An optimal and fast algorithm for web service composition

Pablo Rodriguez-Mier, Manuel Mucientes, and Manuel Lama

Centro de Investigación en Tecnologías de la Información (CITIUS)
Universidad de Santiago de Compostela, Spain
{pablo.rodriguez.mier,manuel.mucientes,manuel.lama}@usc.es

Abstract. Automatic composition is not a trivial problem, especially when the number of services is high and there are different control structures to handle the execution flow. In this paper, we present an A* algorithm for automatic service composition that obtains all valid compositions from a extended service dependency graph, using different control structures (sequence, split, choice). A full experimental validation with eight different public repositories has been done, showing a great performance as the algorithm found all valid compositions in a short period of time.

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

1 Introduction

Service-oriented computing plays an important role in the development of many areas, such as the improvement of business processes, internet marketing or social networks, and this trend is expected to continue in the coming years as more resources become available. Web services are software components that define formally machine readable interfaces for accessing data through the network. This feature allows to enable greater and easier integration and interoperability among systems and applications.

As a result, in the last year several papers have dealt with composition of web services, which consists in the automatic combination of services in order to combine the functionality of different modules. Some approaches, such as [2, 1], treat the composition problem as a planning problem. In general, these approaches have important drawbacks: high complexity, high computational cost and inability to maximize parallel execution of web services.

Other approaches [4, 6, 3], consider the problem as a search problem, where a search algorithm is applied over a graph or a tree in order to find a minimal composition. These proposals are simpler and more effective than the AI Planners, and also many of them can exploit parallel execution of web services.

This paper is an extension of a previous work [5] that presents a heuristic-based search algorithm to address the problem of the automatic web service composition, which the main contributions are: (1) A optimization techniques to optimize the graph size; (2) A heuristic search based on the A* algorithm that finds an optimal[1] composition over a service dependency graph and (3) a method to reduce dynamically the possible paths to explore during the search by filtering equivalent compositions. The novelties of our proposal in relation to [5] are: (1) We extended the definition of the service dependency graph to find all solutions with minimal number of services and minimal runpath and (2) we included the use of the *choice* control to support the selection of different alternatives, where more than one service has equivalent services. The rest of the paper is organized as follows: Section 2 explains the service composition algorithm. Section 3 analyzes the algorithm with eight different repositories. Section 4 concludes the paper.

2 Algorithm for web service composition

A web service can be described as a software component that produces a set of outputs given a mandatory set of inputs. Given a request, the composition problem consists in finding a set of services which once executed, produces the desired outputs.

A web service composition, then, can be formulated as a set of web services whose execution is coordinated by a workflow-like structure. This workflow, therefore, has services and a set of control structures that define both the behavior of the execution flow and the inputs/outputs related to those structures. The automatic web service composition algorithm presented in this paper can handle three different control structures: sequence, split and choice.

The present paper is an extension of a previous work [5], in which a heuristic-based algorithm to obtain only the best compositions was presented. Unlike the previous work, the algorithm presented herein is aimed at obtaining all optimal web service compositions in a very short period of time. Our proposal, based on A* algorithm, follows the next steps: 1) Compute a service dependency graph with a subset of services from repository that solves a request and 2) apply the A* search over the generated graph to obtain all valid solutions.

Given a request, an extended service dependency graph (SDG) is dynamically generated. This graph contains a subset of services from the repository that generates the wanted outputs, and it is organized in layers (splits) connected in sequence. Each layer contains one or more services in parallel that can be executed with the outputs generated by the previous layers. The 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 and I_R is the set of inputs provided by the requester.

The Alg. 1 explains with pseudocode the construction of the graph iteratively. Lines 1-4 initialize the variables used throughout the algorithm. Note that *newOutputs* (outputs generated in the last layer that have not been generated previously) and I_a (available inputs for the current layer) are initialized

[1] A composition with minimal number of services and execution path (runpath).

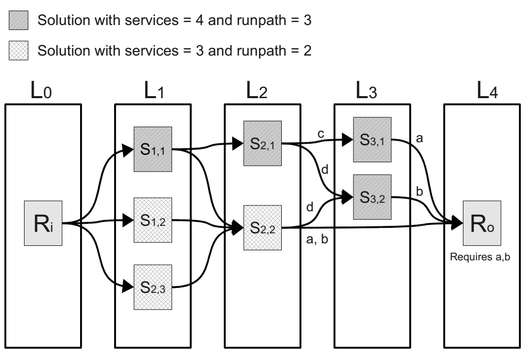
Solution with services = 4 and runpath = 3
Solution with services = 3 and runpath = 2
L0
L1
L2
L3
L4
Ri
S1,1
S1,2
S2,3
S2,1
S2,2
S3,1
S3,2
Ro
Requires a,b
c
d
d
a, b
a
b

Fig. 1. Example of two solutions with different runpath and different number of services: $sequence(S_{1,1}, S_{2,1}, split(S_{3,1}, S_{3,2}))$ and $sequence(split(S_{1,2}, S_{2,3}), S_{2,2})$

with the same value I_R, as the provided inputs are the first available inputs to the composition and have not been used yet by any service. The main loop starts at line 5. Inside this loop, each layer is calculated following these steps:

1. Obtain all outputs from the previous layers. This outputs are the available inputs to the current layer (L. 7-9).
2. For each service in the repository:
   (a) Check if the service has not appeared in previous layers (L. 12).
   (b) Check if the service can be invoked (i.e. Receives all their inputs from previous layers) (L.13).
   (c) Check if the services uses at least one output that has not been used previously (L. 14).
   (d) If (a), (b) and (c) are true, then the service is added to the current layer.
3. Once all services are selected for the *ith* layer, *newOutputs* is updated by adding the outputs of the *ith* layer and deleting the outputs generated in previous layers. Note that with this operation, only the outputs that have not been used before will remain for the next iteration (L. 19).

With the generation of the graph, the problem of handling *splits* and *sequences* is solved, as the services in the same layer can be invoked in parallel and the connection between layers implies a sequence connection. In order to introduce the *choice* control, we use the advantages of the optimization technique called "Offline service compression" defined in [5]. This technique allows to reduce the dimensions of the SDG by detecting equivalents services from each layer. Using this feature, we can replace all equivalent services by a virtual service. This virtual service holds a reference to each equivalent service, allowing the algorithm to explore less number of solutions. Once the A* algorithm has extracted all solutions, virtual services are replaced by a *choice* control with all equivalent services.

Fig. 1 shows an example of a service dependency graph with five layers and two different solutions. The dark gray services correspond with the services of the

Algorithm 1 Extended service dependency graph algorithm

```
 1: $newOutputs := I_R$
 2: $I_a := I_R$
 3: $i := 0$
 4: $Layers := \emptyset$
 5: repeat
 6:     $L_i := \emptyset$
 7:     for $L_j$ $(j < i)$ do
 8:         $I_a := I_a \cup Outputs\{L_j\}$
 9:     end for
10:     for Service $S_i \in$ Repository do
11:         $O_s := Outputs\{S_i\}$
12:         $isNewService := S_i \notin L_j (j < i)$
13:         $hasInputsAvailable := Inputs\{S_i\} \subseteq I_a$
14:         $usesNewInputs := Inputs\{S_i\} \cap newOutputs \neq \emptyset$
15:         if $isNewService \wedge hasInputsAvailable \wedge usesNewInputs$ then
16:             $L_i := L_i \cup S_i$
17:         end if
18:     end for
19:     $newOutputs := newOutputs \cup O_s - I_a$
20:     $Layers := Layers \cup L_i$
21:     $i = i + 1$
22: until $L_i \neq \emptyset$
```

solution with the largest runpath (the first and the last layers are not computed for the runpath). The two solutions uses parallelism (split) as in both cases, services > runpath. R_i and R_o are dummy services. R_i is a service that provides the request inputs, and R_o is a service that requires as inputs the requested outputs.

Once the SDG is generated, a search over it must be performed in order to find all optimal solutions. For this purpose, we implemented the search using the well-known A* search algorithm. This 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) handling one service or more in each layer. A detailed explanation of the composition search can be found in [5].

3 Experiments

In order to evaluate the correctness and the performance of the algorithm, a full validation has been done with eight public repositories from Web Service Challenge 2008 (WSC'08)[2]. The repositories have different degree of complexity, having from 158 to 8119 services. The solutions[3] provided by the WSC'08 are

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

[3] The data provided by WSC'08 are not optimal in all cases, as can be seen in the challenge results: http://cec2008.cs.georgetown.edu/wsc08/downloads/WSCResult.pdf

showed in Table 1. The second column indicates the number of services of the repository. The third column shows the minimum number of services required to reach the solution while the fourth column indicates the minimum solution path that can be obtained (according to WSC'08), and the fifth column shows the number of solutions. As can be seen, the complexity of the selected datasets is enough for a complete validation of our proposal.

Table 1. Web Service Challenge: Repository size & provided solutions

Test	Repo. services	Min. #services	Min.exec.path (runpath)	#solutions
WSC'08-1	158	10	3	3
WSC'08-2	558	5	3	4
WSC'08-3	604	40	23	1
WSC'08-4	1041	10	5	2
WSC'08-5	1090	20	8	2
WSC'08-6	2198	40	9	2
WSC'08-7	4113	20	12	2
WSC'08-8	8119	30	20	2

Table 2 shows the results[4] obtained and it is organized as follows: The second column indicates the number of services in the service dependency graph. The third column shows the number of solutions obtained by our algorithm. The fourth column indicates the number of steps executed by the A* search algorithm until the solution was reached. In the fifth column we show the elapsed time until a solution was found (including the time spent in the generation of the service dependency graph). The sixth indicates the number of services obtained by the algorithm and the last one shows the length of the execution path of the solution (runpath). Columns from 4 to 7 are duplicated to show the same information for the solutions with minimal runpath.

As can be seen, in all cases (except in WSC'08-6) the solution with minimal number of services is the solution with minimal runpath too. The first thing that must be noticed is that the solutions obtained by our algorithm are the best for all datasets (according to the solutions provided by WSC'08, see Table 1), except in the case of the dataset WSC'08-6, where our algorithm finds a solution with less number of services (35 vs 40) and a solution with less runpath (7 vs 10), as well as the offered by the WSC'08. Moreover, the algorithm finds all possible solutions for all datasets, showing a great performance as in all cases the bests solutions were found in a very short period of time. This feature is an important advantage over the other approximations since it is the first time that all solutions from the WS-Challenge 2008 (and not only the best solutions) are obtained.

[4] The algorithm was implemented using Java^{TM} JDK 1.6. 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 2. Algorithm results for the eight datasets

			Solution with min. Services				Solution with min. Runpath			
Test	Gr.serv	#Sol.	Iter.	Time(ms)	#Serv	Runpath	Iter.	Time(ms)	#Serv	Runpath
WSC'08-1	54	7	20	88	10	3	20	88	10	3
WSC'08-2	55	4	19	161	5	3	19	161	5	3
WSC'08-3	60	1	24	440	40	23	24	440	40	23
WSC'08-4	31	2	11	109	10	5	11	109	10	5
WSC'08-5	81	6	54	498	20	8	54	498	20	8
WSC'08-6	162	12	107	2328	35	14	110	2338	42	7
WSC'08-7	124	2	33	3375	20	12	33	3375	20	12
WSC'08-8	92	3	71	3636	30	20	71	3636	30	20

4 Conclusions and Future Work

In this paper we have presented an extended version of the heuristic-based search algorithm for automatic web service composition. The proposed algorithm allows to find all optimal compositions, with a minimal number of services and minimal runpath. Moreover, a full validation has been done using all datasets provided by the WS-Challenge'08, showing a great performance as our algorithm finds all optimal compositions in a very short time.

As future work we plan to improve our algorithm by including non-functional properties in our model, such as cost, reliability, throughput, etc. Quality of Services (QoS) characteristics are important criteria for building real world compositions. Our algorithm can be easily adapted to handle these features.

Acknowledgment

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

References

1. K. Chen, J. Xu, and S. Reiff-Marganiec. Markov-htn planning approach to enhance flexibility of automatic web service composition. In *ICWS 2009*, pages 9–16.
2. J. Hoffmann, P. Bertoli, and M. Pistore. Web Service Composition as Planning, Revisited: In Between Background Theories and Initial State Uncertainty. In *AAAI'07*.
3. W. Jiang, C. Zhang, Z. Huang, M. Chen, S. Hu, and Z. Liu. QSynth: A Tool for QoS-aware Automatic Service Composition. In *ICWS 2010*, pages 42–49.
4. S.C. Oh, J.Y. Lee, S.H. Cheong, S.M. Lim, M.W. Kim, S.S. Lee, J.B. Park, S.D. Noh, and M.M. Sohn. WSPR*: Web-Service Planner Augmented with A* Algorithm. In *2009 IEEE Conference on Commerce and Enterprise Computing*, pages 515–518.
5. P. Rodriguez-Mier, M. Mucientes, and M. Lama. Automatic web service composition with a heuristic-based search algorithm. In *ICWS 2011*. (Accepted).
6. Y. Yan, B. Xu, and Z. Gu. Automatic service composition using and/or graph. In *CEC 2008*, pages 335–338.