A QoS-aware Web Service Composition Approach Based on Genetic Programming and Graph Databases

Alexandre Sawczuk da Silva[1], Ewan Moshi[1], Hui Ma[1], and Sven Hartmann[2]

[1] School of Engineering and Computer Science, Victoria University of Wellington
PO Box 600, Wellington 6140, New Zealand
{sawczualex, ewan.moshi, hui.ma}@ecs.vuw.ac.nz
[2] Department of Informatics, Clausthal University of Technology
Julius-Albert-Strasse 4, Clausthal-Zellerfeld D-38678, Germany
sven.hartmann@tu-clausthal.de

Abstract. A Web service can be thought of as a software module designed to accomplish specific tasks over the Internet. Web services are very popular, as they encourage code reuse as opposed to re-implementing already existing functionality. The process of combining multiple Web services is known as Web service composition. Previous attempts at automatically generating compositions have made use of genetic programming to optimize compositions, or introduced databases to keep track of relationships between services. This paper presents an approach that combines these two ideas, generating new compositions based on information stored in a graph database and then optimising their quality using genetic programming. Experiments were conducted comparing the performance of the newly proposed approach against that of existing works. Results show that the new approach executes faster than the previously proposed works, though it does not always reach the same solution quality as the compositions produced by them. Despite this, the experiments demonstrate that the fundamental idea of combining graph databases and genetic programming for Web service composition is feasible and a promising area of investigation.

1 Introduction

A Web service is a software module designed to accomplish specific tasks over the Internet [8], with the capability of being executed in different platforms [3]. Each Web service requires a set of inputs to be provided and produces a set of outputs. Web services by their very nature promote code reuse and simplify the process of information sharing, as developers can simply invoke the Web service as opposed to rewriting the functionality from scratch [7]. There are problems that a single Web service cannot address, and as a result, multiple Web services are required. This process of combining multiple Web services to solve more complex requests is known as Web service composition [6].

Much research has gone into developing approaches that automatically yield the best possible composition for the given task. Some of these approaches utilise a directed acyclic graph (DAG) to represent the interaction between the Web services with regards to the required inputs and produced outputs [8,9,4]. Before they can find suitable compositions, these methods must load a repository of Web services and discover the relationships between the inputs and outputs of different services. Naturally, repeating this process every time a composition is to be created becomes quite onerous. Genetic Programming (GP) approaches have also been applied for composition, though they present limitations such as producing solutions that are not guaranteed to be functionally correct (i.e. solutions that are not fully executable) [4], employing fitness functions that do not take into consideration the Quality of Service (QoS) of the generated solution [8], and containing redundancies in the GP trees, namely, the reappearance of Web services at different levels of the tree [9]. Similarly, there have been previous attempts at using databases (relational and graph databases) to solve the Web service composition problem [5,12]. These approaches are generally efficient, as they store service repositories and dependencies in the database. However, these approaches are typically not effective at performing global QoS optimization.

Given these limitations, the objective of this paper is to propose a QoS-aware Web service composition approach that combines GP with a graph database. The key idea is to manage services and their connections using the database, then use that information to create and optimize solutions in GP. The advantage of combining these two components is that together they prevent the repository from being repeatedly loaded, while also improving the QoS of the solutions produced during the composition process.

2 Background and Related Work

A typical example of an automated Web service composition is the travel planning scenario [10]. In this scenario, the solution can automatically book hotels and flights according to the customer's request. A *composition request* R is provided to the composition system, specifying I_R input information (such as the destination, departure date, and duration of stay) and the desired O_R output information (such as the return ticket and hotel booking). The system then creates a Web service composition that satisfies this request by combining services from the repository. This Web service composition is illustrated in Figure 1.

In this work, four QoS attributes are considered [11]: availability (A), which is the probability that a service will be available when a request is sent, reliability (R), which is the probability that a service will respond appropriately and timely, cost (C), which is the financial cost associated with invoking the service, and time (T), which indicates the length of time needed for a service to respond to a request. For availability and reliability, higher values indicate better quality, whereas lower values indicate better quality for cost and time. These quality attributes are associated with each atomic service in the repository, and overall

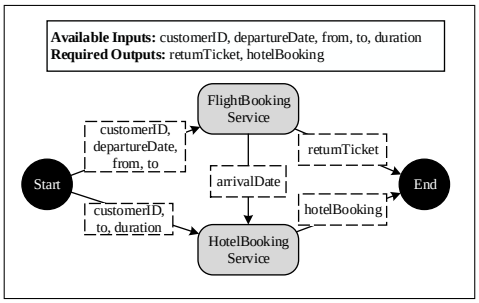


Fig. 1: A Web service composition for the travel planning scenario.

QoS attributes can be completed for a composition by aggregating individual QoS values according to the constructs included in the composition [1]:

- **Sequence construct:** Web services are connected sequentially, so that some outputs of the first service are used as the inputs of the subsequent one. To calculate the aggregate availability and reliability for this construct, values of all the individual services in the composition are multiplied. To calculate the aggregate cost and time, the values of each individual service are added.
- **Parallel construct:** Web services are independently executed in parallel, so that the inputs and outputs of each Web service are produced independently. The same calculations used for the sequential construct are employed here to calculate the availability, reliability, and cost. However, in the case of total time, we must select the Web service with the highest execution time, as the other services in the construct will be executed in parallel.

QoS values are normalised to ensure their contribution to the fitness calculation is proportional. The lower bounds are obtained from the service in the repository with the smallest value for the attribute in question (except for A_{min} and R_{min}, which are both 0), and the upper bounds are obtained from the service with the largest value (for T_{max} and C_{max}, values are multiplied by the size of the repository as an estimate of the largest possible solution). T and C are offset during the normalisation so that higher values denote better quality.

2.1 Related Work

The work presented in [4] automatically produces compositions by building a GP tree using workflow constructs as the non-terminal nodes and atomic Web services as the terminal nodes. One of the underlying problems of this approach is the random initialisation of the population, which results in compositions that are not guaranteed to be *functionally correct* (i.e. a composition that is fully executable given the available input). The work in [8] represents its candidates as a graph, which reduces the associated complexity from the verification of node dependencies while ensuring functional correctness of solutions. The problem

with this approach is that it cannot handle QoS-aware Web service compositions, meaning that the fitness function does not consider the QoS of candidates.

The authors of [5] propose an approach that uses relational databases to store Web services and their dependencies. This approach utilises a database to prevent loading data into memory. It employs an algorithm that creates a path between Web services in the repository. The results are then stored in the relational database for use to satisfy future requests. An issue with this approach is that paths need to be regenerated if a new Web service is introduced in the repository. Additionally, the approach only considers the time property when measuring the QoS of the compositions.

Finally, the approach presented in [12] makes use of a graph database to store services dependencies efficiently and in an easily modifiable way. When receiving a composition request, a subgraph of relevant nodes is filtered from the original dependency graph. Then, a number of functionally correct composition candidates are generated from within this subgraph, and the composition with the best overall QoS is chosen as the solution. Despite being an improvement on the relational database strategy described above, this work does not optimise the QoS attributes of the composition solution.

3 Graph Database and GP Approach

The approach presented in this paper extends the graph database work done in [12], introducing genetic programming as a means of improving the QoS of composition solutions. This process is summarised in Algorithm 1, and the core elements are discussed in the following subsections.

3.1 Fitness Function

The fitness function used to evaluate candidates is based on the one presented in [9] but without penalties, as our DAG-to-tree algorithm ensures that all trees are functionally correct. The values produced by the fitness function range from 0 to 1, with 1 being the best possible solution and 0 being the worst. The fitness function is $Fitness_i = w_1 A_i + w_2 R_i + w_3 T_i + w_4 C_i$, where A_i, R_i, T_i and C_i denote the normalized *availability, reliability, execution cost*, and *response time* of the candidate i, and the weights w_i are rational non-negative numbers where $w_1 + w_2 + w_3 + w_4 = 1$. Each weight is associated with a quality attribute, and their values are configured by users to reflect the importance of each attribute.

3.2 Initialisation

The initialisation begins by obtaining a composition in the form of a DAG with a single start node, which is done by querying the graph database [12]. Before the evolutionary process begins the DAG for each candidate is transformed into a corresponding GP tree. We begin by dividing the DAG into layers, which are identified by traversing every node in the DAG and finding the longest path

ALGORITHM 1. Steps in the proposed graph database and GP composition approach.

Data: Composition request $R(I_R, O_R)$
Result: Best composition solution found

```
Create a graph database for all available Web services in the repository;
From the initial graph database, create a reduced graph database which
contains only the Web services related to the given task's inputs and outputs;
Initialise a population by creating compositions that can satisfy the requested
task from the reduced graph database, transforming them from DAG into a
GP tree form;
Evaluate the initial population according to the fitness function;
while Stopping criteria not met do
    Select the fittest individuals for mating;
    Apply crossover and mutation operators to individuals in order to
    generate offspring;
    Evaluate the fitness of the newly created individuals;
    Replace the least fit individuals in the population with the newly created
    offspring;
return Fittest solution encountered
```

from the start to each node. The length of this path is the node's level (layer number). Figure 2a shows an example of this process. The layer information is then used to build a GP tree. This is done by Algorithm 2, which traverses the layers in reverse and creates sequence, parallel, and terminal nodes depending on the situation. Figure 2b shows a tree version of Figure 2a.

3.3 Crossover and Mutation

The crossover operation ensures that functional correctness is maintained by only allowing two subtrees with equivalent functionality in terms of inputs and outputs, i.e. two *compatible* subtrees, to be swapped. The operation begins by selecting two random subtrees from two different candidates. If the inputs and outputs contained in the root of each subtree are compatible, then the two subtrees can be swapped and functional correctness is still maintained, otherwise another pair is sought. In case there are no compatible nodes across two candidates, crossover does not occur. The mutation operation ensures functional correctness by restricting the newly generated subtree to satisfy the outputs of the subtree that was selected for the mutation. The operation randomly selects a subtree from a candidate, then randomly creates a new subtree based on the inputs and outputs of the selected node and following the initialisation principles described earlier. Finally, the selected subtree is replaced with the new subtree.

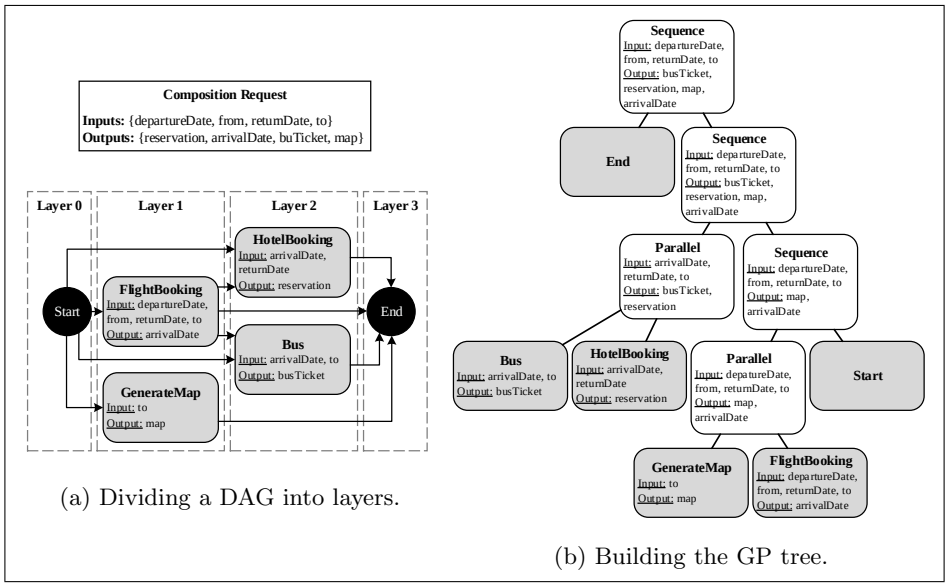
4 Experimental Design and Results

Experimental comparisons were conducted against GraphEvol [8] and a graph database approach [12]. The evaluation of the candidate solutions produced by

ALGORITHM 2. Converting the DAG to a tree.

```
   Data: NodeLayers
   Result: Converted GP Tree
 1 Initialize tree;
 2 Create TreeNode previous = null;
 3 for i = |NodeLayers| -1; i >=0 do
 4     Create SequenceNode sequenceCurrent with no parent;
 5     if previous is not null then
 6         if node in nodelayer i is start node then
 7             Create TerminalNode startNode with previous as parent;
 8             Add startNode to children of previous;
 9             Break;
10         else
11             Set parent of sequenceCurrent to previous;
12             Add sequenceCurrent to children of previous;
13     if number of nodes in current layer = 1 then
14         Create TerminalTreeNode n with the Web service's name and
           sequenceCurrent as parent;
15         Add n to children of sequenceCurrent;
16     else
17         Create ParallelNode parallel with sequenceCurrent as the parent;
18         foreach Node in layer i do
19             Create TerminalTreeNode n with the Web service's name and
               parallel as parent;
20             Add n to children of parallel;
21         Add parallel to children of sequenceCurrent;
22     Add sequenceCurrent to the tree;
23     previous = sequenceCurrent;
24     i = i - 1;
```

these two approaches was performed using the fitness function proposed in this paper. WSC-2008 [2] was the benchmark dataset chosen for the evaluation. Each approach was run 30 independent times on a personal computer with an i7 CPU (3.6 GHz) and 8 GB RAM. In the case of GraphEvol, 500 individuals were evolved for 51 generations, with a crossover probability of 0.8, a mutation probability of 0.1, and a reproduction probability of 0.1. Tournament selection was used, with a tournament size of 2. For the proposed GP approach, 30 individuals were evolved for 50 generations, with a crossover probability of 0.9, a mutation probability of 0.1, and a tournament selection strategy with size 2. Finally, the graph database approach randomly produced 30 solutions for each run, and selected the one with the best quality. For all approaches, fitness function weights were set to 0.25 for each attribute. Experimental results are shown in Table 1, which contains the mean and standard deviation for the execution time and fitness of each approach. For displaying purposes, the QoS attributes of the fi-



(a) Dividing a DAG into layers.

(b) Building the GP tree.

Fig. 2: Initialisation process.

Table 1: Mean and standard deviation for the execution time and solution fitness for the three approaches. Significantly lower values are indicated using ↓.

Dataset	GP Approach		Graph Database Approach		GraphEvol Approach	
2008	Time (ms)	Fitness	Time (ms)	Fitness	Time (ms)	Fitness
1	22.44 ± 0.66 ↓	0.46 ± 0.12 ↓	2197.90 ± 329	0.521 ± 0.169	4845.57 ± 315.42	0.645 ± 0.139
2	369.38 ± 231.27 ↓	0.58 ± 0.07 ↓	5347.13 ± 880	0.48 ± 0.152	3699.77 ± 364.57	0.906 ± 0
3	91.94 ± 2.99 ↓	0.36 ± 0.11 ↓	10961.53 ± 790	0.387 ± 0.1	17221.53 ± 764.85	0.176 ± 0.045
4	201.04 ± 73.11 ↓	0.43 ± 0.07 ↓	3885.70 ± 399	0.431 ± 0.066	6076.7 ± 281.58	0.305 ± 0.066
5	61.25 ± 1.68 ↓	0.37 ± 0.11 ↓	4510.6 ± 468	0.403 ± 0.128	10444.2 ± 572.59	0.164 ± 0.046
6	-	-	258503.33 ± 42324	0.407 ± 0.089	22183.53 ± 1639	0.228 ± 0.06
7	190.6 ± 26.89 ↓	0.38 ± 0.09 ↓	17839.77 ± 763	0.457 ± 0.097	20304.37 ± 1257	0.316 ± 0.039
8	1117.09 ± 219.39 ↓	0.39 ± 0.08 ↓	53003.7 ± 4465	0.468 ± 0.091	18567.03 ± 2055	0.315 ± 0.028

nal composition from each run were re-normalised. This was done by choosing normalisation bounds for each QoS attribute from the set of solutions produced by each approach. Wilcoxon rank-sum tests with a 0.05 significance level were conducted to detect statistically significant differences. Results show that the proposed GP approach required significantly less time to execute than the others for all datasets except Dataset 6, where runs did not conclude even after several hours. Results for those runs are not available. The GP approach produces solutions with significantly lower quality than at least one other approach for each dataset, though its fitness is often not the lowest out of the three.

5 Conclusions

This paper introduced an automated QoS-aware Web service composition approach that combines graph databases and GP to produce service composi-

tion solutions. Experiments were conducted to evaluate the proposed approach against two existing methods, one that purely employs a graph database and another that purely employs an evolutionary computation technique. Results show that the GP approach requires less time to execute, though it does not always match the quality of the solutions produced by other methods. Thus, future work should investigate improvements to the GP component of this approach, in particular the genetic operators, to produce higher fitness solutions.

References

1. van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Web service composition languages: Old wine in new bottles? In: 29th EUROMICRO Conference 2003, New Waves in System Architecture, 3-5 September 2003, Belek-Antalya, Turkey. pp. 298–307 (2003)
2. Bansal, A., Blake, M.B., Kona, S., Bleul, S., Weise, T., Jaeger, M.C.: WSC-08: continuing the Web services challenge. In: E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on. pp. 351–354. IEEE (2008)
3. Hashemian, S.V., Mavaddat, F.: A graph-based approach to Web services composition. In: 2005 IEEE/IPSJ International Symposium on Applications and the Internet (SAINT 2005), 31 January - 4 February 2005, Trento, Italy. pp. 183–189 (2005)
4. Lerina Aversano, M.D., Taneja, K.: A genetic programming approach to support the design of service compositions. In: First International Workshop on Engineering Service Compositions (WESC' 05) (December 2005)
5. Li, J., Yan, Y., Lemire, D.: Full solution indexing using database for QoS-aware Web service composition. In: IEEE International Conference on Services Computing (SCC), June 27 - July 2, 2014, Anchorage, AK, USA. pp. 99–106 (2014)
6. Milanovic, N., Malek, M.: Current solutions for Web service composition. IEEE Internet Computing 8(6), 51–59 (2004)
7. Ni, J., Zhao, X., Zhu, L.: A semantic Web service-oriented architecture for enterprises. In: International Conference on Research and Practical Issues of Enterprise Information Systems (CONFENIS), October 14-16, 2007, Beijing, China. pp. 535–544 (2007)
8. da Silva, A.S., Ma, H., Zhang, M.: Graphevol: A graph evolution technique for Web service composition. In: 26th International Conference on Database and Expert Systems Applications (DEXA), September 1-4, 2015, Valencia, Spain. pp. 134–142 (2015)
9. da Silva, A.S., Ma, H., Zhang, M.: Genetic programming for QoS-aware Web service composition and selection. Soft Computing 20(10), 3851–3867 (2016)
10. Srivastava, B., Koehler, J.: Web service composition - current solutions and open problems. In: In: ICAPS 2003 Workshop on Planning for Web Services. pp. 28–35 (2003)
11. Yu, Y., Ma, H., Zhang, M.: An adaptive genetic programming approach to QoS-aware web services composition. In: IEEE Congress on Evolutionary Computation (CEC), June 20-23, 2013, Cancun, Mexico. pp. 1740–1747 (2013)
12. Zhang, Z., Ma, H.: Using Graph Databases for Automatic QoS-Aware Web Service Composition. Technical report ECSTR 16-07, Victoria University of Wellington (2016), `http://ecs.victoria.ac.nz/foswiki/pub/Main/TechnicalReportSeries/ECSTR16-07.pdf`