Graph-Based heuristic approach to Web Service Composition

Swathi.S¹, T.R.Vidhya² and Ramya RamPrasad ³

¹¹B.Tech, Department of Information Technology, Anand Institute of Higher Technology, Kazhipattur, India
²²M.E, Department of Information Technology, Anand Institute of Higher Technology, Kazhipattur, India

Abstract—Automatic web service composition has become more popular in recent times. These services are composed of several atomic services that comply with well defined process. Dealing with large amount of services is of major concern as it requires efficient algorithms for service composition. In this paper, we present a solution to optimize the computation effort in service composition using A* search algorithm. A service composition graph is modelled representing the external repository containing the services. Then, we compute all shortest paths between services using for the first time.

Index Terms— A* algorithm, Graph model Heuristic search, Web services composition.

I. INTRODUCTION

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network [16]. Web services are self-contained modular applications that can be created and executed over different platforms, and its features are defined using a standard XML-based language [1], [8]. Nowadays, Web Services that are available in Internet are used frequently and their number is growing quickly e.g. Flipkart, Ticketnew uses web services to display information to their potential customers. The characteristics of the Web Services include functional features that indicate the inputs and outputs required to invoke the execution of a web service; non-functional features such as reliability, cost, execution time, robustness etc..

Web services 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 fulfil the request. The involved web services together are considered as Composite web service. 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.

Several papers have dealt with composition of web services. Some approaches, such as [7,11,14] 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, availability, high computational cost and inability to maximize the parallel execution of web services.

Other approaches, such as [2,5,6,8,13,15] 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. Each time when a user request is made, composing the web services either statically or dynamically is tedious and expensive. Optimizing the composition effort which is one of the bigger challenges would improve the efficiency; reduce the response time while satisfying the user request.

This paper focuses on the problem of the web service composition as a graph search problem. In this approach, a directed graph built at the time of publishing representing the web services and the relationship between them. The optimized combination of all component services that compose targeted services is found by traversing the graph. Indeed, by generating the dependency graph before executing user queries, we optimize the composition time and costs.

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 and illustrates the proposed A* algorithm for web service composition.
II RELATED WORK

Many proposals have been made for performing web services composition in recent years. For a detailed survey, we refer to [1]-[8]. In this section, we present a brief overview of some techniques that deal with automatic web service composition. We consider only techniques that use service dependency information, graph models, and semantics. The simple idea behind dependency is that whenever a web service receives some input and returns some output, the output is somehow related or dependent on the given input. By using a graph model, the behaviour of available web services is represented in terms of their input-output information, as well as semantic information about the web data. A graph is a collection of vertices or 'nodes' and a collection of edges that connect pairs of vertices. A graph may be undirected, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another. A weighted graph is a graph where each edge has a weight (some real number) associated with it. The dependency graph is used in finding a composite service to satisfy a given request.

Most of composition graph-based methods build web services dependency graphs at runtime. They use a search algorithm for traversing dependency graphs in order to compose services. The main difference between these methods is attributed to how they search the dependency graph. A*, Dijkstra, Floyd, Forward chaining, backward chaining and bidirectional search algorithms are examples of the most common search techniques.

Hashemian, Mavaddat [9] store I/O dependencies between available Web services in their dependency graph, and then build composite services by applying a graph search algorithm. In their graph, each service and I/O parameter is represented as a vertex, service’s input and output are represented as incoming and outgoing edges, respectively. The authors consider only the matching and dependencies between input and output parameters without considering functional semantics, thus they cannot guarantee that the generated composite services provides the requested functionality correctly.

In [3], the authors use the backward chaining method in combination with depth first search to get the required services for a composite task. Their solution is rather abstract and does not clearly discuss execution plan generation algorithm. Arpinar et al. [2] present an approach which not only use graphs for web service composition, but also use semantic similarity as we present in this work. They consider edges with weights and deploy a shortest-path dynamic programming algorithm based on Bellman-Ford’s algorithm for computing the shortest path. For cost, the authors consider the execution time of each service and input/output similarity but they don’t take into consideration the services’ non-functional attributes.

Gekas et al. [6] develop a service composition registry as a hyperlinked graph network with no size restrictions, and dynamically analyze its structure to derive useful heuristics to guide the composition process. Services are represented in a graph network and this graph is created and explored during the composition process to find a possible path from the initial state to a final state. In order to reduce the time of searching, a set of heuristics is used. But according to the authors, creating the graph at the time of composition is very costly in term of computation and limits the applicability of graph-based approaches to the problem of web service composition.

Talantikite et al. [15] propose to pre-compute and store a network of services that are linked by their I/O parameters. The link is built by using semantic similarity functions based on ontology. They represent the service network using a graph structure. Their approach utilizes backward chaining and depth-first search algorithms to find sub-graphs that contain services to accomplish the requested task. They propose a way to select an optimal plan in case of finding more than one plan. However, they also create the graph at the time of composition which incurs substantial overhead.

Hajar Elmaghraoui, Imane ZaouI, D. Chiadmi [5] Propose a graph based solution which models the semantic relationship between the involved web services as directed graph and compute the shortest paths using for the first time, therefore optimize the composition by reducing the computational effort at the time of composition. Their approach uses the Floyd-Warshall algorithm.

Guluru,P Niyogi,R [7] have focused on Multi-input and single output web services composition. Two algorithms are proposed for finding the composition. First algorithm uses Breadth First Search to find all the paths from input to output nodes and merge them but it does not provide an optimal number of services. The second algorithm incorporates cost parameters for the web services to overcome the limitation of the first algorithm.
In our proposal, based on A* algorithm, a web service dependency graph representing the repository of web services is generated. Then, the A* search is applied over the graph, which finds an optimal service composition, with minimal number of services and execution path.

### III A* ALGORITHM FOR WEB SERVICES COMPOSITION

The algorithm that is selected to solve the composition problem must be efficient in terms of both complexity and time. Large repositories containing the services require that the space used by the algorithm is minimal. This problem is solved by the usage of cost-based heuristics and optimization techniques during data processing. The composition process is depicted in fig 1.

The process begins with receiving the user request, which consists of the Input and output desired from the web service. Based on the user request, the 'start' and 'goal' nodes of the final composition setup are identified. Based on the services available in the repository and the dependencies between them, the Service Dependency Graph is constructed at publish time. The A* algorithm is applied to the graph to find the path between the identified start and goal nodes. If multiple path exists, then the shortest path is returned. The final composition candidates that is returned to the user consists of the start, goal and intermediate nodes.

A* algorithm is widely used in graph traversal and path finding. The functions used by the algorithm are:

1. Heuristic function $h(n)$ to estimate the cost from the current node to a goal node.
2. $g(n)$ to calculate the cost from the starting node to the current node.

The search cost is defined as $f(n) = g(n) + h(n)$

It is important to select the ideal heuristic function $h(n)$ as it directly impacts the speed of the search process. A heuristic function that overestimates the cost to reach the goal ends up with a solution that has a higher cost than the best or optimal one.

Our proposal, based on A* algorithm, involves computing a web service dependency graph based on the steps specified in section A. The start and goal nodes are identified based on user request. The A* algorithm is applied to find the optimal service composition, with minimal number of services. These steps will be described in the following sections.

#### A. Web service dependency graph

The local web service repository contains web services and their inputs and outputs. They also contain the dependencies between the services. This information is used to generate the Service Dependency Graph (SDG). When the input of a service is equivalent to the output of another service, then the second service is dependent on the first and these two services can be executed sequentially.

The SDG is a directed graph $G = (V,E)$ where $V$ is a set of vertices representing a set of web services and $E$ is the set of edges representing the dependencies that exist between them.
Steps to create the SDG:

1) A vertex $V_i$ is created to represent the web service $WS_i$ in the repository.
2) An edge $E_i$ is introduced between $V_i$ and $V_j$ when a dependency exists between them.
3) A vertex with multiple inputs will have the said number of arrows directed towards it, and the directed edge emerging from it represents the single output.
4) The graph is constructed so that every pair of dependent vertices has an edge connecting them.

<table>
<thead>
<tr>
<th>Web Service</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS1</td>
<td>{a, b}</td>
<td>{d}</td>
</tr>
<tr>
<td>WS2</td>
<td>{a}</td>
<td>{c}</td>
</tr>
<tr>
<td>WS3</td>
<td>{d}</td>
<td>{e}</td>
</tr>
<tr>
<td>WS4</td>
<td>{d, e}</td>
<td>{f}</td>
</tr>
<tr>
<td>WS5</td>
<td>{c, d}</td>
<td>{h}</td>
</tr>
<tr>
<td>WS6</td>
<td>{c}</td>
<td>{i}</td>
</tr>
<tr>
<td>WS7</td>
<td>{h, i}</td>
<td>{j}</td>
</tr>
<tr>
<td>WS8</td>
<td>{f, h}</td>
<td>{k}</td>
</tr>
<tr>
<td>WS9</td>
<td>{e}</td>
<td>{k}</td>
</tr>
<tr>
<td>WS10</td>
<td>{c, x}</td>
<td>{g}</td>
</tr>
<tr>
<td>WS11</td>
<td>{g}</td>
<td>{j}</td>
</tr>
</tbody>
</table>

For the sample web service repository (see Table I), the Service Dependency Graph is shown in fig 2.

For the input set as $\{a, b\}$ and output as $\{k\}$, we have two possible paths:

The minimal cost solution path is represented in Figure 3.

**B. Identifying start and goal nodes**

The directed graph SDG is being constructed during the publish time, represents the services in the repository from which the vertices are identified upon receiving the user request for service composition. The vertices represent the input and output parameters and the graph is temporarily updated as:

- Create a start node that is connected to services (nodes) containing at least one input given by the service requester.
- A goal node is created that is connected to all services that provide the requested output.

The service composition which is based on the shortest path from the input (represented by start node) to the output (represented by goal node) is guided by these additional nodes.

**C. A* search algorithm**

Once the graph is calculated, a search over it must be performed. The search algorithm will traverse the graph from the start node to the goal node identified from the user request. The heuristic algorithm is based on A* pathfinder. The three principal concepts in this algorithm are the neighbourhood function, cost function and the heuristic function.

During the search process multiple solutions may be found, each having different cost to satisfy the service composition request. A solution with minimum number of services is preferred over a solution with more number of services.

The composition request is solved the following way: A cost of 100 is assigned to each node and zero to Input and output nodes as they are present.
in all the possible paths and do not contribute to comparison. We use function $S(j)$ to denote cost of reaching node $j$ in the graph from given input nodes. $S(j)$ is initialized with the following values depending on type of node $j$.

$S(j) = 0$ if $j$ is input node.

$S(j) = L$ if $j$ is not there in the input list.

Since Node ‘$j$’ is not part of the solution graph initially it is assigned a larger value $L$, once node ‘$j$’ is explored while finding the final graph setup it is updated with minimal cost possible to reach ‘$j$’. While the search is carried out $S(j)$ i.e. Cost of reaching node ‘$j$’ from inputs is calculated using the following formulae:

If ‘$j$’ is an operation node having parents $ip_1, ip_2, ip_3... ip_i$ then
Cost of reaching node $j$ is given by

$$S(j) = \min(C(p_j)+E(j))$$

In the Service Dependency Graph every node has following attributes namely $g(n) =$ cost of reaching from start node, cost of node.

The composition algorithm is presented below:
// input request: input and output the service
// output: Solution Graph

1. Add all Inputs of the service request to open list. open contains all the nodes that are yet to be traversed in the Service Dependency graph.
2. Update CFS as zero for all the nodes in open list that is created in previous step.
3. Create a priority queue named queue with the nodes in the list.
4. Get the node current at the front of the queue in the case it is not empty. If current is output of the request then skip to step 10. otherwise go to step 5.
5. Get all the neighbours of the current node and store them in adNodes list. //adNodes contains all the neighbours of the current node.
6. If $next.CFS$ (cost) is more than sum of $next.cost$ and current.CFS then update $next.CFS$ to sum of $next.cost$ and current.CFS; update next node's predecessor as current. If next is not explored earlier and is not in the queue mark next as known and add it to queue.
7. If all the predecessors of the next node are known (this can be verified from the repository) then update next node's CFS as sum of all predecessor CFS and update next to the closed list. If next is not explored earlier and not in queue mark next as known and add it to queue.
8. Display the solution graph by traversing the subsequent marked Service Dependency Graph nodes.

The above algorithm gives the minimal cost and shortest path solution graph when there are multiple candidate solution graphs available to satisfy the particular user goal request with given inputs and outputs.

IV. CONCLUSION

Web service composition is an emerging trend among large scale organisations that are making their services available online. In this paper we have proposed a service composition method that exploits the graph structure using a shortest path search algorithm. The composition process increases the scope of atomic web services by combining them and improving their functionalities. The proposed A* algorithm makes use of a cost-based heuristic to efficiently find the optimum solution to the user request.

ACKNOWLEDGMENT

The authors are thankful to Mrs.Ramya RamPrasad for her guidance and help with the review process. They are also grateful for the helpful comments and suggestions from the Professors and the Head of the Department for this paper.

REFERENCES


Swathi S is pursuing her bachelor’s degree in Information Technology from Anand Institute of Higher Technology and is set to graduate in 2015. Her current research interests include web services.

T.R. Vidhya is pursuing her bachelor’s degree in Information Technology from Anand Institute of Higher Technology and is set to graduate in 2015. Her current research interests include web services.

Mrs. Ramya Ram Prasad, has completed her Master degree in Computer Science and Engineering. She is working as an Assistant Professor in Anand Institute of Higher Technology, Chennai, Tamil Nadu, India and having an academic experience of 7 years and 2 months. Her field of interest is Web Services and its Composition. She has guided projects for UG and PG students in Engineering streams and published papers in National and International Conferences. To discover new methods in strengthening her professional and teaching capabilities and to meet the challenges of educating students for personal, professional life she has attended several seminars and faculty developments programs conducted by various reputed Universities and Engineering institutions.