GraphEvol: A Graph Evolution Technique for Web Service Composition

Alexandre Sawczuk da Silva, Hui Ma, Mengjie Zhang

School of Engineering and Computer Science,
Victoria University of Wellington, New Zealand
{Alexandre.Sawczuk.Da.Silva, Hui.Ma, Mengjie.Zhang}@ecs.vuw.ac.nz

Abstract. Web service composition can be thought of as the combination of reusable functionality modules available over the network to create applications that accomplish more complex tasks. Evolutionary Computation (EC) techniques have been applied with success to the problem of fully automated Web service composition, which is when candidate services are selected at the same time that the best configuration in which to connect those candidates is determined. Genetic Programming (GP) is the EC technique traditionally employed in fully automated Web service composition, with solutions encoded as trees instead of their natural Directed Acyclic Graph (DAG) form. The problem with a tree representation is that it complicates the enforcement of dependencies between service nodes, which is much easier accomplish in a DAG. This motivates the proposal of GraphEvol, an evolutionary technique that uses DAGs directly to represent and evolve Web service composition solutions. GraphEvol is analogous to GP, but it implements the mutation and crossover operators differently. Experiments were carried out comparing GraphEvol with GP for a series of composition tasks, with results showing that GraphEvol solutions either match or surpass the quality of those obtained using GP, at the same time relying on a more intuitive representation.

1 Introduction

A Web service can be defined as a software module that accomplishes a specific task and that is made available for requests over the Internet [9]. The fundamental benefit of such modules is that they can be interwoven with new applications, preventing developers from rewriting functionality that has already been implemented. Service-Oriented Architecture (SOA) is a paradigm that expands on this idea, advocating that the main atomic components of a software system should be Web services, since this maximizes code reuse and information sharing [17,6]. As services are typically made available through standard interfaces, the possibility arises to create approaches capable of combining them automatically according to the final desired system, in a process known as *Web service composition* [16]. The objective of these approaches is to produce a workflow, i.e. a directed acyclic graph (DAG), stipulating the sequence in which each atomic service should be executed, as well as the output-input connections between services. Many approaches to Web service composition have been proposed in the literature, from variations on AI planning techniques [8,4] to the employment of integer linear programming solvers [1,26]. In particular, promising results have been achieved with the use of Evolutionary Computation (EC) techniques [24,22], because their search methods successfully handle the large search space characteristic of the composition problem.

Compositions can be accomplished using a variety of techniques such as Genetic Algorithms (GA) and Particle Swarm Optimisation (PSO) [14,24,25] when a general service workflow has already been provided, however *fully automated composition* (no pre-selected workflows) in EC has traditionally been restricted to techniques employing the traditional Genetic Programming (GP) model [19]. In this paradigm, each composition candidate is a tree that represents an underlying graph solution, making it difficult and computationally costly to check and impose the dependencies between nodes. Because of this issue, the question arises whether it is possible to represent a candidate directly as a graph, thus reducing the complexity associated with the verification and enforcement of node dependencies. The objective of this paper is to present and analyse *GraphEvol*, an evolutionary computation technique for Web service composition where each candidate is represented as a DAG and modified while remaining in that form. The main advantage of GraphEvol is that it represents each solution in an intuitive and direct way, allowing for a more powerful way of ensuring that solutions meet correctness constraints.

The remainder of this paper is organised as follows: Section 2 contains background information concerning Web service composition; Section 3 presents the proposed technique; Section 4 discusses the experiments comparing the performance GraphEvol to that of a GP approach; Section 5 describes the experiment results; Section 6 concludes the paper.

2 Background

2.1 The Web Service Composition Problem

In a standard composition problem, a user would like to obtain a composite Web service with a particular functionality. The user makes a request to a Web service composition system specifying the *inputs* the composite service should accept, the *outputs* it should produce, the *constraints* it should consider (e.g. the resulting composite service should have the lowest possible execution time), and the *repository* from which it should select the atomic services to include in the composition. Given this information, the composition system *discovers* relevant services from the repository, and uses these services as potential candidates to be included in the final composition. A composition algorithm is run to *select* appropriate services at the same time it *builds* a workflow denoting how each service in the composition interacts with the others with regards to their required input and produced output.

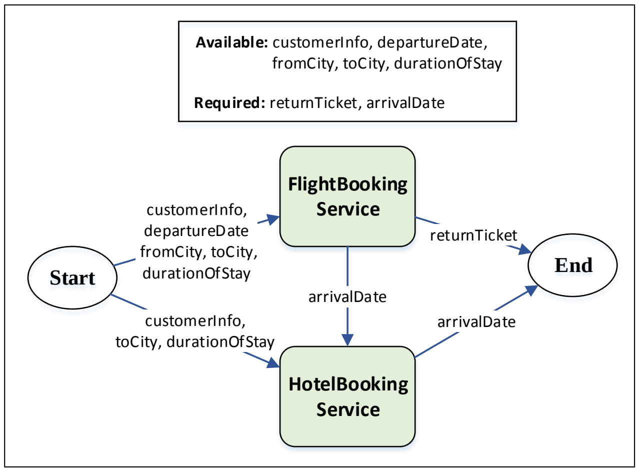

Available: customerInfo, departureDate, fromCity, toCity, durationOfStay
Required: returnTicket, arrivalDate
FlightBooking Service
customerInfo, departureDate fromCity, toCity, durationOfStay
returnTicket
Start
End
arrivalDate
arrivalDate
customerInfo, toCity, durationOfStay
HotelBooking Service

Fig. 1. Example of a solution to a Web service composition task.

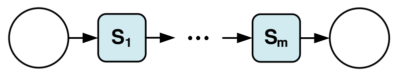
A classic example of automated Web service composition is the travel planning scenario [21], where the objective is to create a system capable of automatically reserving hotels and flights according to customer preferences. In this scenario the customer preference types, such as departure date and destination city, are the composition *inputs*, and the reservation outcomes, such as issued tickets and receipts, are the composition *outputs*. The relevant composition candidates are a set of hotel and flight booking services that are to be combined into a cohesive task workflow by the composition system. Figure 1 shows a simple composition solution that performs flight and hotel reservations according to a customer's information. More specifically, when using this composite service the customer provides her/his personal information and travel details, such as the departure date, the destination city and the duration of the stay. This information is then used to book return flight tickets, and to determine the customer's arrival date at the destination city.

2.2 Quality of Service (QoS) and Composition Constructs

However, it is not enough to simply create compositions whose component services are purely selected based on their inputs and outputs. If a service takes too long to produce a response, for example, then it should not be selected for the composition if there is an equivalent service that is faster. This means that it is necessary to pay attention to the quality properties of each service selected as part of the composition, i.e. a QoS-aware composition approach is needed. There exist many Web service quality properties, from security levels to service throughput [15]. In previous works [11,27] four of them are considered: the probability of a service being available (A), the probability of a service finishing execution within the the expected time limits (R), the expected service time limit between sending a request to the service and receiving a response (T), and the execution cost to be paid by the service requestor (C). The higher the probabilities of a service being available and of it being responsive, the higher its quality with regard to A and R; conversely, the services with the lowest response time and execution cost have the highest quality with regard to T and C. Note that different QoS properties could just as easily have been included.

Web service composition languages such as OWL-S and BPEL4WS recognise the four basic flow constructs for representing the way in which services interact: sequence, choice, parallel and loop [27]. There exist approaches in which the DAG representation includes all four constructs, which means that the composition is in fact a critical path within the graph [28]. In this work, however, only the sequence and parallel constructs are considered, and as a result the entire graph represents a composition. These two constructs are described as follows [27,5]:

Sequence construct The component services of a sequence construct are executed in order, according to the edge flow. This makes the total time (T) and cost (C) of the sequence the sum of the value of those properties in each component service. As the availability (A) and reliability (R) of the sequence represent probabilities, they can be obtained by multiplying the value of those properties in each component service. This construct is shown in Figure 2.



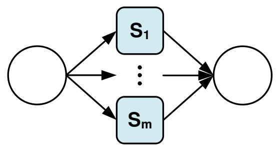
$$T = \sum_{n=1}^{m} t_n \qquad C = \sum_{n=1}^{m} c_n \qquad A = \prod_{n=1}^{m} a_n \qquad R = \prod_{n=1}^{m} r_n$$

Fig. 2. Sequence construct and calculation of its QoS properties [27].

Parallel construct The components of a parallel construct are all executed simultaneously, since their incoming edges originate in a common node and their outgoing edges also converge to a common node. All QoS properties are calculated as for the sequence construct, except for the total time (T), which corresponds to that of the component service with the highest time. This construct is shown in Figure 3.

2.3 Related Work

There are many publications on the subject of EC applied to Web service composition, but this subsection will focus on those which perform fully automated composition, since their results are analogous to those of GraphEvol. One of the pioneering GP composition approaches [2] uses workflow constructs as the non-terminal tree nodes and Web service candidates as the terminal nodes, where



$$T = MAX\{t_n | n \in \{1, \ldots, m\}\}$$

$$C = \sum_{n=1}^{m} c_n \qquad A = \prod_{n=1}^{m} a_n \qquad R = \prod_{n=1}^{m} r_n$$

Fig. 3. Parallel construct and calculation of its QoS properties [27].

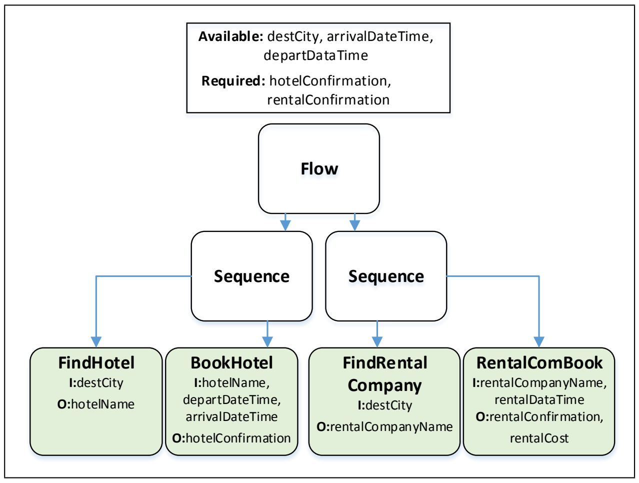

Available: destCity, arrivalDateTime, departDataTime
Required: hotelConfirmation, rentalConfirmation
Flow
Sequence
Sequence
FindHotel
I:destCity
O:hotelName
BookHotel
I:hotelName, departDateTime, arrivalDateTime
O:hotelConfirmation
FindRental Company
I:destCity
O:rentalCompanyName
RentalComBook
I:rentalCompanyName, rentalDataTime
O:rentalConfirmation, rentalCost

Fig. 4. Example of a typical GP candidate composition tree. Adapted from [2].

workflow constructs represent the output-input connections between two services. For example, if two services are sequentially connected, so that output of service A is used as the input of service B, this would be represented by a *sequence* workflow construct having A as the left child and B as the right one. An example of a tree generated by this technique is shown in Figure 4, adapted from an illustration in the referenced work. The initial population is created randomly, which means that the initial compositions represented in that generation are very unlikely to be executable, since their inputs and outputs are mismatched. The aim of the technique is to improve how well these output-input connections match (i.e. how many of the inputs of each Web service can be fulfilled) by employing a fitness function that calculates their correspondence and awards higher fitness scores to those candidates with larger output-input overlaps. The genetic operators employed for this evolutionary process are crossover, where two subtrees from two individuals are randomly selected and swapped, and mutation, where a subtree for an individual is replaced with a randomly generated substitute.

Another GP approach [19] follows a similar tree representation to the one discussed above, as well as a similar implementation of mutation and crossover operators. The greatest distinction between this approach and others is in its use of a context-free grammar to generate the initial population of candidates and to create the subtrees used during mutation. The grammar restricts the tree structure configurations allowed in the tree, thus reducing the search space considered when exploring possible solutions. Another key difference of this approach is in its fitness function, which not only minimises the output-input mismatches in candidates, but also minimises the overall number of atomic services in the composition and the size of the largest sequential chain of such services. These two additions to the fitness function are aimed at reducing the overall execution time and cost of the resulting compositions. This technique was tested with two well-known semantically-annotated datasets, OWL-S TC [13] and WSC 2008 [3], showing that it yields valid solutions to all composition tasks while requiring a relatively low amount of execution time.

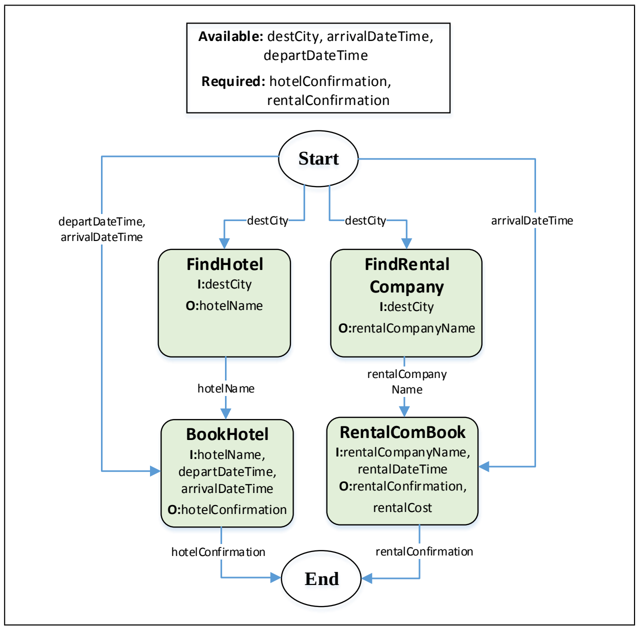

Available: destCity, arrivalDateTime, departDateTime
Required: hotelConfirmation, rentalConfirmation
Start
departDateTime, arrivalDateTime
destCity
destCity
arrivalDateTime
FindHotel
I:destCity
O:hotelName
FindRental Company
I:destCity
O:rentalCompanyName
hotelName
rentalCompany Name
BookHotel
I:hotelName, departDateTime, arrivalDateTime
O:hotelConfirmation
RentalComBook
I:rentalCompanyName, rentalDateTime
O:rentalConfirmation, rentalCost
hotelConfirmation
rentalConfirmation
End

Fig. 5. Example of a candidate composition in GraphEvol.

In addition to GP, an approach using PSO has also been shown to be a suitable method for fully automated Web service composition [20]. In this technique, candidates encoded as a vectors of weights (particles), each ranging from 0 to 1. These weights are mapped to the edges of a *master graph*, which is a data structure created at initialisation time to represent all the possible output-input relationships between relevant candidate services. The central idea of this technique is to utilise a greedy algorithm to extract non-redundant functional solutions from the master graph, using the weights as guides that prioritise the choice of certain edges and nodes of the structure. Since the master graph structure is built ensuring full matching of services inputs and outputs, the fitness function employed for evolution optimises the population solely according to non-functional quality (QoS) measures such as response time and availability.

While the approaches discussed above have merit, they also present some issues that limit their composition power. The biggest disadvantage concerning the two GP approaches is that their initial populations do not have a high degree of output-input matches, meaning that the solutions in their initial populations correspond to compositions that are not fully executable due to a lack of correct service inputs. In the case of [2], even after the evolutionary process some of the solutions produced could not be fully executed. The use of a grammar by [19] does help with this issue, but at the cost of increased execution complexity and repeated tree adjustments. As for the PSO approach, its biggest drawback is that it requires the decoding a solution from the master graph every time its fitness need to be calculated. As the relevant services in the repository grow, so does the master graph, causing the performance of the approach to decrease significantly [20]. These issues could be successfully addressed by representing solutions directly as graphs, since they do not require any form of decoding and can be easily built using algorithms that always generate fully executable compositions. This is the main motivation for the creation of the approach presented in this work.

3 GraphEvol

GraphEvol, an evolutionary computation approach proposed in this work, bears many similarities with the GP approaches discussed above. Namely, it initialises a population of candidates that are encoded using non-linear data structures, evolves this population using crossover and mutation operators, and evaluates the quality of each candidate based on the nodes included in its structure. However, as opposed to representing candidate compositions as trees that correspond to underlying graph structures, GraphEvol represents them directly as graphs with Web service nodes. Figure 5 shows a graph representation example that is equivalent to the candidate tree shown in Figure 4. As a consequence of this direct graph representation, the mutation and crossover operators must be implemented differently, and so must the fitness function. Another important aspect of GraphEvol is that it uses a graph-building algorithm for creating new solutions, and for performing mutation and crossover. The general procedure for GraphEvol is shown in Algorithm 1, however each fundamental aspect of the proposed technique is explored in more detail in the following subsections.

3.1 Graph building algorithm

The initialisation of candidates is performed by employing a graph-building algorithm that is based on the planning graph approach discussed in the composition

Algorithm 1. Steps of the GraphEvol technique.

```
1. Initialise the population using the graph building algorithm.
2. Evaluate the fitness of the initialised population.
while max. generations not met do
    3. Select the fittest graph candidates for reproduction.
    4. Perform mutation and crossover on the selected candidates, generating
    offspring.
    5. Evaluate the fitness of the new graph individuals.
    6. Replace the lowest-fitness individuals in the population with the new
    graph individuals.
```

literature [10,8,23,7]. There are three main differences between the algorithm presented here and the ones previously proposed: it builds the graph forwards (from the start node to the end node) instead of backwards, therefore it naturally avoids the formation of cycles; it does not require building the graph in layers, which simplifies the construction process; it describes the edge configuration when connecting a new node to the graph, which is not shown by the other algorithms.

Algorithm 2 begins by adding the *start* node to the graph, marking it as one of the *seenNodes*, and adding some initial candidates to the *candidateList* to be considered for connection. These candidates are identified using the *findCands* function, which discovers which elements from the *relevant* set have at least some of their input satisfied by the nodes already in the graph (i.e. the *seenNodes*). Then the building process begins, continuing as long as the composition outputs represented in *end.inputs* have not been fulfilled by the currently available graph outputs in *currentEndInputs*. In this process, a candidate *cand* is selected at random from the *candidateList*. If it has not already been used in the graph and all of its inputs can be fulfilled by the currently available graph outputs, then it is connected to the graph using the *connectNode* function. This function identifies a random, minimal set of edges connecting the new node (n) to already existing nodes in the graph so that the inputs of this new node are fully satisfied.

After connecting *cand* to the graph, it is added to the set of *seenNodes* and the *candidateList* is updated to include services that may be fulfilled by the outputs of *cand*. Finally, once the composition's required output has been reached, the *end* node is connected. This particular graph building algorithm often results in *dangling nodes*, which are chains of nodes that are connected to the graph but whose output is not used to fulfil any other nodes. Because of this, a routine to remove such chains (*removeDangling*) is executed on the graph before the completed structure is returned. It is important to highlight that the selection of each candidate to connected with the graph, as well as the edges that should be used in this connection, is done stochastically, meaning that the resulting graph varies with each run of the algorithm.

ALGORITHM 2. Generating a new candidate graph.

```
    Input  : $I$, $O$, $relevant$
    Output: candidate graph $G$
1:  $start.outputs \leftarrow \{I\}$;
2:  $end.inputs \leftarrow \{O\}$;
3:  $G.edges \leftarrow \{\}$;
4:  $G.nodes \leftarrow \{start\}$;
5:  $seenNodes \leftarrow \{start\}$;
6:  $currEndInputs \leftarrow \{start.outputs\}$;
7:  $candidateList \leftarrow$ findCands($seenNodes, relevant$);
8:  while $end.inputs \not\sqsubseteq currentEndInputs$ do
9:      $cand \leftarrow candidateList.next()$;
10:     if $cand \notin seenNodes \wedge cand.inputs \sqsubseteq currEndInputs$ then
11:         connectNode($cand, G$);
12:         $currEndInputs \leftarrow currEndInputs \cup \{cand.outputs\}$;
13:         $seenNodes \leftarrow seenNodes \cup \{cand\}$;
14:         $candidateList \leftarrow candidateList \cup$ findCands($seenNodes, relevant$);
15: connectNode($end, G$);
16: removeDangling($G$);
17: return $G$;
18: Procedure connectNode($n, G$)
19:     $G.nodes \leftarrow G.nodes \cup \{n\}$;
20:     $inputsToFulfil \leftarrow \{cand.inputs\}$;
21:     while $|inputsToFulfil| > 0$ do
22:         $graphN \leftarrow G.nodes.next()$;
23:         if $|n.inputs \sqcap graphN.outputs| > 0$ then
24:             $inputsToFulfil \leftarrow$
                $inputsToFulfil - (n.inputs \sqcap graphN.outputs)$;
25:             $G.edges \leftarrow G.edges \cup \{graphN \rightarrow n\}$;
```

3.2 Mutation

Intuitively, the mutation operator for a graph candidate implements the same idea of the corresponding tree operator [2]: a subpart of the candidate should be removed and replaced with a new randomly generated fragment, while maintaining the functional properties of the original subpart (i.e. correct output-input matches where the subpart connects with the main part of the candidate). Accomplishing this in a tree representation is quite straightforward, since the only point of dependency between the subtree to be mutated and the overall tree is the selected root node. For a graph, on the other hand, multiple dependency points require more careful consideration. To perform a graph mutation, we begin by randomly selecting a node in the graph (excluding the end node) to act as the 'root' of the subpart to be replaced. Subsequently, all nodes that are directly or indirectly dependent on the outputs of this root are also identified, all the way to the end node, as shown in Figure 6. These nodes are removed from the graph, and all of its connections (edges) to the main part of the graph are severed. Finally, the incomplete graph is fed into Algorithm 2, but beginning execution from line 8. This completes the graph and results in an offspring with the same main part as its parent, but with a distinct subpart.

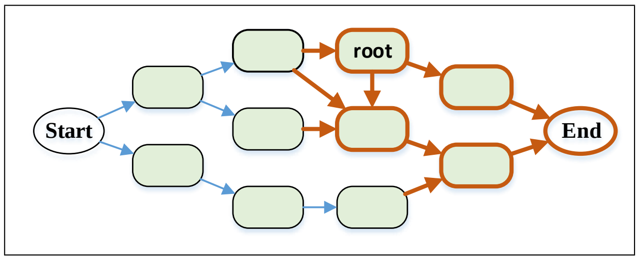


Fig. 6. Example of nodes and edges marked for deletion during the graph mutation operation.

3.3 Controlled Mutation

The mutation operator described above replaces graph subparts of varying sizes, which means that sometimes the modification to a candidate may be too drastic. Such a radical structural change may be undesirable, as it does not allow for the discovery of small improvements to an existing candidate configuration. This issue motivated the creation of a controlled mutation operator which selects a fixed-size graph subpart for replacement, under the hypothesis that this may lead to solutions with a better quality overall. To perform a controlled graph mutation of size n, we begin by randomly selecting a root node and adding it to subgraph S_1. Subsequently, $n-1$ nodes are also selected, provided that the ancestors to each of these nodes are all part of S_1 and the node is not the end node, as shown in Figure 7. S_1 is removed from the graph and all of its connections to the graph are severed, keeping track of all incoming connections from the graph to S_1, and all outgoing connections from S_1 to the graph. The inputs and outputs of the incoming and outgoing connections are then used to generate a new subgraph S_2 to replace S_1, and S_2 is reconnected to the graph using the necessary incoming and outgoing connections. This results in a new candidate that preserves most of the original structure but also includes a controlled modification.

3.4 Crossover

Traditionally, the crossover operation involves the exchange of genetic material by two candidates in order to produce offspring with characteristics from both of its parents, thus encouraging further improvements to solutions that may

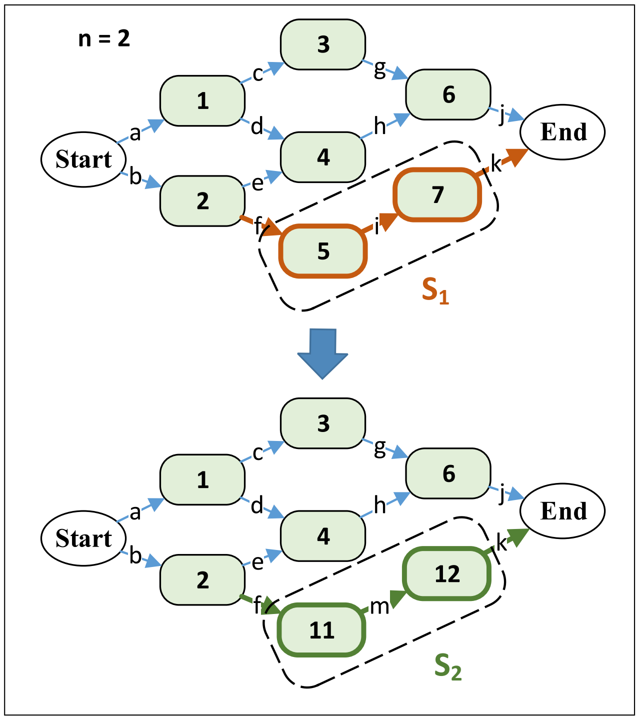


Fig. 7. Example of a controlled mutation with $n = 2$.

already possess a certain degree of maturity [18]. In GP this exchange can be done simply by swapping two subtrees of two distinct candidates [2], however in the GraphEvol context doing so would affect dependencies throughout the graph by compromising the correctness of the connections between the outputs and inputs of service nodes. Therefore, the idea of *merging* and *extracting* graphs has been employed in the implementation of this operator. The basic intuition is to select two candidate graphs, merge them into a single structure, and then extract a new candidate out of this merged structure. This strategy restricts the offspring to utilise structures already present on either parent, thus achieving a similar outcome as traditional crossover operations.

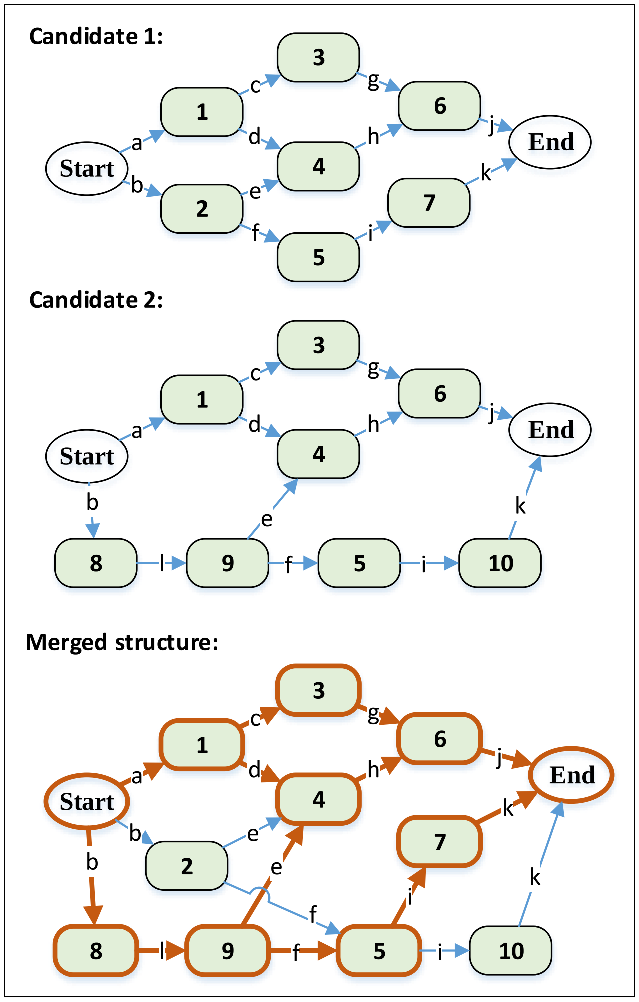


Fig. 8. Example of the merge and extraction process used in the graph crossover operation.

The merging process is depicted in Figure 8, and consists of combining any two nodes that represent the same service into a single node, maintaining all original dependencies from both graphs but resulting in the presence of redundant nodes and edges. Once the merge has taken place, an offspring can be extracted from the structure to obtain a new non-redundant solution. A modified version of Algorithm 2 is used for this task, where instead of adding candidates to the *candidateList* from the entire set of *relevant* nodes, only nodes from the merged structure can be considered. In particular, whenever a node is added to the extracted solution graph, only the service nodes it connects to (through an outgoing edge) in the merged graph structure are added to the *candidateList*.

Figure 8 highlights one of the possible solutions that could be extracted from the merged structure.

3.5 Structure-based Fitness function

One of the fitness functions considered in the evolutionary process is based on the one presented in [19]. In the original paper, the fitness function also presents a third criterion measuring the degree of input satisfaction for each service node in the solution. In our case, however, this is not necessary, since the graph building algorithm used throughout the evolution process ensures that the inputs of each service included in a solution are always fully satisfied.

The function seeks to produce solutions with the smallest possible number of service nodes and with the shortest possible paths from the start node to the end node. The rationale behind this decision is that it encourages features indicative of the quality of the overall composition [19]: the number of service nodes indicates the complexity of the overall composition (it should be as small as possible); the length of the longest path from the start node to the end node indicates the execution time of the composition solution, with a shorter path indicating more parallelisation of tasks and consequently a lower execution time. The fitness function is formally described as follows:

$$fitness_i = \omega_1 \cdot \frac{1}{runPath_i} + \omega_2 \cdot \frac{1}{\#atomicService_i} \tag{1}$$

where $\omega_1 + \omega_2 = 1$, $runPath_i$ is the longest path from the start node to the end node of a solution i (measured using a longest path algorithm), and $\#atomicService_i$ is the total number of service nodes included in a solution i.

3.6 QoS-based Fitness Function

The other fitness function considered optimises candidates according to their overall QoS values, taking into account the four properties introduced in subsection 2.2. It shares common elements with the work of Zeng et al. [29], for example, but is fundamentally different in which it does not maximise or minimise values by placing them as numerators or denominators of a fraction. The QoS properties are combined into the following function for a particle i:

$$fitness_i = w_1 A_i + w_2 R_i + w_3(1 - T_i) + w_4(1 - C_i) \tag{2}$$

where $\sum_{i=1}^{4} w_i = 1$

A, C, and R are calculated using all services in the particle according to the formulae presented in Figures 2 and 3. T is determined by adding the times of the longest path in the workflow, from start to end. By identifying the longest path it is possible to correctly calculate time both in the case of parallel constructs and of sequence constructs.

The output of the fitness function is within the range [0, 1], with 1 representing the best possible fitness and 0 representing the worst. The fitness function weights add to 1, so the values of A, C, R and T must all be normalised between 0 and 1 to ensure that the function produces results in the required range. This is done by identifying the minimum and maximum values in the dataset for each QoS attribute, and using the following formula:

$$normalise(value) = \begin{cases} \frac{value - min}{max - min} & \text{if } max - min \neq 0. \\ 1 & \text{otherwise.} \end{cases} \quad (3)$$

4 Experiment Design

Two sets of experiments were carried out. The first set compared the performance of GraphEvol against the traditional GP approach presented in [19], seeking to validate the hypothesis that the total number of service nodes and longest path in the resulting compositions will be similar or smaller to those produced by GP. The validation of this hypothesis would indicate that GraphEvol is a suitable alternative to Web service composition using GP, matching the effectiveness of the latter while at the same time providing the benefits of a direct solution representation. The authors who introduced the GP approach also presented experimental results of that method's application to two datasets using a variety of composition tasks, thus those results will be used as the basis of this comparison. The structure-based fitness function was used for these experiments. The second set of experiments compared the performance of GraphEvol when using the uncontrolled mutation operator against GraphEvol using the controlled mutation operator, to validate the hypothesis that the controlled operator will lead to solutions with better overall quality. As the focus of this set of experiments was on the quality of the services, the QoS-based fitness function was employed.

4.1 Datasets and Parameters

Both sets of experiments were conducted using a personal computer with an Intel Core i7-4770 CPU (3.4 GHz), and 8 GB RAM. The datasets employed in the first set of experiments were OWL-S TC V2.2 [13] and WSC 2008 [3], both of which present service collections of varying sizes. Tasks 1–5, which are outlined in [19], were used when to test GraphEvol with the OWL-S TC dataset; tasks WSC 2008-1, WSC 2008-2, and WSC 2008-5 were used for testing with the WSC 2008 dataset. To match the GP approach, a population of 200 candidates was evolved during 20 generations for each composition task, and this process was repeated over 30 independent runs. The fitness function weights ω_1 and ω_2 were both set to 0.5, the mutation probability to 0.05, and the crossover probability to 0.5. Individuals were chosen for breeding using tournament selection with a tournament size of 2, mirroring the design of the experiment conducted by the authors of the GP approach [19]. The second set of experiments used the datasets WSC 2008 and WSC 2009, taking into account the QoS information from the

	GraphEvol			**GP Approach**		
Task	**Time (ms)**	**runPath**	**#atomicService**	**Time (ms)**	**runPath**	**#atomicService**
OWL-S TC-1	469.60 ± 108.10	1.00 ± 0.00	1.00 ± 0.00	749.00 ± 364.10	1.00 ± 0.00	1.00 ± 0.00
OWL-S TC-2	326.73 ± 29.07	$1.00 \pm 0.00 \downarrow$	$1.00 \pm 0.00 \downarrow$	484.50 ± 139.20	2.00 ± 0.00	2.00 ± 0.00
OWL-S TC-3	517.17 ± 99.98	2.00 ± 0.00	2.00 ± 0.00	473.60 ± 76.19	2.00 ± 0.00	2.00 ± 0.00
OWL-S TC-4	674.23 ± 107.50	$2.00 \pm 0.00 \downarrow$	$4.00 \pm 0.00 \downarrow$	3010.20 ± 422.91	2.20 ± 0.40	5.70 ± 1.19
OWL-S TC-5	472.23 ± 46.33	1.00 ± 0.00	$3.00 \pm 0.00 \downarrow$	1098.30 ± 240.72	1.00 ± 0.00	3.30 ± 0.46
WSC 2008-1	699.43 ± 93.79	$3.00 \pm 0.00 \downarrow$	$10.00 \pm 0.00 \downarrow$	6919.70 ± 1612.99	6.00 ± 1.26	15.8 ± 5.71
WSC 2008-2	734.63 ± 102.84	$3.00 \pm 0.00 \downarrow$	$5.00 \pm 0.00 \downarrow$	11137.20 ± 3106.75	3.50 ± 0.67	6.00 ± 0.89
WSC 2008-5	918.40 ± 120.13	$8.00 \pm 0.00 \downarrow$	$20.00 \pm 0.00 \downarrow$	95390.20 ± 43521.30	9.20 ± 2.96	49.90 ± 16.84

Table 1. Mean execution times, longest path lengths, and number of service nodes for each task in GraphEvol and GP [19], for the first set of experiments.

services. A population of 500 candidates was evolved during 51 generations for each composition task, and this process was repeated over 30 independent runs. The fitness function weights were all set to 0.25, the mutation probability to 0.1, the crossover probability to 0.8, and the reproduction probability to 0.1. Individuals were chosen for breeding using tournament selection with a tournament size of 2. These parameters are based on those proposed in [12].

5 Results

Results for the first set of experiments are presented in columns 1, 2, and 3 of Table 1, side by side with the previously published GP results. Column 1 displays the mean execution time for GraphEvol in milliseconds; column 2 presents the mean length of the longest path in a run's best solution; column 3 shows the mean number of service nodes included in a run's best solution. Means are accompanied by their respective standard deviations, and all values in the table are rounded to 2 decimal points of precision. While the two approaches were executed in different platforms, meaning that it is not possible to perform a direct comparison on execution time, the large time gap when executing tasks OWL-S TC-4, WSC 2008-1, WSC 2008-2, and WSC-2008-3 provides strong evidence that the GraphEvol approach can handle certain service collections more effectively than the GP approach. For the other tasks, on the other hand, there is no obvious time difference.

Since both approaches were tested using the same datasets and tasks, as well as employing equivalent fitness functions during the evolutionary process, it is possible to perform a direct comparison on the longest path lengths and the overall number of nodes of the solutions produced. Unpaired t-tests at 0.05 significance level were conducted to verify whether there are statistically significant differences between the results produced by each technique. Such differences are denoted using ↑ for significantly higher results and ↓ for significantly lower. The tests revealed that GraphEvol yielded solutions with equivalent or significantly smaller longest paths and numbers of nodes, that is, the quality of the solutions produced by GraphEvol always matched or surpassed that of the solutions produced by GP. These results establish the GraphEvol approach as a powerful alternative when performing fully automated Web service composition.

	Uncontrolled		**Controlled**	
Task	**Time (ms)**	**Fitness**	**Time (ms)**	**Fitness**
WSC 2008-1	2006.70 ± 242.47	0.47 ± 0.00	$1667.07 \pm 231.66 \downarrow$	0.47 ± 0.00
WSC 2008-2	1326.10 ± 192.96	0.58 ± 0.00	1164.33 ± 123.72	0.58 ± 0.00
WSC 2008-3	8505.43 ± 805.25	0.43 ± 0.00	$5648.67 \pm 572.83 \downarrow$	0.43 ± 0.00
WSC 2008-4	2345.47 ± 296.85	0.46 ± 0.00	$1925.07 \pm 143.37 \downarrow$	0.47 ± 0.00
WSC 2008-5	4496.73 ± 491.43	0.47 ± 0.00	$3580.43 \pm 414.06 \downarrow$	0.47 ± 0.00
WSC 2008-6	11286.10 ± 1060.36	0.47 ± 0.00	$8208.17 \pm 413.18 \downarrow$	0.47 ± 0.00
WSC 2008-7	9182.80 ± 715.37	0.48 ± 0.00	$5892.13 \pm 416.96 \downarrow$	0.48 ± 0.00
WSC 2008-8	8708.93 ± 630.56	0.46 ± 0.00	$6117.77 \pm 379.25 \downarrow$	0.46 ± 0.00
WSC 2009-1	1609.30 ± 163.21	0.55 ± 0.01	1502.10 ± 200.56	0.55 ± 0.01
WSC 2009-2	5370.43 ± 649.83	0.48 ± 0.00	$3983.37 \pm 367.27 \downarrow$	0.48 ± 0.00
WSC 2009-3	3229.33 ± 337.36	0.49 ± 0.00	$2883.80 \pm 209.76 \downarrow$	0.49 ± 0.00
WSC 2009-4	17778.80 ± 1772.65	0.48 ± 0.00	$12842.03 \pm 871.14 \downarrow$	0.48 ± 0.00
WSC 2009-5	13046.97 ± 1482.35	0.47 ± 0.00	$7141.00 \pm 599.37 \downarrow$	0.47 ± 0.00

Table 2. Mean execution time and mean fitness for the second set of experiments.

Results for the second set of experiments are shown in Table 2. Columns 1 and 2 show the mean execution time and the mean best fitness for GraphEvol with the uncontrolled mutation operator, both columns including standard deviations; columns 3 and 4 show the mean time and fitness for GraphEvol with controlled mutation. As the results show, the hypothesis that using a controlled mutation operator would lead to compositions with better overall quality has not been confirmed, given that the fitness values produced by GraphEvol are similar when using uncontrolled an controlled mutation operators. However, the execution time results show a clear pattern: the time required for executing

GraphEvol with uncontrolled mutation is significantly lower for all composition tasks, with the exception of WSC 2008-2 and WSC 2009-1. This is likely because the complexity associated with replacing a small subgraph in a candidate, as done by the controlled mutation operator, is lower than that of the larger subgraphs selected by the uncontrolled mutation.

6 Conclusions

This work presented GraphEvol, an evolutionary computation technique aimed at performing fully automated Web service composition using graph representations for solutions, as opposed to encoding them into tree or vector representations. A graph building algorithm was proposed for generating the initial population, and variations of it were employed during the evolutionary process. The traditional mutation and crossover operations were modified to work with graph candidates, involving graph merging and traversal procedures, and a controlled mutation operator was also designed. Finally, experiments were conducted comparing GraphEvol to an analogous GP approach, as well as comparing the two GraphEvol mutation operators. Results showed that the quality of the results produced by GraphEvol always matched or surpassed those produced by GP, and that the controlled mutation operator leads to lower execution times for GraphEvol.

References

1. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. Software Engineering, IEEE Transactions on 33(6), 369–384 (2007)
2. Aversano, L., Di Penta, M., Taneja, K.: A genetic programming approach to support the design of service compositions. International Journal of Computer Systems Science & Engineering 21(4), 247–254 (2006)
3. 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)
4. Bucchiarone, A., De Sanctis, M., Pistore, M.: Domain objects for dynamic and incremental service composition. In: Service-Oriented and Cloud Computing, pp. 62–80. Springer (2014)
5. Cardoso, J., Sheth, A., Miller, J., Arnold, J., Kochut, K.: Quality of service for workflows and web service processes. Web Semantics: Science, Services and Agents on the World Wide Web 1(3), 281 – 308 (2004), http://www.sciencedirect.com/science/article/pii/S157082680400006X
6. Channabasavaiah, K., Holley, K., Tuggle, E.: Migrating to a service-oriented architecture. IBM DeveloperWorks 16 (2003)
7. Chen, M., Yan, Y.: Qos-aware service composition over graphplan through graph reachability. In: Services Computing (SCC), 2014 IEEE International Conference on. pp. 544–551. IEEE (2014)
8. Deng, S., Wu, B., Yin, J., Wu, Z.: Efficient planning for top-k web service composition. Knowledge and information systems 36(3), 579–605 (2013)

9. Gottschalk, K., Graham, S., Kreger, H., Snell, J.: Introduction to web services architecture. IBM systems Journal 41(2), 170–177 (2002)
10. Huang, Z., Jiang, W., Hu, S., Liu, Z.: Effective pruning algorithm for qos-aware service composition. In: Commerce and Enterprise Computing, 2009. CEC'09. IEEE Conference on. pp. 519–522. IEEE (2009)
11. Jaeger, M.C., Mühl, G.: Qos-based selection of services: The implementation of a genetic algorithm. In: Communication in Distributed Systems (KiVS), 2007 ITG-GI Conference. pp. 1–12. VDE (2007)
12. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)
13. Kuster, U., Konig-Ries, B., Krug, A.: Opossum-an online portal to collect and share sws descriptions. In: Semantic Computing, 2008 IEEE International Conference on. pp. 480–481. IEEE (2008)
14. Liu, S.L., Liu, Y.X., Zhang, F., Tang, G.F., Jing, N.: Dynamic web services selection algorithm with qos global optimal in web services composition. Ruan Jian Xue Bao(Journal of Software) 18(3), 646–656 (2007)
15. Menascé, D.A.: Qos issues in web services. Internet Computing, IEEE 6(6), 72–75 (2002)
16. Milanovic, N., Malek, M.: Current solutions for web service composition. IEEE Internet Computing 8(6), 51–59 (2004)
17. Perrey, R., Lycett, M.: Service-oriented architecture. In: Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on. pp. 116–119. IEEE (2003)
18. Qi, X., Palmieri, F.: Theoretical analysis of evolutionary algorithms with an infinite population size in continuous space. part ii: Analysis of the diversification role of crossover. Neural Networks, IEEE Transactions on 5(1), 120–129 (1994)
19. Rodriguez-Mier, P., Mucientes, M., Lama, M., Couto, M.I.: Composition of web services through genetic programming. Evolutionary Intelligence 3(3-4), 171–186 (2010)
20. da Silva, A., Ma, H., Zhang, M.: A graph-based particle swarm optimisation approach to qos-aware web service composition and selection. In: Evolutionary Computation (CEC), 2014 IEEE Congress on. pp. 3127–3134 (July 2014)
21. Srivastava, B., Koehler, J.: Web service composition-current solutions and open problems. In: ICAPS 2003 workshop on Planning for Web Services. vol. 35, pp. 28–35 (2003)
22. Su, K., Liangli, M., Xiaoming, G., Yufei, S.: An efficient parameter-adaptive genetic algorithm for service selection with end-to-end qos constraints. Journal of Computational Information Systems 10(2), 581–588 (2014)
23. Wang, A., Ma, H., Zhang, M.: Genetic programming with greedy search for web service composition. In: Database and Expert Systems Applications. pp. 9–17. Springer (2013)
24. Wang, L., Shen, J., Yong, J.: A survey on bio-inspired algorithms for web service composition. In: Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on. pp. 569–574. IEEE (2012)
25. Yin, H., Zhang, C., Zhang, B., Guo, Y., Liu, T.: A hybrid multiobjective discrete particle swarm optimization algorithm for a sla-aware service composition problem. Mathematical Problems in Engineering 2014 (2014)
26. Yoo, J.J.W., Kumara, S., Lee, D., Oh, S.C.: A web service composition framework using integer programming with non-functional objectives and constraints. algorithms 1, 7 (2008)

27. Yu, Y., Ma, H., Zhang, M.: An adaptive genetic programming approach to qos-aware web services composition. In: Evolutionary Computation (CEC), 2013 IEEE Congress on. pp. 1740–1747. IEEE (2013)
28. Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proceedings of the 12th international conference on World Wide Web. pp. 411–421. ACM (2003)
29. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: Qos-aware middleware for web services composition. Software Engineering, IEEE Transactions on 30(5), 311–327 (2004)