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. However, all of these techniques encode candidate solutions, which can be naturally thought of as Directed Acyclic Graphs (DAGs), into some other representation, commonly trees. This paper proposes GraphEvol, an evolutionary technique that investigates the direct use of DAGs to represent and evolve Web service composition solutions. GraphEvol is analogous to Genetic Programming (GP), but implements the mutation and crossover operators differently. Experiments were carried out comparing GraphEvol to a GP technique for a series of composition tasks, with results showing that GraphEvol solutions either match or surpass the quality of those obtained using GP, requiring equal or less execution time.

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 [Gottschalk *et al.*, 2002]. 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 [Perrey and Lycett, 2003]. As services are typically present standard interfaces, the possibility arises to create approaches capable of combining services automatically according to the final desired system, in a process known as *Web service composition* [Milanovic and Malek, 2004]. 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.

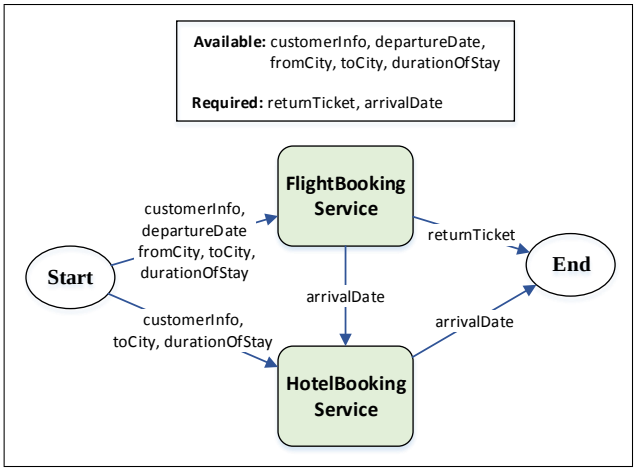

Figure 1: Example of a solution to a Web service composition task.

Many approaches to Web service composition have been proposed in the literature, from variations on AI planning techniques [Deng *et al.*, 2013] to the employment of integer linear programming solvers [Yoo *et al.*, 2008]. In particular, promising results have been achieved with the use of Evolutionary Computation (EC) techniques [Wang *et al.*, 2012], because their search methods successfully handle the large search space characteristic of the composition problem. Existing evolutionary techniques for fully automated Web service composition can be divided into two different groups according to the extent of the composition they perform: *semi-automated composition techniques*, assume that a general workflow of abstract tasks has already been provided and that the algorithm must simply find the best candidate for each task slot; *fully automated composition techniques*, on the other hand, make no such assumption, meaning that they construct the task workflow and select the most suitable candidates simultaneously. Because of their greater capabilities, fully automated composition techniques are the focus of this paper.

While semi-automated composition can be accomplished

using a variety of techniques such as Genetic Algorithms (GA) and Particle Swarm Optimisation (PSO) [da Silva *et al.*, 2014], fully automated composition using EC is currently mostly restricted to techniques employing the traditional Genetic Programming (GP) model [Rodriguez-Mier *et al.*, 2010]. In this paradigm, each each composition candidate is a tree that represents an underlying graph solution that can be obtained through a translation process. The question then arises whether it is possible to represent a candidate directly as a graph, and whether the system's performance is affected from doing so. Thus, 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. Apart from possible performance gains, the main advantage of GraphEvol is that it represents each solution in an intuitive and direct way, allowing a more powerful way of ensuring that solutions meet correctness constraints. The remainder of this paper is organised as follows: Section 2 presents background information concerning Web service composition; Section 3 presents the proposed technique; Section 4 discusses the experiments conducted to compare the performance GraphEvol to that of a GP composition 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.

A classic example of automated Web service composition is the travel planning scenario [Srivastava and Koehler, 2003], 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

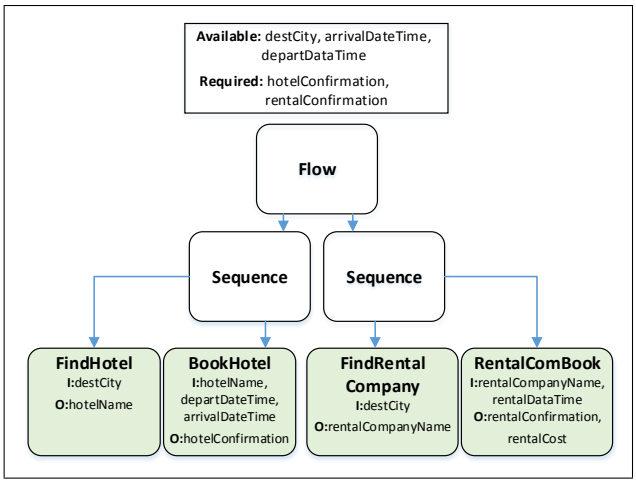

Figure 2: Example of a typical GP candidate composition tree [Aversano *et al.*, 2006].

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 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 composition outcomes are analogous to those of GraphEvol. One of the pioneering GP composition approaches [Aversano *et al.*, 2006] uses workflow constructs as the non-terminal tree nodes and Web service candidates as the terminal nodes, where 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. Likewise, if two services *A* and *B* have independent inputs and outputs and can be executed in parallel, this can be encoded as a *flow* construct having *A* and *B* as its children. An example of a tree generated by this technique is shown in Figure 2, adapted from an illustration in the referenced work. The initial population is initialised 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 these output-input matches by employing a fitness function that calculates their correspondence and awards higher fitness scores to those candidates with larger 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 [Rodriguez-Mier *et al.*, 2010] 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

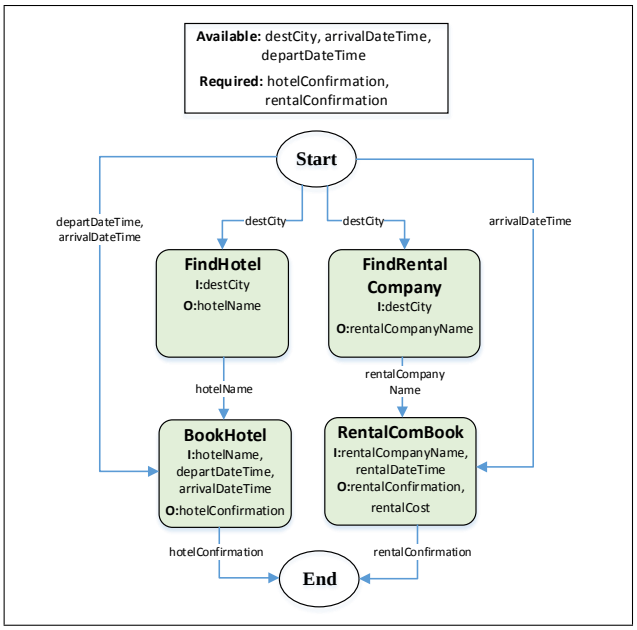

Figure 3: Example of a candidate composition in GraphEvol.

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 searching for a solution. 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 against two well-known semantically-annotated datasets, OWL-S TC [Kuster *et al.*, 2008] and WSC 2008 [Bansal *et al.*, 2008], showing that it yields valid solutions to all composition tasks while requiring a relatively low amount of execution time.

ALGORITHM 1. Steps of the GraphEvol technique.

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

In addition to GP, an approach using PSO has also been shown to be a suitable method for fully automated Web service composition [da Silva *et al.*, 2014]. In this technique,

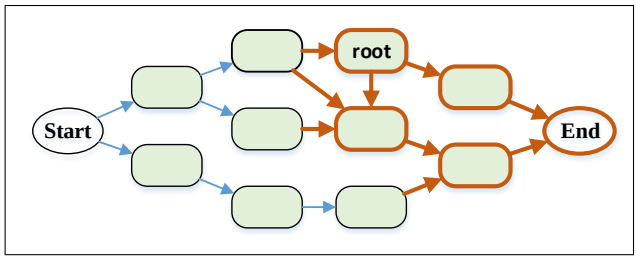

Figure 4: Example of nodes and edges marked for deletion during the graph mutation operation.

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. The weights are evolved using the classical PSO algorithm, and at each generation the fitness of all solutions is calculated by extracting the underlying structure from the master graph. Since the master graph structure is built ensuring full matching of services inputs and outputs, the fitness function employed for evolution optimises the population according to non-functional quality (QoS) measures such as response time, availability, and reliability.

3 GraphEvol

GraphEvol, the evolutionary computation approach proposed in this work, bears many similarities with the GP approaches discussed above. Namely, it initialises and 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 3 shows a graph representation example that is equivalent to the candidate tree shown in Figure 2. 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 is 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 literature [Chen and Yan, 2014; Deng *et al.*, 2013; Huang *et al.*, 2009]. At first,

an algorithm is run to filter all services that will be potentially useful when building a solution, similar to the one presented in [Wang *et al.*, 2013]. These relevant services, along with the composition's desired inputs and outputs, are then presented to Algorithm 2, whose task is to connect some of these nodes and form a non-redundant functionally correct solution (i.e. a solution where all service inputs and composition outputs are fulfilled without any superfluous connections between services).

ALGORITHM 2. Generating a new candidate graph.

```
   Input  : I, O, relevant
   Output: candidate graph G
 1: start.outputs ← {I};
 2: end.inputs ← {O};
 3: G.edges ← {};
 4: G.nodes ← {start};
 5: seenNodes ← {start};
 6: currEndInputs ← {start.outputs};
 7: candidateList ←
    findCands(seenNodes, relevant);
 8: while end.inputs ⋢ currentEndInputs do
 9:     cand ← candidateList.next();
10:     if cand ∉ seenNodes ∧ cand.inputs ⊑
        currEndInputs then
11:         connectNode(cand, G);
12:         currEndInputs ←
            currEndInputs ∪ {cand.outputs};
13:         seenNodes ← seenNodes ∪ {cand};
14:         candidateList ← candidateList∪
            findCands(seenNodes, relevant);
15: connectNode(end, G);
16: removeDangling(G);
17: return G;
18: Procedure connectNode(n, G)
19:     G.nodes ← G.nodes ∪ {n};
20:     inputsToFulfil ← {cand.inputs};
21:     while |inputsToFulfil| > 0 do
22:         graphN ← G.nodes.next();
23:         if |n.inputs ⊓ graphN.outputs| > 0 then
24:             inputsToFulfil ← inputsToFulfil −
                (n.inputs ⊓ graphN.outputs);
25:             G.edges ← G.edges ∪ {graphN → n};
```

Algorithm 2 builds the graph from the *start* to the *end* node, thus preventing the formation of cyclical structures. It 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*, and 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 producing the completed structure. 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.

3.2 Mutation

Intuitively, the mutation operator for a graph candidate implements the same idea of the corresponding tree operator [Aversano *et al.*, 2006]: a subpart of the candidate should be removed and replaced with a new randomly generated fragment, all the 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). To do so, 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 4. 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.

3.3 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 on solutions that may already possess a certain degree of maturity [Qi and Palmieri, 1994]. In GP this exchange can be done simply by swapping two subtrees of two distinct candidates [Aversano *et al.*, 2006], 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. Two candidate graphs are selected for the crossover, and these graphs are then merged into a single structure that maintains the structures of both graphs but as a

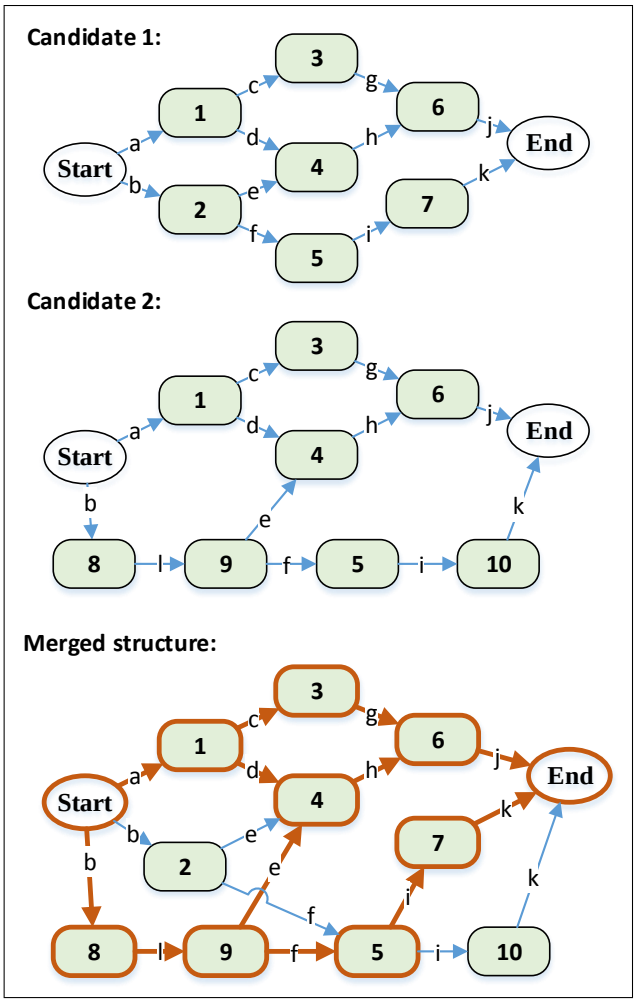

Figure 5: Example of the merge and extraction process used in the graph crossover operation.

consequence contains redundant nodes and edges. The merging process is depicted in Figure 5, and consists of combining any two nodes that represent the same service into a single node, maintaining all original 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*. This strategy restricts the offspring to utilise structures already present on either parent, thus achieving a similar outcome as traditional crossover operations. Figure 5 highlights one of the possible solutions that could be extracted from the merged structure.

3.4 Fitness function

The fitness function employed for the evolutionary process is based on the one presented in [Rodriguez-Mier *et al.*, 2010]. It 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 function is that it encourages features indicative of the quality of the overall composition [Rodriguez-Mier *et al.*, 2010]: 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:

$$fitness_i = \omega_1 \cdot \frac{1}{runPath_i} + \omega_2 \cdot \frac{1}{\#atomicProcess_i} \quad (1)$$

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 $\#atomicProcess_i$ is the total number of service nodes included into a solution i. In the original paper, the fitness function also presents a third criterion measuring the degree of satisfaction of the inputs 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.

4 Experiment Design

Experiments were carried out to compare the performance of GraphEvol against the GP approach presented in [Rodriguez-Mier *et al.*, 2010], seeking to validate the hypothesis that the execution time, 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.

4.1 Datasets

The datasets employed in this experiments were OWL-S TC V2.2 [Kuster *et al.*, 2008] and WSC 2008 [Bansal *et al.*, 2008], both of which present service collections of varying sizes accompanied by ontologies. These ontologies detail how the input and output types of the services in the collection relate to each other at a conceptual level. For example, suppose the ontology contains information stating that an *integer* is a type of *number*. Given this information, if a service is annotated as requiring a *number* as an input and another is annotated as producing an *integer* as an output, then one may infer that the *integer* output can be used to fulfil the *number* input. As shown by this example, the advantage of having such information encoded in an ontology is that it enables the matching of data types that are named differently

	GraphEvol			GP Approach		
Task	**Time (ms)**	**runPath**	**#atomicProcess**	**Time (ms)**	**runPath**	**#atomicProcess**
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 [Rodriguez-Mier *et al.*, 2010].

but are nevertheless compatible. Tasks 1–5, which are outlined in [Rodriguez-Mier *et al.*, 2010], were used when to test GraphEvol against the OWL-S TC dataset; tasks WSC 2008-1, WSC 2008-2, and WSC 2008-5 were used for testing against the WSC 2008 dataset.

4.2 Parameters

Testing was conducted using a personal computer with an Intel Core i7-4770 CPU (3.4 GHz), and 8 GB RAM. 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 [Rodriguez-Mier *et al.*, 2010].

5 Results

Experiment results for GraphEvol 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 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 $\uparrow$ for significantly higher results and $\downarrow$ 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.

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. Finally, experiments were conducted comparing GraphEvol to an analogous GP approach. Results showed that the quality of the results produced by GraphEvol always matched or surpassed those produced by GP, requiring roughly the same or less execution time. It is believed that improvements can be made to the crossover operator (e.g. restrict crossover operation to graphs that present a certain degree of similarity), therefore research on this topic is a natural future development. Another possibility is the execution of additional tests, this time expanding the experiment design to include fitness functions that measure formal Quality of Service (QoS) attributes [Jaeger and Mühl, 2007] of candidate compositions.

References

[Aversano *et al.*, 2006] Lerina Aversano, Massimiliano Di Penta, and Kunal Taneja. A genetic programming approach to support the design of service compositions. *International Journal of Computer Systems Science & Engineering*, 21(4):247–254, 2006.

[Bansal *et al.*, 2008] Ajay Bansal, M Brian Blake, Srividya Kona, Steffen Bleul, Thomas Weise, and Michael C Jaeger. 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*, pages 351–354. IEEE, 2008.

[Chen and Yan, 2014] Min Chen and Yuhong Yan. Qos-aware service composition over graphplan through graph reachability. In *Services Computing (SCC), 2014 IEEE International Conference on*, pages 544–551. IEEE, 2014.

[da Silva *et al.*, 2014] A.S. da Silva, Hui Ma, and Mengjie Zhang. A graph-based particle swarm optimisation approach to qos-aware web service composition and selection. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 3127–3134, July 2014.

[Deng *et al.*, 2013] Shuiguang Deng, Bin Wu, Jianwei Yin, and Zhaohui Wu. Efficient planning for top-k web service composition. *Knowledge and information systems*, 36(3):579–605, 2013.

[Gottschalk *et al.*, 2002] Karl Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to web services architecture. *IBM systems Journal*, 41(2):170–177, 2002.

[Huang *et al.*, 2009] Zhenqiu Huang, Wei Jiang, Songlin Hu, and Zhiyong Liu. Effective pruning algorithm for qos-aware service composition. In *Commerce and Enterprise Computing, 2009. CEC'09. IEEE Conference on*, pages 519–522. IEEE, 2009.

[Jaeger and Mühl, 2007] Michael C Jaeger and Gero Mühl. Qos-based selection of services: The implementation of a genetic algorithm. In *Communication in Distributed Systems (KiVS), 2007 ITG-GI Conference*, pages 1–12. VDE, 2007.

[Kuster *et al.*, 2008] U Kuster, Birgitta Konig-Ries, and Andreas Krug. Opossum-an online portal to collect and share sws descriptions. In *Semantic Computing, 2008 IEEE International Conference on*, pages 480–481. IEEE, 2008.

[Milanovic and Malek, 2004] Nikola Milanovic and Miroslaw Malek. Current solutions for web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.

[Perrey and Lycett, 2003] Randall Perrey and Mark Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.

[Qi and Palmieri, 1994] Xiaofeng Qi and Francesco Palmieri. 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.

[Rodriguez-Mier *et al.*, 2010] Pablo Rodriguez-Mier, Manuel Mucientes, Manuel Lama, and Miguel I Couto. Composition of web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010.

[Srivastava and Koehler, 2003] Biplav Srivastava and Jana Koehler. Web service composition-current solutions and open problems. In *ICAPS 2003 workshop on Planning for Web Services*, volume 35, pages 28–35, 2003.

[Wang *et al.*, 2012] Lijuan Wang, Jun Shen, and Jianming Yong. A survey on bio-inspired algorithms for web service composition. In *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*, pages 569–574. IEEE, 2012.

[Wang *et al.*, 2013] Anqi Wang, Hui Ma, and Mengjie Zhang. Genetic programming with greedy search for web service composition. In *Database and Expert Systems Applications*, pages 9–17. Springer, 2013.

[Yoo *et al.*, 2008] John Jung-Woon Yoo, Soundar Kumara, Dongwon Lee, and Seog-Chan Oh. A web service composition framework using integer programming with non-functional objectives and constraints. *algorithms*, 1:7, 2008.