An Optimal and Complete Algorithm for Automatic Web Service Composition

Pablo Rodriguez-Mier, University of Santiago de Compostela, Spain

Manuel Mucientes, University of Santiago de Compostela, Spain

Juan C. Vidal, University of Santiago de Compostela, Spain

Manuel Lama, University of Santiago de Compostela, Spain

ABSTRACT

The ability of web services to build and integrate loosely-coupled systems has attracted a great deal of attention from researchers in the field of the automatic web service composition. The combination of different web services to build complex systems can be carried out using different control structures to coordinate the execution flow and, therefore, finding the optimal combination of web services represents a non-trivial search effort. Furthermore, the time restrictions together with the growing number of available services complicate further the composition problem. In this paper the authors present an optimal and complete algorithm which finds all valid compositions from the point of view of the semantic input-output message structure matching. Given a request, a service dependency graph which represents a suboptimal solution is dynamically generated. Then, the solution is improved using a backward heuristic search based on the A algorithm which finds all the possible solutions with different number of services and runpath. Moreover, in order to improve the scalability of our approach, a set of dynamic optimization techniques have been included. The proposal has been validated using eight different repositories from the Web Service Challenge 2008, obtaining all optimal solutions with minimal overhead.*

Keywords: *A* Algorithm (A Star Algorithm), Dependency Graph, Semantic Input-Output Matching, Web Service Composition, Web Services*

1. INTRODUCTION

Nowadays, Service-Oriented Architectures (SOA) (Papazoglou & Georgakopoulos, 2003) are gaining importance because of the ability to build interoperable services that can be shared over a network within multiple platforms. Thus, companies are starting to apply this principle to their business, allowing them to remain cost effective, flexible and competitive. Applications in SOA are built based on services consumed by clients that are not concerned with the underlying implementation. Specifically, web

DOI: 10.4018/jwsr.2012040101

services are the preferred standard-based way to realize SOA.

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 (Alonso, Casati, Kuno, & Machiraju, 2004). One of the advantages of web services is to enable greater and easier integration and interoperability among systems and applications through web service composition. This advantage allows web services to be combined by connecting their inputs and ouputs to create larger services (composite services) whose execution is orchestrated by a set of control structures defined in composition languages like WS-BPEL (Weerawarana, Curbera, Leymann, Storey, & Ferguson, 2005; Rouached, Fdhila, & Godart, 2010). Thus, the goal of web service composition is to construct new services from existing web services in order to satisfy a request (basically a set of provided inputs and a set of wanted outputs by the client) which cannot be solved by a single web service. The matching between inputs and outputs can either be done syntactically, using the information described in WSDL (Christensen, Curbera, Meredith, & Weerawarana, 2001), or semantically, using semantic markup languages like OWL-S (Burstein et al., 2004) or WSMO (de Bruijn et al., 2005).

The automatic composition problem may seem trivial problem when there are a limited number of services in a single-service architecture. However, the problem increases in complexity when the goal is to obtain optimal compositions over large web service repositories using different control structures to manage the composition flow. In fact, the web service composition problem can be reduced to the boolean satisfiability problem, i.e., the problem is NP-complete and therefore it cannot be solved in polynomial time (Lee & Kumara, 2005).

Research in this field has grown rapidly in recent years. Some approaches (Hoffmann, Bertoli, & Pistore, 2007; Sirin, Parsia, Wu, Hendler, & Nau, 2004; Klusch, Gerber, & Schmidt, 2005; Pistore, Barbon, Bertoli, Shaparau, & Traverso, 2004; Xu, Chen, & Reiff-Marganiec, 2011) treat the service composition as an artificial intelligence (AI) planning problem, where a sequence of actions lead from a initial state (inputs and preconditions) to a goal state (required outputs). These techniques work well when the repository size is relatively small and the number of constraints is high. 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 (Aversano & Taneja, 2006; Ghafarian & Kahani, 2009; Rodrıguez-Mier, Mucientes, Lama, & Couto, 2010) scale better than other techniques when the interactions among services and the number of constraints is huge. Despite being scalable, these techniques do not guarantee to obtain the optimal solution, and also are extremely slow and memory intensive. The most recent approaches (On & Larson, 2005; Kona, Bansal, Blake, & Gupta, 2008; Yan, Xu, & Gu, 2008; Wu, Li, Wu, & Yin, 2011; Weise, Bleul, Kirchhoff, & Geihs, 2008; Shiaa, Fladmark, & Thiell, 2008; Hennig & Balke, 2010; Hashemian & Mavaddat, 2006; Jiang, Zhang, Huang, Chen, Hu, & Liu, 2010), consider the problem as a graph/tree search problem, where a search algorithm is applied over a sub-optimal graph in order to find a optimal (or near-optimal) solution. These proposals are simpler than the AI planners due, in part, to the use of a smaller number of constraints during the search. However, most of these approaches rely on very complex dependency graphs that have not been optimized to reduce data redundancy. 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 from the point of view of the semantic input-output message structure matching, i.e., we do not take into consideration the non-functional properties (NFPs). The novelties of our proposal are:

1. The method is able to calculate, given a request, an extended service dependency graph which represents a valid but sub-optimal solution for the request.
2. The heuristic search algorithm, based on the well-known A*, finds all optimal solutions from the point of view of the number of services and execution path (runpath). This, it maximizes the parallel execution of services and minimizes the number of services.
3. We define set of optimizations to reduce the graph size, based on the redundancy analysis and service dominance.
4. We include a method to reduce dynamically the possible paths to explore during the search by filtering equivalent compositions.

We have validated our algorithm with the eight datasets defined by the Web Service Challenge 2008 (Bansal, Blake, Kona, Bleul, Weise, & Jaeger, 2008). Also we have compared our approach with the results of the participants of the Web Service Challenge 2008.

The rest of the paper is organized as follows: Section 2 describes the different approaches that have already been proposed. Section 3 introduces the basis of web service composition. Section 4 illustrates the proposed A* algorithm for web service composition. Section 5 presents some optimization techniques to improve the performance of the algorithm. Section 6 analyzes the algorithm with eight different repositories and compares the results with other approaches. Section 7 points out the conclusions.

2. RELATED WORK

Heuristic algorithms have proved their efficiency in the field of the automatic web service composition. Particularly, the use of graph-based and tree-based search algorithms has been studied before (Liang & Su, 2005; Milanovic & Malek, 2006) to solve a web service composition in large repositories, showing great results. 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. (2008) 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 speed up 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. (2008) 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. (2008) 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 algorithm does not guarantee to obtain the optimal compositions from the point of view of the number of services, as can be seen in

the results of the competition. Moreover, the authors have not implemented optimization techniques in order to improve the scalability of the algorithm.

Oh, Lee, and Kumara (2007) and Oh et al. (2009) propose a Web-Service Planner using the A* search algorithm (WSPR*), an improvement of the WSPR planner, which was at third place in the WSC'08. 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.

Wu et al. (2011) presented AWSP, an automatic web service planner based on heuristic state space search. In this work, an A* is used to search minimal compositions in terms of execution path. The search is performed using different operators which allow the movement from one state to another, adding a new service in each step. This movement can be done either forward or backward, although the last one is clearly better. To do this, two different heuristics were implemented based on a parameter distance defined by the authors. This approach has some drawbacks: firstly, authors do not consider the use of stratified methods previous to the search. These methods allow to quickly reduce the search space size, and can be used in dynamic environments as the computation of service graphs has not an important impact on the overall performance. This, in dynamic environments, where inputs and outputs can change, the recalculation of the graph can be done without affecting too much the performance. In second place, the algorithm cannot manage parallel execution of services. Third, they do not take into account the detection of redundancy, which can seriously affect search performance. Finally, in fourth place, more tests are required to confirm the advantages of this approach, comparing it with other similar AI planners as WSPR*.

Aiello, Benthem, and Khoury (2008) got the second place in the Web Service Challenge 2008 with RugCo, an automatic web service compositor. This algorithm uses a tree based search to find compositions that satisfy a request. The search is performed expanding nodes and resolving the new dependencies generated in each step until no more dependencies are discovered. Since during the search a large number of expanded nodes are generated, the authors introduce a heuristic approximation (beam search) to analyze only the most promising nodes. Despite the authors found solutions for the three datasets proposed in the WSC'08, the major drawbacks of this approach are: 1) the beam search does not guarantee to obtain optimal solutions, as only the most promising nodes are expanded, so the algorithm is neither complete nor optimal; 2) the search minimizes the number of services in the composition, but not the execution path; and 3) beam search does not scale well with the size of the search space, which implies bad performance in large datasets.

Weise, Bleul, Kirchhoff, and Geihs (2008) obtained the forth place in the WSC'08 with an architecture which combines three different algorithms (uninformed search based on ID-DFS, a greedy search and a genetic algorithm) (Weise, Bleul, Comes, & Geihs, 2008). The architecture integrates a module called "Strategy Planner" which decides the best algorithm in each case. The results obtained with this system are not surprising. The ID-DFS is an uninformed search based on the depth-first search (DFS) with iterative deepening (ID). This method is very simple and ineffective to solve a web service composition problem as the time complexity grows exponentially with the depth. When the dataset is too big for the ID-DFS algorithm, the greedy algorithm is used instead of the ID-DFS. This approach is very similar to the DFS, but a heuristic is used to sort the set of candidate nodes to explore. The greedy algorithm works as bad as the ID-DFS in the worst case scenario. On the other hand, a genetic algorithm is used for all those cases

where the ID-DFS and the greedy search cannot find a solution. This algorithm uses a set of evolutionary operators to obtain near-optimal compositions minimizing multiple objectives. However, the results obtained in the WSC'08 show the ineffectiveness of this approach. The major drawback of this algorithm is the fitness function. The fitness is measured by calculating two objectives: composition size and number of wanted (unsatisfied) parameters. This evaluation does not work well when the solutions have a long runpath and the last service or services provide all wanted parameters. In this scenario, there is no information about which solution is better until the complete composition is reached, so in each generation, the best individuals are those with a less number of services. This evaluation can prevent the algorithm to find a solution. Moreover, the algorithm is an order of a magnitude slower than the other approaches.

With this state of the art, we can conclude that the main differences between our proposal and other approaches are:

1. 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.
2. 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.
3. The detection of all valid compositions with different number of services and runpath. Other approaches only find an optimal composition with minimum number of services or minimum runpath.
4. The use of dynamic optimization during the search that reduces the number of possible paths to explore by combining equivalent combination of services.

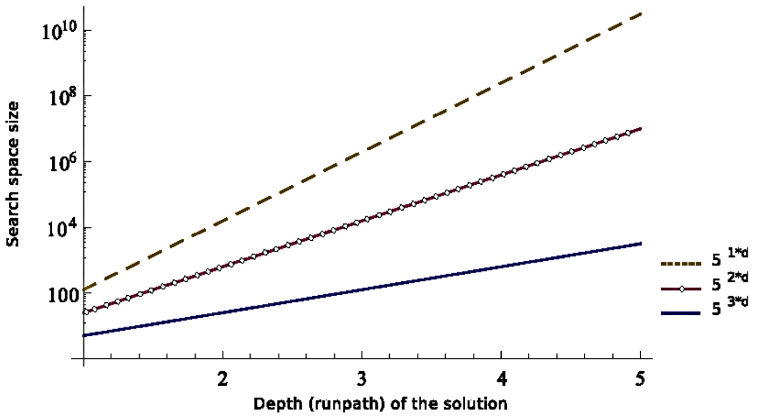
In the following sections we describe in detail the composition problem and how it can be solved with our proposal.

3. WEB SERVICE COMPOSITION

In order to compose web services, we must define the relationship among services. From a functional 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 $W_{in}=\{I_1,I_2,\ldots\}$ and a set of outputs $W_{out}=\{O_1,O_2,\ldots\}$. 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 $\left(R_{ij}=\left\{I_{in}^1,I_{in}^2,\ldots\right\}\right)$ provided by the requester, and a set of outputs $\left(R_{out}=\left\{O_{out}^1,O_{out}^2,\ldots\right\}\right)$ that the requester expects to obtain. Given a request $R_{user}=\{R_{in},R_{out}\}$, where $R_{user}=\{R_{in},R_{out}\}$ and $R_{out}=\left\{O_R^1,O_R^2,\ldots\right\}$, and given a web service $S=\{S_{in},S_{out}\}$ where $S_{in}=\left\{I_S^1,I_S^2,\ldots\right\}$ and $S_{out}=\left\{O_S^1,O_S^2,\ldots\right\}$, 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 construction 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 behavior of the execution flow

Figure 1. Search space size for n=1, n=2, n=3 (1, 2 and 3 inputs per service) and m=5 (5 services per output on average) with variable runpath



and the inputs/outputs of the services related to those structures. Despite the amount of different control structures defined in composition languages like WS-BPEL, we take into account only two of the most important ones: sequence and split. These structures allow building 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 outputs.

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 represents a set of services executed in parallel that can be expanded again. Figure 1 shows the size of the search space for different values of the runpath (d = 1...5), with n = 1, n = 2, n = 3 (one, two and three inputs respectively for each service in repository) and m = 5 (5 services per output on average).

As can be seen, this kind of composition has 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 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 Section 5.

4. 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, Nilsson, and Raphael (1968), is one of the most popular path finding 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 (Section 4.1.). Then, a reduction on the number of services is performed by eliminating unused services and combining equivalent services (Section 5). Finally, the A* search is applied over the reduced graph, which finds all optimal service compositions, with minimum number of services and execution path (Section 4.2.). These steps will be described in the following sections.

4.1. Extended 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, an extended 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. Figure 2 shows an example of a SDG with i layers and n services in each layer. The expression for a layer can be defined as follows:

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

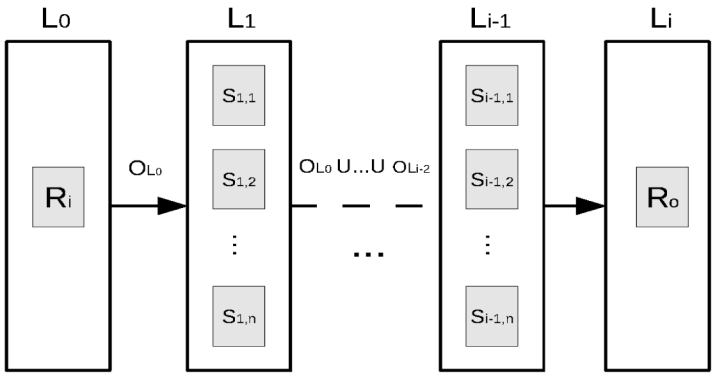
where, for each layer L_i:

- S_i is a service on the i_{th} layer.
- Oi 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. Algorithm 1 describes with pseudocode the construction of the graph iteratively. Lines 1-5 initialize the variables used throughout the algorithm: *newOutputs* (outputs generated in the last layer that have not been generated previously), *Ia* (available inputs for the current layer), *i* (current layer) and *Layers* (set of all generated layers). Note that *newOutputs* and *Ia* are initialized 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 6. Inside this loop, each layer is calculated following these steps:

1. Obtain all outputs from the previous layers. These outputs are the available inputs to the current layer (L. 8-10).
2. For each service in the repository.
   a. Check if the service has not appeared in previous layers (L. 13).
   b. Check if the service can be invoked (i.e., receives all its inputs from previous layers) (L.14).

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



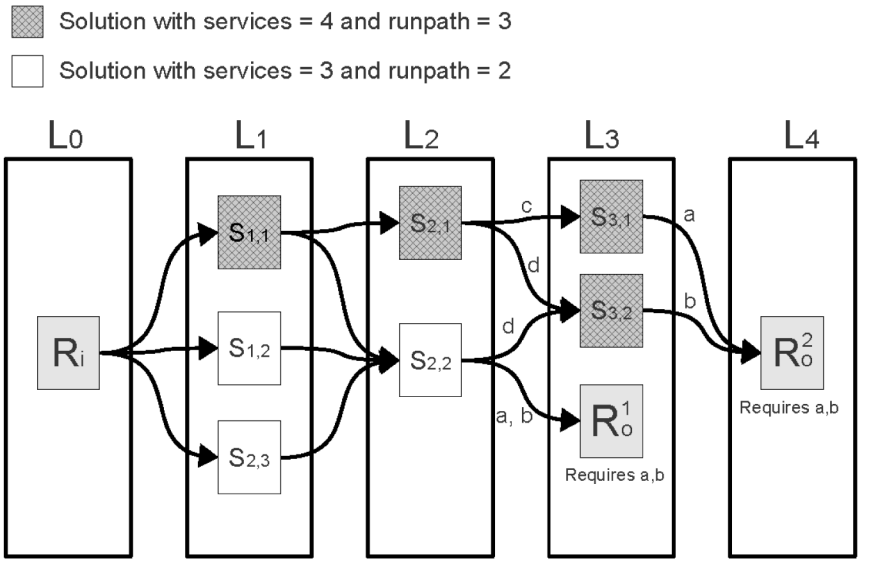
c. Check if the service use at least one output that has not been used previously (L. 15).
d. If (a), (b), and (c) are true, then the service is added to the current layer.

3. If the available inputs to this layer contain the wanted outputs (solution reached) and the previous layer produces at least one of the wanted outputs, then a dummy service (R_o^n) is added to the current layer. All R_o^n services are the initial nodes of the search (each initial node will lead to a solution with different runpath) (L.20-25).
4. Once all services are selected for the *i-th* layer, *newOutputs* is updated by adding the outputs of the *i-th* 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. 26).

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. Figure 3 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 solution with the largest runpath (the first and the last layers are not computed for the runpath). R_i, R_o^1 and R_o^2 are dummy services. R_i is a service which provides the requested inputs, R_o^1 is a service which uses the requested outputs (so there is a solution with a runpath of 2) and R_o^2 is a service which uses the requested outputs but in the layer 4 (runpath of 3). Thus, in this example, two different solutions for the same request can be observed: *Sequence*(R_i, *Split*, ($S_{1,2}$, $S_{2,3}$),$S_{2,2}$, R_o^1) and *Sequence*(R_i, $S_{1,1}$, $S_{2,1}$, *Split* ($S_{3,2}$, $S_{3,3}$), R_o^2).

Generally, stratified methods like this have a high performance, and allow reducing the total search space easily, as some constraints (in this case, inputs and outputs) are exploited to reduce the number of services that can be used. These methods work well in static environments, where the service information does not change. In real word, where the inputs and outputs, service availability and other parameters may change, these methods must be adapted. Basically, to ensure the validity in dynamic environments, a fast check can be done while the algorithm is searching for a solution. If any change is detected on any of the services selected by the search algorithm, the service dependency graph must be recalculated starting from the layer which contains the service. Specifically, two situations may occur:

Figure 3. Example of two solutions with different runpath and different number of services: Sequence(R_i, Split, ($S_{1,2}$, $S_{2,3}$),$S_{2,2}$, R_o^1) and Sequence(R_i, $S_{1,1}$, $S_{2,1}$, Split ($S_{3,2}$, $S_{3,3}$), R_o^2)



- *A service is not accessible*: given that our algorithm finds all possible solutions, when a service becomes unavailable, the solutions which contain the unavailable service must be discarded. The other solutions will be still valid.
- *A new service is available*: in this case, the service dependency graph must be partially rebuilt starting from layer L_{i+1}, where L_i is the layer at which the inputs required by the new service are provided (i.e., the layer which contains the service, according to the definition of L_i defined before).

4.2. 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* heuristic search. 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.

In order to perform the search 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 of the path. Thus, the starting node will only contain the service labeled as R_0 in Figure 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 among services from each list. These combinations will

Algorithm 1. Extended service dependency graph algorithm

```
1: newOutputs := I_R
2: I_a := I_R
3: i := 0
4: Layers := ∅
5: n := 0
6: repeat
7:    L_i := ∅
8:    for L_j (j < i) do
9:       I_a := I_a ∪ Outputs{L_j}
10:   end for
11:   for Service S_i ∈ Repository do
12:      O_s := Outputs{S_i}
13:      isNewService := S_i ∉ L_j(j < i)
14:      hasInputsAvailable := Inputs{S_i} ⊆ I_a
15:      usesNewInputs := Inputs{S_i} ∩ newOutputs ≠ ∅
16:      if isNewService ∧ hasInputsAvailable ∧ usesNewInputs then
17:         L_i := L_i ∪ S_i
18:      end if
19:   end for
20:   if I_a ⊇ wantedOutputs ∧ newOutputs ∩ wantedOutputs ≠ ∅ then
21:      R_o^n.inputs = wantedOutputs
22:      R_o^n.outputs = ∅
23:      L_i := L_i ∪ R_o^n
24:      n := n + 1
25:   end if
26:   newOutputs := newOutputs ∪ O_s − I_a
27:   Layers := Layers ∪ L_i
28:   i = i + 1
29: until L_i ≠ ∅
```

generate all possible neighbors from the current node.

3. Remove all equivalent neighbors. This process will be described in Section 5.

For example, given a node *N* with a service *S* in layer L_i, with I_S={*a*,*b*} and a set of services *X, Y, Z* in 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 behavior of the A* algorithm depends on two functions: *g*(*N*), the cost, and *h*(*N*), the 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 cost function should calculate the length of a composition based on the number of services. On this basis, we define a function *g*(*N*) as:

$$g\left(N\right) = \sum_{i=L_N}^{\neq L} \cos t\left(N_i\right) \quad (1)$$

where L_N is the first layer of the current composition service, #*L* is the number of layers and *cost* is a function that retrieves the number of

services from node N_i. The dummy services in a node will not contribute to this cost.

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 to 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:

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

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

$$f\left(N\right) = \sum \cos t\left(N_i\right) + distance\left(N_i\right) \tag{3}$$

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

5. OPTIMIZATION TECHNIQUES

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

5.1. Offline Service Compression

The essence of this technique is to replace equivalent services from each layer in the graph by the representative service, 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:

1. Remove unused services.
   a. Create an empty list *M*. This list will contain all the required inputs to get the solution.
   b. Create an empty list *U*. This list will contain all unused services.
   c. Traverse backwards the graph, starting from the final layer.
   d. For each layer L_i in the graph:
      i. Create an empty list *R*. This list will contain all the required inputs for this layer.
      ii. For each service *S* in the current layer:
         1. Check if $O_s \subseteq M$, where O_s are the service outputs. If *M* is empty, skip this step.
         2. If S meets the condition or M is empty, add all inputs from S to the list R.
         3. In other case, add S to the list U.
      iii. Add all inputs from *R* to the list *M*.
   e. Finally, remove from the graph, each service in *U*.
2. Detect and replace equivalent services by the representative service. 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. Check if $S_i \geq S_j$ for each service S_i and S_j from a group.
      ii. If S_i meets the previous restrictions, then select S_i as the representative service. S_j must be deleted.

One service S_i with parameters $P_{S_j} = \left\{P_{S_j}^1, P_{S_j}^2, ..., P_{S_j}^n\right\}$ dominates other service $S_j (S_i \geq S_j)$ with parameters $P_{S_j} = \left\{P_{S_j}^1, P_{S_j}^2, ..., P_{S_j}^n\right\}$ if:

$$\begin{aligned} &\forall k \in \left\{1,\ldots,n\right\} P_{S_i}^k \geq P_{S_j}^k \\ &\wedge \exists k \in \left\{1,...,n\right\}, P_{S_i}^k > P_{S_j}^k \end{aligned} \tag{4}$$

In our case, we consider only the outputs of a service $S_i\left(O_{S_i}\right)$ as the single parameter of S_i. The inputs are not considered as the services are grouped by the equivalence of their inputs. To clarify this point, the dominance between two services S_i and S_j with outputs O_{S_i} and O_{S_j} respectively can be done as follows:

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 $List_i$ 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 $List_i \supseteq List_j = X(a,b,c)$, $Y(a,c)$, 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, $S_i \gtrsim S_j$.

This technique can be used in both static and dynamic environments. Suppose that service *S*, which generates the outputs *a* and *b,* $(S \rightarrow (a,b))$ is the representative service of the group which contains services $U \rightarrow (a)$ and $V \rightarrow (b)$. If service *S* becomes unavailable, then services *U* and *V* can be selected to replace the representative service. The generation of all possible replacements can be done in the same way as the calculation of the neighborhood of a node, as explained in Section 4.2.

B. Online Node Reduction

This technique consists in the combination of equivalent neighbors during the A* search process. Given that a node 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.
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.

6. 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.

6.1. Web Service Challenge 2008 Datasets

In order to evaluate the correctness and the performance of our algorithm in different situations, we have carried out some experiments[1] using eight public repositories from Web Service Challenge 2008 (Georgetown University, 2008a). These repositories contain from 158 to 8119 services defined using WSDL. Also,

Table 1. Characteristics of the Web service challenge repositories

Test	#Services	#Inputs	#Outputs
WSC'01	158	735	778
WSC'02	558	2,972	2,890
WSC'03	604	3,254	3,129
WSC'04	1,041	5,781	5,611
WSC'05	1,090	5,816	5,953
WSC'06	2,198	12,218	11,831
WSC'07	4,113	22,324	22,392
WSC'08	8,119	44,569	44,628

inputs and outputs are semantically described in a XML file. Although there are other benchmark datasets for automatic web service composition (Oh & Lee, 2009; Georgetown University, 2005), the most efficient algorithms have been evaluated using the WSC datasets.

Table 1 shows in detail the characteristics of each dataset. The first column indicates the number of services in the repository (#Services). As can be seen, the number of services is variable and enough for a full validation. Table also shows the total number of inputs (#Inputs) and the total number of outputs (#Outputs). The solutions provided by the WSC'08 are showed in Table 2. Column '#Services" indicates the number of services for the shortest[2] solution (in number of services). Column "exec. path" shows the runpath for that solution. Finally, column "#Solutions" indicates the number of different solutions for that dataset.

6.2. Results

Our algorithm was implemented using JavaTM JDK 1.6 and tested with JavaTM 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 R Xeon R X5670 and 16GB RAM DDR-3. Table 3 shows the results obtained with a minimum runpath and a minimum number of services. Table 2 is organized as follows: the first column indicates the dataset name. The second column indicates the number of services in the service dependency graph (including dummy services). "#Sol" represents the number of solutions obtained by our algorithm, and "Iter." indicates the number of steps executed by the A* search algorithm until the solution was reached. "Time" is the elapsed time until a solution was found (including the time spent in the generation of the service dependency graph), while "#Serv." indicates the number of services obtained by the algorithm. Finally, "runpath" represents the length of the execution path of the solution. Columns 8-11 have the same meaning as columns 4-7 but for the solutions with minimum runpath.

As can be seen, in all cases (except in WSC'08-6) the solution with minimum number of services is the solution with minimum 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 2), except in the case of the dataset WSC'08-6, where our algorithm finds a solution with lower number of services (35 vs. 40) and a solution with shorter runpath (7 vs. 10). Our approach also scales well with the number of services (3,345 ms for the dataset with 4,113 services and 3,608 ms for the dataset with 8,119 services).

Moreover, the algorithm finds all possible solutions (Column "#Sol.") 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 shows that is possible to compose services automatically using an optimal and complete algorithm.

Table 2. Solutions provided by the WSC'08

Test	#Services	Exec. path	#Solutions
WSC'01	10	3	3
WSC'02	5	3	4
WSC'03	40	23	1
WSC'04	10	5	2
WSC'05	20	8	2
WSC'06	40	9	2
WSC'07	20	12	2
WSC'08	30	20	2

6.3. Comparison

In order to prove the validity of our approach, a comparison with the participants of the challenge has been done, following the rules defined by the WSC'08 (Georgetown University, 2008b). The quality of each composition is measured using three parameters (number of services, runpath and time) in accordance with the scoring rules as follows:

- +6 Points for finding the minimum set (Min. Services) of services that solves the challenge.
- +6 Points for finding the composition with the minimum execution length (Min. Execution) that solves the challenge.
  - +6 Points for the composition system which finds the minimum set of services or execution steps that solves the challenge in the fastest time (Time (ms)).
  - +4 Points for the composition system which solves the challenge in the second fastest time.
  - +2 Points for the composition system which solves the challenge in the third fastest time.

As can be seen, these rules are conflicting, given that some solutions have the minimum runpath but not the minimum number of services. For example, in the WSC'06 dataset, the solution with the minimum number of services (35 services) has a runpath of 14. On the other side, the solution with the minimum runpath has 42 services. With these rules, both solutions obtain 6 points, as the first one has the minimum number of services and the second one the minimum runpath. Despite our algorithm finds both solutions, only one solution is taken into account. Thus, our algorithm is clearly penalized by this rating. Regardless of this disadvantage, our algorithm obtained 44 points, the same score as the winners. Note that the time has not been

Table 3. 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	46	7	25	81	10	3	25	81	10	3
WSC'08-2	45	4	9	147	5	3	9	147	5	3
WSC'08-3	42	1	24	436	40	23	24	436	40	23
WSC'08-4	27	2	11	101	10	5	11	101	10	5
WSC'08-5	72	6	69	487	20	8	69	487	20	8
WSC'08-6	132	12	115	3,306	35	14	126	3,508	42	7
WSC'08-7	110	2	33	3,345	20	12	33	3,345	20	12
WSC'08-8	78	3	128	3,608	30	20	128	3,608	30	20

Table 4. Comparison with the participants of the WSC'08

	Tsinghua		Groningen		Pennsylvania		Kassel		USC	
WSC'08-4	Result	Points	Result	Points	Result	Points	Result	Points	**Result**	**Points**
Min services	10	6	10	6	10	6	10	6	**10**	**6**
Min execution	5	6	5	6	5	6	5	6	**5**	**6**
Time (ms)	312	2	219	4	28,078	0	828	0	**101**	**6**
WSC'08-5										
Min services	20	6	20	6	20	6	21	0	**20**	**6**
Min execution	8	6	10	0	8	6	8	6	**8**	**6**
Time (ms)	250	6	14,734	2	726,078	0	300,219	0	**487**	**4**
WSC'08-6										
Min services	46	0	37	6	-	0	-	0	**42/35**	**0/6**
Min execution	7	6	17	0	-	0	-	0	**7/14**	**6/0**
Time (ms)	406	6	241,672	2	-	0	-	0	**3,508/3,306**	**4/4**
TOTAL		44		32		24		18		**44**

measured under the same conditions because the source code of the other participants was not available. Therefore, the objective criteria for the comparative analysis should be only the number of services and the runpath.

If we compare the quality of the solutions, our algorithm finds better solutions than the other approaches. As can be seen in Table 4, the result with the minimum runpath for the dataset WSC'08-6 obtained by our algorithm has 42 services, while the University of Tsinghua obtained a solution with 46 services and the same runpath. On the other hand, if we compare the solution with the minimum number of services, our algorithm finds a composition with 35 services and a runpath of 14, which is clearly better than that provided by the University of Groningen with 37 services and a runpath of 17.

6.4. Optimization Effect

All the above experiments were performed using all the optimization techniques described in Section 5. In this section, we compare the effect of the optimization over the global performance on each dataset, and it is divided into three parts: (1) performance using offline service compression; (2) performance using online node reduction; and (3) performance improvement with both optimizations.

6.4.1. Offline Service Compression

The results are presented in Table 5. As can be seen, the average compression obtained over the graph using "Offline service compression" was close to 40%. The other columns show the average inputs per service, the average outputs per service and the average number of available services in the service dependency graph that provides the same output (with and without optimization). These values can be used to estimate the complexity for each dataset, as explained in Section 3. Note that the number of services per output decreases as the compression ratio increases (Column 11). This ratio has an important effect on the search performance. More specifically, a worse performance occurs when the number of available services per output is high, since the generation of neighbors in each step is slower. Despite the reduction obtained over the graph size, the complexity of the repository 6 still remained too high, so the algorithm cannot find a solution in a reasonable period of time (all tests were executed using a time limit of 5 minutes).

6.4.2. Online Node Reduction

This technique reports a large improvement in performance, as the algorithm obtains solutions

Table 5. Complexity of the service dependency graph (SDG) with and without using offline service compression

	SDG Services			Avg inputs/service		Avg outputs/service		Avg services/input		
	Non-optimized	Optimized	% Compr.	Non-optimized	Optimized	Non-optimized	Optimized	Non-optimized	Optimized	% Compr.
WSC'01	64	46	28.13	3.35	3.15	4.09	4.06	2.09	1.32	36.84
WSC'02	67	45	32.84	3.23	3.17	4.05	4.02	3.19	1.66	47.96
WSC'03	107	42	60.75	3.84	3.95	4.04	4.23	3.80	1.00	73.68
WSC'04	46	27	41.30	4.69	4.66	4.28	4.25	5.20	2.05	60.58
WSC'05	106	72	32.08	3.25	3.11	4.56	4.68	2.43	1.39	42.80
WSC'06	208	132	36.54	5.77	5.87	4.31	4.59	3.83	2.10	45.17
WSC'07	166	110	33.73	3.12	3.05	4.54	4.85	4.47	2.32	48.10
WSC'08	134	78	41.79	3.60	3.56	4.25	4.57	2.61	1.24	52.49
Average			38.39							50.95

in all repositories, including the WSC 2008-6 (23,704 ms, see Table 5). In most cases, this method obtains at least the same performance as the offline service compression. Table 6 shows in detail the time obtained for each dataset and using different optimizations. Column 2 indicates the time needed to get the solution with the minimum number of services without any optimization. Columns 3, 4, and 5 show the same information but using different techniques. Note that "Offline Service Compression" is not enough to obtain a solution in the dataset WSC'08-6.

6.4.3. Both Optimizations

After applying both techniques, our algorithm is able to solve the eight datasets showing a good performance. Table 7 shows the percentage of optimization obtained with the different techniques. In Figure 4, we compare the speedup[3] obtained with each optimization over the non-optimized algorithm. Note that with all optimizations, the speedup is over 1.0x, i.e., there is a substantial performance improvement. The improvement on the WSC'06 dataset cannot be measured as there are no results without optimizations, but a comparison can be done using only the results obtained with "Online node reduction" and "All optimizations." For this case, using the values in Table 6, we obtain a speedup of 7x with all optimizations (23,704 ms vs. 3,306 ms). This is due to the large number of equivalent combinations of services (neighbor nodes) that can be generated in each step.

7. CONCLUSION

In this paper we have presented a complete and optimal algorithm for automatic web service composition based on a heuristic search over a services graph. The graph has been optimized applying different techniques that reduce useless and equivalent services. The proposed A*-based composition algorithm is executed over the re-

Table 6. Performance of the algorithm using different optimizations

Test/Opt	No optimizations (ms)	Offline service comp. (ms)	Online node reduction (ms)	All optimizations (ms)
WSC'01	98.09	82.91	83.96	81.56
WSC'02	157.59	148.03	152.33	147.36
WSC'03	552.92	432.37	438.17	436.48
WSC'04	118.97	115.77	103.26	101.73
WSC'05	1,930.84	490.67	511.28	487.66
WSC'06	∞	∞	23,704.44	3,306.36
WSC'07	3,476.31	3,377.66	3,363.01	3,344.28
WSC'08	5,117.71	3,609.55	3,598.46	3,608.58

Table 7. Speedup obtained with the different optimizations

Speedup	Offline service compr.	Online node reduction	All optimizations
WSC'01	18 %	17 %	20 %
WSC'02	6 %	3 %	7 %
WSC'03	28 %	26 %	27 %
WSC'04	3 %	15 %	17 %
WSC'05	494 %	478 %	496 %
WSC'06	0,00 %	∞	∞
WSC'07	3 %	3 %	4 %
WSC'08	42 %	42 %	42 %

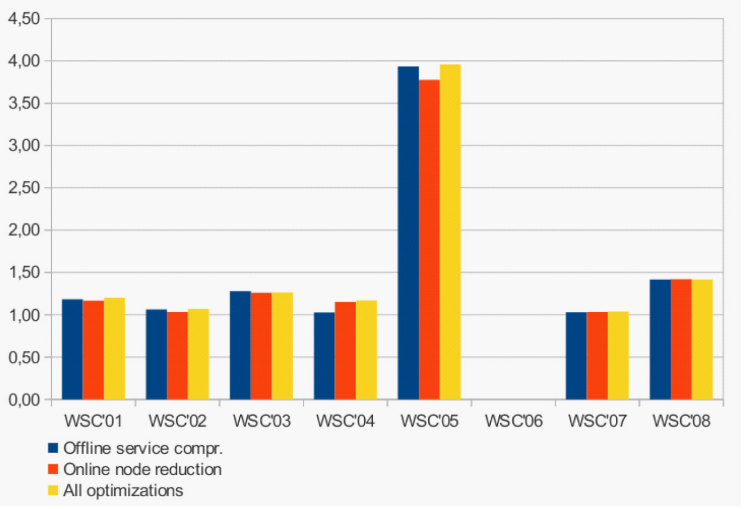
duced graph using dynamic node reduction and a cost function, which minimizes the number of services and maximizes the parallelization. Moreover, a full validation has been done using eight different repositories from Web Service Challenge 2008, showing a good performance as in all the tests the best solutions, regarding the number of services and runpath, were always found. Also, our algorithm is able to find all the existing solutions. This is not fulfilled by the other algorithms of the WSC'08.

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

ACKNOWLEDGMENTS

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 TIN2011-22935. Manuel Mucientes is supported by the

Figure 4. Speedup with different optimizations



Ramón y Cajal program of the Spanish Ministry of Science and Innovation.

REFERENCES

Aiello, M., Benthem, N., & Khoury, E. (2008, July). Visualizing compositions of services from large repositories. In *Proceedings of the 10th IEEE Conference on e-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, e-Commerce and e-Services* (pp. 359-362). Washington, DC: IEEE Computer Society.

Alonso, G., Casati, F., Kuno, F., & Machiraju, V. (2004). *Web services: Concepts, architectures and applications*. New York, NY: Springer.

Aversano, L., & Taneja, K. (2006). A genetic programming approach to support the design of service compositions. *International Journal of Computer Systems Science & Engineering*, *21*(4), 247–254.

Bansal, A., Blake, M. B., Kona, S., Bleul, S., Weise, T., & Jaeger, M. C. (2008). WSC-08: Continuing the Web services challenge. In *Proceedings of the 10th IEEE Conference on e-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, e-Commerce and e-Services* (pp. 351-354). Washington, DC: IEEE Computer Society.

Burstein, M., Hobbs, J., Lassila, O., Mcdermott, D., Mcilraith, S., & Narayanan, S. …Sycara, K. (2004). *OWL-S: Semantic markup for Web services*. Retrieved from http://www.w3.org/Submission/OWL-S/

Christensen, E., Curbera, F., Meredith, G., & Weerawarana, S. (2001). *Web Services Description Language (WSDL) 1.1*. Retrieved from http://www.w3.org/TR/wsdl

de Bruijn, J. D., Bussler, C., Domingue, J., Fensel, D., Hepp, M., & Keller, U. (2005). Web service modeling ontology. *Applied Ontology*, *1*(1), 76–106.

Georgetown University. (2005). *Introducing the Web service challenge*. Retrieved from http://ws-challenge.georgetown.edu/ws-challenge/Tech-Details.htm

Georgetown University. (2008a). *Challenge results*. Retrieved from http://cec2008.cs.georgetown.edu/wsc08/downloads/ChallengeResults.rar

Georgetown University. (2008b). *WSC results*. Retrieved from http://cec2008.cs.georgetown.edu/wsc08/downloads/WSCResult.pdf

Ghafarian, T., & Kahani, M. (2009, July). Semantic web service composition based on ant colony optimization method. In *Proceedings of the First International Conference on Networked Digital Technologies* (pp. 171-176). Washington, DC: IEEE Computer Society.

Hart, P., Nilsson, N., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100–107. doi:10.1109/TSSC.1968.300136

Hashemian, S., & Mavaddat, F. (2006). A graph-based framework for composition of stateless Web services. In *Proceedings of the European Conference on Web Services* (pp. 75-86). Washington, DC: IEEE Computer Society.

Hennig, P., & Balke, W.-T. (2010, July). Highly scalable Web service composition using binary tree-based parallelization. In *Proceedings of the IEEE International Conference on Web Services* (pp. 123-130). Washington, DC: IEEE Computer Society.

Hoffmann, J., Bertoli, P., & Pistore, M. (2007). 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* (pp. 1013-1018). Palo Alto, CA: AAAI.

Jiang, W., Zhang, C., Huang, Z., Chen, M., Hu, S., & Liu, Z. (2010, July). QSynth: A tool for QoS-aware automatic service composition. In *Proceedings of the IEEE International Conference on Web Services* (pp. 42-49). Washington, DC: IEEE Computer Society.

Klusch, M., Gerber, A., & Schmidt, M. (2005). Semantic Web service composition planning with OWLS-Xplan. In *Proceedings of the AAAI Fall Symposium on Semantic Web and Agents*, Arlington, VA. Palo Alto, CA: AAAI.

Kona, S., Bansal, A., Blake, M. B., & Gupta, G. (2008, September). Generalized semantics-based service composition. In *Proceedings of the IEEE International Conference on Web Services* (pp. 219-227). Washington, DC: IEEE Computer Society.

Lee, D., & Kumara, S. R. T. (2005). A comparative illustration of AI planning-based Web services composition. *SIGecom Exchanges*, *5*(5), 1–10.

Liang, Q., & Su, S. (2005). AND/OR graph and search algorithm for discovering composite Web services. *International Journal of Web Services Research*, *2*(4), 48–67. doi:10.4018/jwsr.2005100103

Milanovic, N., & Malek, M. (2006). Search strategies for automatic Web service composition. *International Journal of Web Services Research*, *3*(2), 1–32. doi:10.4018/jwsr.2006040101

Oh, S., & Lee, D. (2009). WSBen: A Web services discovery and composition benchmark toolkit. *International Journal of Web Services Research*, *6*(1), 1–19. doi:10.4018/jwsr.2009092301

Oh, S., Lee, D., & Kumara, S. (2007). Web service planner (WSPR): an effective and scalable web service composition algorithm. *International Journal of Web Services Research*, *4*(1), 1–22. doi:10.4018/jwsr.2007010101

Oh, S., Lee, J.-Y., Cheong, S.-H., Lim, S.-M., Kim, M.-W., & Lee, S.-S. …Sohn, M. M. (2009, July). WSPR*: Web-service planner augmented with A* algorithm. In *Proceedings of the IEEE Conference on Commerce and Enterprise Computing* (pp. 515-518). Washington, DC: IEEE Computer Society.

On, B., & Larson, E. (2005). BF*: Web services discovery and composition as graph search problem. In *Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service* (pp. 784-786). Washington, DC: IEEE Computer Society.

Papazoglou, M., & Georgakopoulos, D. (2003). Service-oriented computing. *Communications of the ACM*, *46*(10), 25–28.

Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., & Traverso, P. (2004). Planning and monitoring web service composition. In C. Bussler & D. Fensel (Eds.), *Proceedings of the 11th International Conference on Artificial Intelligence: Methodology, Systems and Applications* (LNCS 3192, pp. 106-115).

Rodríguez-Mier, P., Mucientes, M., Lama, M., & Couto, M. I. (2010). Composition of web services through genetic programming. *Evolutionary Intelligence*, *3*(3-4), 171–186. doi:10.1007/s12065-010-0042-z

Rouached, M., Fdhila, W., & Godart, C. (2010). Web services compositions modelling and choreographies analysis. *International Journal of Web Services Research*, *7*(2), 87–110. doi:10.4018/jwsr.2010040105

Shiaa, M., Fladmark, J., & Thiell, B. (2008, July). An incremental graph-based approach to automatic service composition. In *Proceedings of the IEEE International Conference on Services Computing* (pp. 397-404). Washington, DC: IEEE Computer Society.

Sirin, E., Parsia, B., Wu, D., Hendler, J., & Nau, D. (2004). HTN planning for Web service composition using SHOP2. *Web Semantics: Science Services and Agents on the World Wide Web*, *1*(4), 377–396. doi:10.1016/j.websem.2004.06.005

Weerawarana, S., Curbera, F., Leymann, F., Storey, T., & Ferguson, D. F. (2005). *Web services platform architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and more*. Upper Saddle River, NJ: Prentice Hall.

Weise, T., Bleul, S., Comes, D., & Geihs, K. (2008, June). Different approaches to semantic Web service composition. In *Proceedings of the Third International Conference on Internet and Web Applications and Services* (pp. 90-96). Washington, DC: IEEE Computer Society.

Weise, T., Bleul, S., Kirchhoff, M., & Geihs, K. (2008, July). Semantic Web service composition for service-oriented architectures. In *Proceedings of the 10th IEEE Conference on e-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, e-Commerce and e-Services* (pp. 355-358). Washington, DC: IEEE Computer Society.

Wu, B., Li, Y., Wu, J., & Yin, J. (2011). AWSP: An automatic Web service planner based on heuristic state space search. In *Proceedings of the IEEE International Conference on Web Services* (pp. 403-410). Washington, DC: IEEE Computer Society.

Xu, J., Chen, K., & Reiff-Marganiec, S. (2011). Using Markov decision process model with logic scoring of preference model to optimize HTN Web services composition. *International Journal of Web Services Research*, *8*(2), 53–73. doi:10.4018/jwsr.2011040103

Yan, Y., Xu, B., & Gu, Z. (2008). Automatic service composition using AND/OR graph. In *Proceedings of the 10th IEEE Conference on e-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, e-Commerce and e-Services* (pp. 335-338). Washington, DC: IEEE Computer Society.

ENDNOTES

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] Note that these values are only indicative. Smaller values have been found by our algorithm and by other participants.

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

Pablo Rodriguez-Mier is a PhD student within the Information Technologies Research Center (CITIUS) at the University of Santiago de Compostela. He received the MSc degree in computer science from the University of Santiago de Compostela in 2011. His research interests include automatic composition and service-oriented computing.

Manuel Mucientes received the MSc and PhD degrees in physics from the University of Santiago de Compostela, Spain, in 1997 and 2002, respectively. He is currently a Ramón y Cajal research fellow with the Information Technologies Research Center (CITIUS), University of Santiago de Compostela. He has authored or coauthored more than 50 papers in international journals, book chapters, and conferences. His current research interests are evolutionary algorithms, genetic fuzzy systems, motion planning, and control in robotics, visual SLAM (Simultaneous Localization and Mapping), web services and process mining.

Juan C. Vidal was born in Lausanne, Switzerland, in 1975. He received the BEng degree in computer science from the University of La Coruña, Spain, in 2000, worked as senior consultant for an IT firm several years, and received his PhD degree from the University of Santiago de Compostela, Spain, 2010. His research interests include knowledge discovery, semantic annotation, semantic modeling of workflows and services, and the use of artificial intelligence for business intelligence.

Manuel Lama is associate professor in the Department of Electronics and Computer Science at the University of Santiago de Compostela. He received his PhD in computer science from the University of Santiago de Compostela in 2000. His research interests focuses on discovery and composition of web services, semantic annotation, and process mining.