	12
WSC'08	78	198	6398	30	20

B. Results

Our algorithm was implemented using Java™ JDK 1.6 and tested with Java™ SE build 1.6.0_22-b04 64-bit. All the experiments were performed under an Ubuntu 64-bit server workstation (kernel 2.6.32-27) with 2.93GHz Intel® Xeon® X5670 and 16GB RAM DDR-3. Table III shows experimental results obtained for each dataset. The first column indicates the dataset name. The second column

[1]An online application is available to test our algorithm with the same datasets used in this experiments: http://citius.usc.es/wiki/inv:composit

[2]http://cec2008.cs.georgetown.edu/wsc08/downloads/ChallengeResults.rar

indicates the number of services in the service dependency graph. This value is a more useful complexity indicator than the services of a repository, as the graph contains a subset of services where the solution is, and therefore represents the real search space for a request. The third column indicates the iteration[3] where the algorithm found a solution. In the fourth column we show the elapsed time until a solution was found (including the time spent in the generation of the graph). The fifth indicates the number of solutions with minimal execution path obtained by the algorithm and the last one shows the length of that execution path of the solution.

The first thing that must be noticed is that the solutions obtained by our algorithm are the best for all datasets[4] (comparing with Table II). Also, the performance is very good, as in all the tests the best solution was found in a very short period of time. As can be seen, the algorithm can find a composition of 10 services in less than 100 ms and a composition of 40 services in only 1929 ms. In general, the worst performance was obtained in test 5, 7 and 8 due to the high complexity of the repositories.

C. Optimization effect

All the above experiments were performed using all optimization techniques described on Section V. In this section, we compare the effect of the optimization over the global performance on each dataset, and its divided into three parts: (1) performance using *static service compression*; (2) performance using *dynamic node reduction* and (3) performance improvement with both optimizations.

1) Static service compression: The results are presented in Table IV. As can be seen, the average compression obtained over the graph was close to 40%. Despite the reduction obtained over the graph size, the complexity of the repository 3 still remain too high, so the algorithm cannot find a solution in a reasonable period of time, partly given by the complexity of the solution, composed by a set of 40 services with an execution path of 23.

Table IV
OPTIMIZATION RESULTS USING STATIC SERVICE COMPRESSION

Test	% compression	time (ms)
WSC'01	54	99
WSC'02	48	145
WSC'03	43	–
WSC'04	32	376
WSC'05	37	311377
WSC'06	35	607963
WSC'07	29	16789
WSC'08	36	6837

[3]An iteration in the context of the A* algorithm is defined as an expansion of the neighbors of the current path

[4]In the case of the dataset WSC'06, our algorithm found a solution with shorter runpath than the results provided by the challenge (and with fewer services than the WSC-2008 winners)

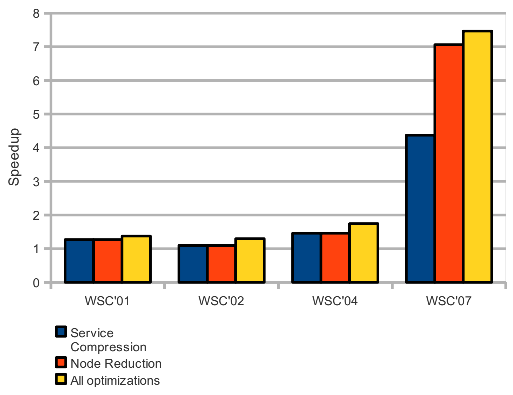

Figure 4. Speedup with different optimizations

2) Dynamic node reduction: This technique reports a big improvement in performance, as the algorithm obtains solutions in all repositories except in WSC 2008-5 due to the large number of services in the graph without compression. Thus, this technique must be combined with the service compression in order to reduce the complexity even further.

Table V
OPTIMIZATION RESULTS USING DYNAMIC NODE REDUCTION

Test name	time (ms)
WSC'01	126
WSC'02	147
WSC'03	7457
WSC'04	1002
WSC'05	–
WSC'06	5562
WSC'07	10396
WSC'08	7318

3) Both optimizations: After applying both techniques, our algorithm is able to solve the eight datasets showing a good performance, as can be seen in Table III. In Figure 4, we compare the speedup[5] obtained with each optimization over the non-optimized algorithm. Given that single techniques cannot solve all datasets, we compare only the performance with some datasets (1, 2, 4 and 7). Note that with all optimizations, the speedup is over 1.0x, i.e., there is a substantial performance improvement.

VII. CONCLUSION

In this paper we have presented a heuristic-based search algorithm for automatic web service composition over an optimized graph. The graph has been is optimized applying different techniques that reduce useless and equivalent services.

[5]The speedup is calculated as the division of the optimized result by the non-optimized result. Thus, a speedup of 2.0x indicates that the optimized result is two times faster than the non-optimized one.

The proposed A*-based composition algorithm is executed over the reduced graph using dynamic node reduction and a cost function, which minimizes the number of services and maximizes the parallelization. Moreover, a full validation has been done using eight different repositories from Web Service Challenge 2008, showing a good performance as in all the tests the best solution was found. Also, the search times obtained for each composition are quite low, allowing to use our proposal on-line.

ACKNOWLEDGMENT

This work was supported in part by the Dirección Xeral de I+D of the Xunta de Galicia under grant 09SIN065E and the Spanish Ministry of Science and Innovation under grant TSI2007-65677-C02-02. Manuel Mucientes is supported by the Ramón y Cajal program of the Spanish Ministry of Science and Innovation.

REFERENCES

[1] M. P. Papazoglou and D. Georgakopoulos, "Service-oriented computing," *Communications of the ACM*, vol. 46, no. 10, pp. 25–28, October 2003.

[2] A. Gustavo, F. Casati, H. Kuno, and V. Machiraju, *Web services: concepts, architectures and applications*. Springer Verlag, September 2003.

[3] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne *et al.*, "OWL-S: Semantic markup for web services," *W3C Member Submission*, vol. 22, pp. 2007–04, 2004.

[4] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte *et al.*, "Business process execution language for web services, version 1.1," *Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation*, 2003.

[5] M. Carman, L. Serafini, and P. Traverso, "Web service composition as planning," in *ICAPS 2003 Workshop on Planning for Web Services*, Trento, Italy, July 2003.

[6] J. Hoffmann, P. Bertoli, and M. Pistore, "Web Service Composition as Planning, Revisited: In Between Background Theories and Initial State Uncertainty," in *Proceedings of the 22nd National Conference of the American Association for Artificial Intelligence (AAAI'07)*. Vancouver, Canada: AAAI Press, July 2007, pp. 1013–1018.

[7] M. Klusch, A. Gerber, and M. Schmidt, "Semantic web service composition planning with owls-xplan," in *Proceedings of the AAAI Fall Symposium on Semantic Web and Agents, Arlington VA, USA, AAAI Press*, 2005.

[8] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso, "Automated composition of web services by planning at the knowledge level," in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005)*, L. P. Kaelbling and A. Saffiotti, Eds. Edinburgh, Scotland, UK: AAAI Press, July 2005, pp. 1252–1259.

[9] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN planning for web service composition using SHOP2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 4, pp. 377–396, 2004.

[10] K. Chen, J. Xu, and S. Reiff-Marganiec, "Markov-htn planning approach to enhance flexibility of automatic web service composition," in *IEEE International Conference on Web Services (ICWS 2009)*. Los Angeles, CA, USA: IEEE, July 2009, pp. 9–16.

[11] M. Shiaa, J. Fladmark, and B. Thiell, "An Incremental Graph-based Approach to Automatic Service Composition," in *2008 IEEE International Conference on Services Computing (SCC 2008)*, vol. 1. Honolulu, Hawaii, USA: IEEE, July 2008, pp. 397–404.

[12] P. Hennig and W. Balke, "Highly Scalable Web Service Composition Using Binary Tree-Based Parallelization," in *2010 IEEE International Conference on Web Services (ICWS 2010)*. IEEE, 2010, pp. 123–130.

[13] S. Hashemian and F. Mavaddat, "A graph-based framework for composition of stateless web services," in *Fourth IEEE European Conference on Web Services (ECOWS 2006)*. Zürich, Switzerland: IEEE, 2006, pp. 75–86.

[14] W. Jiang, C. Zhang, Z. Huang, M. Chen, S. Hu, and Z. Liu, "QSynth: A Tool for QoS-aware Automatic Service Composition," in *2010 IEEE International Conference on Web Services (ICWS 2010)*. Miami, Florida, USA: IEEE, July 2010, pp. 42–49.

[15] Y. Yan, B. Xu, and Z. Gu, "Automatic service composition using and/or graph," in *10th IEEE International Conference on E-Commerce Technology (CEC 2008) / 5th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (EEE 2008)*. Washington, DC, USA: IEEE, 2009, pp. 335–338.

[16] N. Milanovic and M. Malek, "Search strategies for automatic web service composition," *International Journal of Web Services Research*, vol. 3, no. 2, pp. 1–32, 2006.

[17] S. Oh, J. Lee, S. Cheong, S. Lim, M. Kim, S. Lee, J. Park, S. Noh, and M. Sohn, "WSPR*: Web-Service Planner Augmented with A* Algorithm," in *2009 IEEE Conference on Commerce and Enterprise Computing*. IEEE, 2009, pp. 515–518.

[18] A. Bansal, M. Blake, S. Kona, S. Bleul, T. Weise, and M. Jaeger, "WSC-08: Continuing the Web Services Challenge," in *10th IEEE International Conference on E-Commerce Technology (CEC 2008) / 5th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (EEE 2008)*. Washington, DC, USA: IEEE, 2009, pp. 351–354.

[19] S. Kona, A. Bansal, M. Blake, and G. Gupta, "Generalized semantics-based service composition," in *2008 IEEE International Conference on Web Services (ICWS 2008)*. Beijing, China: IEEE, September 2008, pp. 219–227.

[20] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.