

# A Graph-based QoS-Aware Method for Web Service Composition with Branching

## ABSTRACT

The concept of Service-Oriented Architecture, where individual services can be combined to accomplish more complex tasks, provides a flexible and reusable approach to application development. Their composition can be performed manually, however doing so may prove to be challenging if many service alternatives with differing qualities are available. Evolutionary Computation (EC) techniques have been employed successfully to tackle this problem, especially Genetic Programming (GP), since it is capable of encoding conditional constraints on the composition's execution paths. While compositions can naturally be represented as Directed Acyclic Graphs (DAGs), GP needs to encode candidates as trees, which may pose conversion difficulties. To address that, this work proposes an EC approach that represents solutions directly as DAGs. The tree-based and graph-based composition approaches are compared, showing significant gains in execution time when using graphs, sometimes up to two orders of magnitude. The quality levels of the solutions produced, however, are somewhat higher for the tree-based approach. This, in addition to a convergence test, shows that the genetic operators employed by the graph-based approach can be potentially improved. Nevertheless, the graph-based approach is shown to be capable of handling compositions with multiple conditional constraints, which is not possible when using the tree-based approach.

## CCS Concepts

•Information systems → Web services; •Computing methodologies → Bio-inspired approaches;

## Keywords

Web service composition; evolutionary computing; conditional branching; quality of service

## 1. INTRODUCTION

In recent years, the concept of Service-Oriented Architecture (SOA) has become increasingly popular [12]. According to SOA, software systems should be organised into independent functionality modules known as *Web services* [4], which are accessible over a network and employed in conjunction to fulfil the overall system's objectives. In other words, SOA encourages the idea of *Web service composition*, where existing services are reused to create new applications instead of re-implementing functionality that is already available. Such compositions can be performed manually, however this process may become time-consuming as the number of relevant candidate services increases. Thus, the ability to perform Web service compositions in a fully automated manner has become an important area of research in the field of service computing [10].

The process of Web service composition is challenging for three reasons: firstly, the *functional correctness* of service composition solutions must be ensured, so that for a given service request, a feasible service composition solution can be generated to fulfil the given service request; secondly, the creation of compositions with multiple execution *branches* must be supported, so that applications incorporating conditional constraints can be produced to adapt to dynamic changes of the environment [17]; thirdly, the *quality* (e.g. reliability, availability) of the candidate Web services to be included in composite services must be examined, so that the composite services can achieve the best possible qualities. Several automated approaches for Web service composition exist, but the great majority of them fails to address these three dimensions simultaneously. For instance, AI planning approaches focus on ensuring interaction correctness and branch support [15], while Evolutionary Computation (EC) techniques focus on interaction correctness and Quality of Service (QoS) measures [16].

Recently, a genetic programming (GP) approach was proposed to address these three composition dimensions simultaneously [3], but the tree representation used in this work may lead to situations in which the same service appears multiple times throughout a candidate tree, making it difficult to ensure the functional correctness of the composition solution (as further explained in section 2). If solutions are represented as Directed Acyclic Graphs (DAGs), on the other hand, it becomes much easier to verify the correctness of the connections between component services in the composition. Thus, the objective of this work is to propose a graph-based Evolutionary Computation Web service composition solution that simultaneously addresses the three di-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GECCO '16 July 20–24, 2016, Denver, CO, USA

© 2016 ACM. ISBN 123-4567-24-567/08/06...\$15.00

DOI: 10.475/123-4

mensions discussed above. The remainder of this paper is organised as follows. Section 2 describes the composition problem and existing works. Section 3 introduces the proposed graph-based composition approach. Section 4 outlines the experiments conducted to compare the performance of graph-based and tree-based approaches. Section 5 discusses the experimental results. Section 6 concludes the paper.

## 2. BACKGROUND

### 2.1 Existing Works

From the existing EC techniques, GP is perhaps the most flexible within the domain of Web service composition, since it allows a variable amount of services to be encoded into a solution in multiple configurations that are dictated by the chosen composition constructs. GP represents each solution as a tree, using one of two possible representations: the *input-as-root* representation [13, 3], where composition constructs (e.g. parallel, sequence) are encoded as non-terminal nodes of the tree and atomic Web services as terminal nodes, and the *output-as-root* representation [7], where the root node of the tree represents the composition output, the other non-terminal nodes are atomic Web services, and all of the terminal nodes represent the composition input.

The advantage of these two representations is that their crossover and mutation operations are relatively simple to implement, since modifications are restricted to certain subtrees. However, both of these approaches also present some disadvantages. The input-as-root representation using strongly-typed GP has the disadvantage of duplicating Web services throughout the tree, which means that modifying a part of the tree that contains a duplicated service may remove some fundamental connections to this service, thus breaking the correctness of the solution. The output-as-root representation, on the other hand, does not have this type of modification problem; however, it does not support conditional branching. That is because the set of composition outputs is represented as the root of the tree, meaning that there is only one set of outputs can be represented at a time. Another disadvantage of these techniques is that a graph representation must be generated first, and subsequently translated into a tree representation to ensure the functional correctness of trees.

Mabu, Hirasawa and Hu [8] introduce a technique for the evolution of graph-based candidates, called Genetic Network Programming (GNP). GNP has a fixed number of nodes within its structure, categorised either as processing nodes (responsible for processing data) or as judgment nodes (perform conditional branching decisions), and works by evolving the connections between these fixed nodes. While this approach has the advantage of using simple genetic operations, the number of nodes and outgoing edges per node must be fixed throughout the evolutionary process.

The works in [1, 11] present a graph-based genetic algorithm that is used to evolve representations of molecules. Atoms are represented as nodes, and their bonds as edges. Two types of genetic operations are supported: mutation, which can be the appending or removing of a node and its connecting bonds, and crossover, where edges are removed from each candidate until each graph is divided into two disconnected subgraphs that are then reconnected to create new child candidates. In the Web service composition domain, however, the need for additional restrictions means

that these genetic operators are no longer suitable.

To address these limitations, Silva, Ma and Zhang [14] introduce GraphEvol, an EC technique where Web service composition solutions are represented as Directed Acyclic Graphs (DAGs). The initial population is created using an algorithm that ensures functional correctness of compositions (i.e., the connections between service nodes are correct), and that genetic operators maintain this correctness throughout the evolutionary process. However, GraphEvol neither supports QoS-aware composition nor allows conditional constraints to be used. Thus, in this paper we will employ a DAG representation to perform QoS-aware service composition with conditional constraints. While the work of Wang et al. [17] does take conditional constraints into account, it does not allow for multiple possible outputs (i.e. separate branches that are not merged) and it is not QoS-aware.

### 2.2 A Motivating Example

A typical Web service composition scenario is that of an online book shopping system, as described in [3] and shown in Figure 1. The objective of this system, constructed using existing Web services, is to allow a customer to purchase a book using different methods of payment according to their account balance. Namely, if the customer has enough money to pay for the selected book, then they would like to pay for it in full; otherwise, they would like to pay for it in instalments. The idea is to construct this system in a fully automated manner, meaning that the services to be used, the connections between these services, and the overall flow of the system are determined automatically. When requesting the creation of a system with the desired functionality, the composition request provided consists of the overall inputs required by the system (e.g. book title, author, customer information), conditional constraints (e.g. the customer's account balance is less than the price of the chosen book), and the potential outputs produced by the system (e.g. a receipt, if the customer paid in full, or an initial bill, if the customer is paying in instalments).

### 2.3 Problem Formalization

A *Web service*  $S$  is represented using a functional description that includes an input concept  $I$  and an output concept  $O$  specifying what operation the service will perform, a categorical description of inter-related concepts specifying the service operation according to a common terminology  $\mathcal{T}$  in the application area, and a quality of service (QoS) description of non-functional properties such as response time and cost. For the categorical description, an ontology with definitions of “concepts” and the relationships between them must be specified. In previous works [6] a terminology for service ontology using description logic has been defined. A *terminology* is a finite set  $\mathcal{T}$  of assertions of the form  $C_1 \sqsubseteq C_2$  with concepts  $C_1$  and  $C_2$  as defined in [6].

A *service repository*  $\mathcal{D}$  consists of a finite collection of atomic services  $s_i, i \in \{1, n\}$  together with a service terminology  $\mathcal{T}$ . A *service request*  $R$  is normally defined with a pair  $(I_R, O_R)$ , where  $I_R$  is the input concept that users provide and  $O_R$  specifies the output concept that users require. In this paper, we consider that users may prefer different outputs depending on some condition [17]. In this case, we specify the output as  $(c?O_R, O_{R'})$ , meaning that if the value of the condition  $c$  is TRUE, then it produces

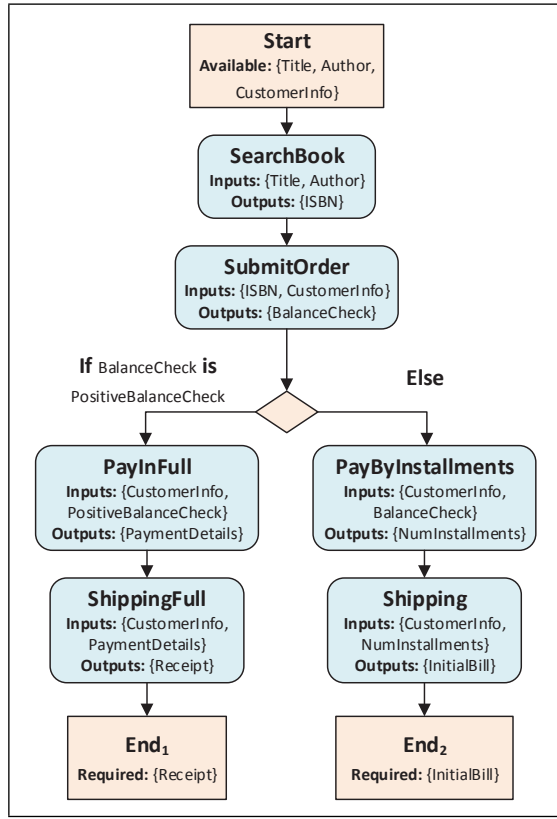


Figure 1: A Web service composition for an online book shopping system [3].

output  $O_R$ , otherwise  $O_{R'}$ . A service request can be represented as two special services, a start service  $s_0 = (\emptyset, I_R)$  and an end service  $s_e = \{O_R, \emptyset\}$  or  $s_{e'} = \{O_{R'}, \emptyset\}$ , when conditional constraints are used. Each output has a given probability of being produced, which is calculated through statistical analysis on the behaviour of the service(s) used to satisfy the request.

Services can be composed by process expressions [6]. The set of *process expressions* over a repository  $\mathcal{D}$  is the smallest set  $\mathcal{P}$  containing all *service constructs* that is closed under sequential composition construct  $\cdot$ , parallel construct  $\parallel$ , choice  $+$  and iteration  $*$ . That is, whenever  $s_i, s_j \in \mathcal{P}$  hold, then all  $s_i \cdot s_j$ ,  $s_i \parallel s_j$ ,  $s_i + s_j$  and  $s_i^*$  are process expressions in  $\mathcal{P}$ , with  $s_i$  and  $s_j$  are component services. A *service composition* is a process expression with component services  $s_i$ , each of which associating with an input and output type  $I_i$  and  $O_i$ . For example  $s_1 \cdot (s_2 \parallel s_3)$  is a service composition, meaning that  $s_1$  processed first followed with  $s_2, s_3$  processed in parallel.

We need to ensure that composite services are *feasible*, i.e. *functionally correct*. In particular, when two services are composed by a sequential construct, i.e.,  $s_i \cdot s_j$ , we need to ensure that service  $s_j$  matches  $s_i$ . A service  $s_j$  *fully matches* another service  $s_i$  if and only if  $O_i$  subsumes concept  $I_j$ , i.e.,  $O_i \sqsubseteq I_j$ . A service  $s_j$  *partially matches* another service  $s_i$  if  $O_i \sqcap I_j \sqsubset \perp$ . A service composition  $\mathcal{S}$  is a *feasible* solution for the service request  $R$  if the following conditions are satisfied:

- All the inputs needed by the composition can be provided by the service request, i.e.  $I_R \sqsubseteq I_{\mathcal{S}}$ ;

- All the required outputs can be provided by the composition, i.e.  $O_{\mathcal{S}} \sqsubseteq O_R$  if  $c$  is TRUE and  $O_{\mathcal{S}} \sqsubseteq O_{R'}$ , otherwise;
- For each component service  $s_j$  its input  $I_j$  should be provided by services  $s_i$  that were executed before it, i.e. the union of the output of all the services  $s_i$  is a sub-concept of  $I_j$ .

Service compositions can be naturally represented as Directed Acyclic Graphs (DAGs) [14]. Given a service repository  $\mathcal{D}$  and a service request  $R$ , the service composition problem is to find a directed acyclic graph  $G = \{V, E\}$ , where  $V$  is the set of vertices and  $E$  is the set of edges of the graph, with one starting service  $s_0$  and end service  $s_e$  and a set of intermediate vertices  $\{V_1, \dots, V_m\}$ , which represent atomic services selected from the service repository.

Service compositions can be also represented as trees, with intermediate nodes representing composition constructs and terminal nodes representing atomic services [3]. Each intermediate node is actually a composite service itself, with child nodes arranged according to the composition construct it represents. Therefore, to check the functional correctness of service composition represented as a tree we just need to check that each of the component services, including atomic services, satisfies the following: its input concept must be subsumed by the union of the properties of the input from the service requests and the properties produced by its preceding nodes.

## 2.4 QoS Model and Composition Constructs

In addition to the functional aspects of Web service composition, the goodness of the services included in a solution also plays a part in the creation of a composite system. This non-functional set of attributes is known as Quality of Service (QoS) [9], and it measures characteristics that are desirable in a service from a customer's point of view. In this work, four QoS attributes are considered: Time ( $T$ ), which measures the response time of a service once it has been invoked, Cost ( $C$ ), which specifies the financial cost of using a given service, Availability ( $A$ ), which measures the likelihood of a service being available at invocation time, and Reliability ( $R$ ), which is the likelihood of a service responding appropriately when invoked. The existing languages for Web service composition (e.g. BPEL4WS [18]) use certain constructs to control the flow of the resulting systems with regards to input satisfaction.

### 2.4.1 Sequence construct:

Services are chained sequentially, so that the outputs of a preceding service are used to satisfy the inputs of a subsequent service, as shown in Figure 2. The total cost and time of this construct are calculated by adding the values of its individual services, and the total availability and reliability by multiplying them.

### 2.4.2 Parallel construct:

Services are executed in parallel, so their inputs are independently matched and their outputs are independently produced, as shown in Figure 3. The availability, reliability and cost are calculated the same way as they are in the sequence construct, and the total time is determined by identifying the service with the longest execution time.

### 2.4.3 Choice construct:

Only one service path is executed, depending on whether the value of its associated conditional constraint is met at runtime. This is shown in Figure 4. In this construct, all overall QoS attributes are calculated as a weighted sum of the services from each individual path, where each weight corresponds to the probability of that path being chosen during runtime. These weights add up to 1.

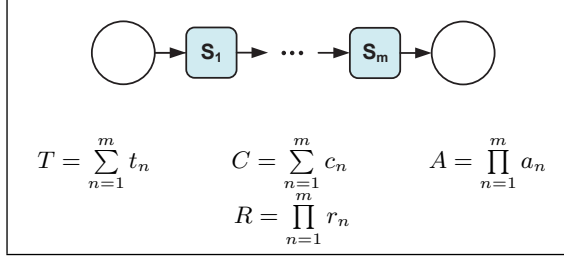


Figure 2: Sequence construct and calculation of its QoS properties [3].

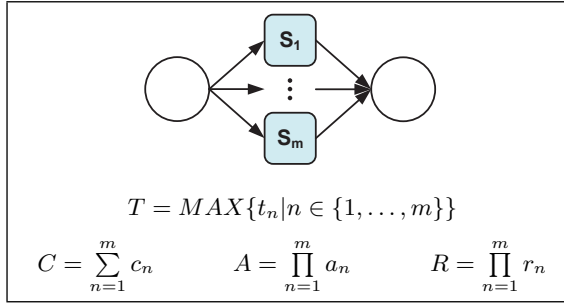


Figure 3: Parallel construct and calculation of its QoS properties [3].

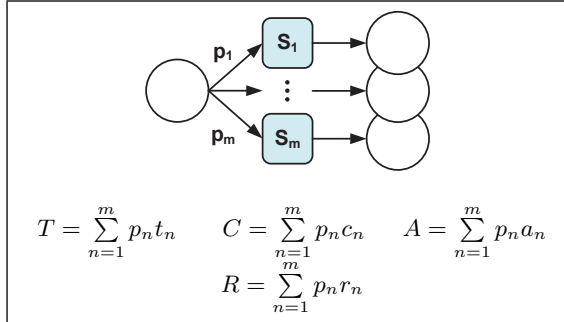


Figure 4: Choice construct and calculation of its QoS properties [3].

## 3. PROPOSED APPROACH

The approach proposed in this work allows for the representation of composite services with multiple execution branches, depending on conditional constraints specified in the composition request. A candidate may look like the representation shown in Figure 1, with each block encoded as a graph node, and edges flowing from the input (start) towards the output (end) nodes. However, nodes with multiple children and/or multiple parents can also be included when necessary, meaning that Directed Acyclic Graphs (DAGs) are

the basis of this representation. Algorithm 1 describes the general procedure followed by GraphEvol, and the following subsections explain the ways in which this basic technique was extended to allow for conditional branching.

---

### Algorithm 1: Steps of the GraphEvol technique [14].

---

- 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.
- 

### 3.1 Graph building algorithm

Candidates are created using the *buildGraph* procedure shown in Algorithm 2. This procedure requires *InputNode*, which is the root of the composition's request tree, a list of *relevant* candidate services from the service repository, and optionally a *candMap*. The *candMap* is used for the crossover procedure only, so it will be discussed later. Given these inputs, the algorithm proceeds to connect nodes to the graph, one at a time, until a complete solution is found. As explained earlier, the resulting composition will have several independent branches, thus the recursive procedure *buildBranch* has been created to handle each part of the composition. After connecting the *start* node to the graph, we execute *buildBranch* providing the first task it should achieve (i.e. *TaskNode*, which initially will be a conditional branching node), a list of candidates *candList* that contains services that are executable/reachable using the *start* node outputs, the partially built graph *G*, and other relevant data structures. Once the *buildBranch* procedure has finished executing, the graph *G* will be completed. The algorithm used for construction creates graphs from the *start* node to the *end* nodes in order to prevent cycles from forming, but this may lead to *dangling* nodes, which are nodes that do not have any outgoing edges despite not being *end* nodes. These are redundant parts of the solution, and thus they must be removed once *G* is built. Finally, the creation of the new candidate graph is finished.

Algorithm 2 also describes the *connectNode* procedure, used for adding a node to an existing graph. In addition to adding the given node *n* to *G*, and connecting it using the edges provided in the *connections* list, this procedure also checks if the current *TaskNode* objective has been reached. If the *TaskNode* represents a conditional node, we check that we are now capable of producing both the values required by the *if* case and by the *else* case when using the service we have just connected. On the other hand, if the *TaskNode* represents the *end* of a branch, we check that the list *allInputs* of potential inputs contain all values necessary to satisfy the inputs for the *end* node.

Algorithm 3 shows the *buildBranch* procedure, which recursively creates the branched Web service composition. Given a *TaskNode*, this procedure repeatedly adds services to the graph *G*, until the *TaskNode* goal has been reached.

**Algorithm 2:** Procedures for building a new candidate graph and for connecting a particular node to the graph [14].

---

```

1: Procedure buildGraph(InputNode, relevant,
2: candMap)
3:   start.outputs  $\leftarrow$  {InputNode.values};
4:   TaskNode  $\leftarrow$  InputNode.child;
5:   G.edges  $\leftarrow$  {};
6:   G.nodes  $\leftarrow$  {};
7:   allInputs  $\leftarrow$  {};
8:   connections  $\leftarrow$  {};
9:   connectNode(start, connections, G,
10:  allInputs, TaskNode);
11:   allowedAncestors  $\leftarrow$  {start};
12:   if candMap is null then
13:     candList  $\leftarrow$  findCands(start, allowedAncestors,
14:  relevant);
15:   else
16:     candList  $\leftarrow$  {node | (start, node)  $\in$  candMap};
17:   buildBranch(TaskNode, candList, connections,
18:  allInputs, G, relevant, allowedAncestors, candMap);
19:   removeDangling(G);
20:   return G;

21: Procedure connectNode(n, connections, G,
22: allInputs, TaskNode)
23:   n.objective  $\leftarrow$  TaskNode;
24:   G.nodes  $\leftarrow$  G.nodes  $\cup$  {n};
25:   G.edges  $\leftarrow$  G.edges  $\cup$  connections;
26:   if TaskNode is ConditionNode then
27:     if |n.outputs| > 1 then
28:       return
29:         ((TaskNode.general  $\sqsubseteq$  n.outputs.general  $\wedge$ 
30:         TaskNode.specific  $\sqsubseteq$  n.outputs.specific), n);
31:     else
32:       return (false, n);
33:   else
34:     return (TaskNode.outputs  $\sqsubseteq$  allInputs, n);

```

---

More specifically, nodes from the *candList* are considered for addition. A candidate *cand* is randomly chosen from *candList*, and it is connected to the graph (using the *connectNode*) procedure if all of its inputs can be matched by the *ancestor* outputs (i.e. the outputs of nodes already present in that particular execution branch). The set of services in *connections*, which are used to connect *cand* to *G*, is a minimal set, meaning that the output of these services matches all the inputs of *cand*, but if any connection is removed from the set that is no longer the case. After each *cand* service is connected to *G*, the *candList* is updated to contain any services that have now become executable due to the outputs of *cand*, and to exclude *cand*. Once the *TaskNode* goal has been reached, the *connectTaskNode* procedure is called to finish the construction of that branch, either by connecting an *end* node to it or by further splitting the branch according to a new *TaskNode* condition. In case of the latter, *connectTaskNode* will invoke the *buildBranch* procedure again.

As previously explained, Algorithm 4 is responsible for finishing the construction of a given execution branch, according to one of two scenarios. In the first scenario, the *TaskNode* reached is a conditional node, meaning that the branch will be further split into an *if-and-else* structure. In this case, the *TaskNode* is added to *G*, connected through the previously added service in *connResult*[1] (i.e. the service that matched the outputs required for the condition to be checked). Since the branching occurs based on the values

**Algorithm 3:** Indirectly recursive procedure for building one of the branches of the new candidate graph.

---

```

1: Procedure buildBranch(TaskNode, candList,
2: allInputs, G, relevant, allowedAncestors, candMap)
3:   goalReached  $\leftarrow$  false;
4:   connResult;
5:   while  $\neg$ goalReached do
6:     found  $\leftarrow$  false;
7:     foreach cand  $\in$  candList do
8:       connections  $\leftarrow$  {};
9:       ancestors  $\leftarrow$  {x.outputs | x  $\in$  G  $\wedge$  x  $\in$ 
10:  allowedAncestors};
11:       if cand.inputs  $\sqsubseteq$  ancestors then
12:         connections  $\leftarrow$  connections  $\cup$  {x  $\leftarrow$ 
13:  minimal(ancestors)};
14:         found  $\leftarrow$  true;
15:       if found then
16:         connResult  $\leftarrow$ 
17:         connectNode(cand, connections, G,
18:         allInputs, TaskNode);
19:         goalReached  $\leftarrow$  connResult[0];
20:         allowedAncestors  $\leftarrow$ 
21:         allowedAncestors  $\cup$  {cand};
22:         if candMap is null then
23:           candList  $\leftarrow$  candList  $\cup$ 
24:           findCands(cand, allowedAncestors,
25:           relevant);
26:         else
27:           candList  $\leftarrow$  candList  $\cup$ 
28:           {node | (cand, node)  $\in$  candMap};
29:         break;
30:     candList  $\leftarrow$  candList - {cand}
31:   connectTaskNode(TaskNode, connResult, G,
32:  allowedAncestors, candList, candMap);

```

---

produced by *connResult*[1], the probabilities of producing these different output possibilities are copied from this service. Then, the *buildBranch* procedure is invoked twice more, once for the *if* branch and once for the *else* branch, providing the appropriate children of *TaskNode* to the next construction stages. In the second scenario, the *TaskNode* reached is an output node, meaning that the branch leads to an *end* node without any further splitting. In this case, the *TaskNode* is simply connected to *G*, using a minimal set of services already in the graph which produce all the outputs required by this *end* node.

## 3.2 Mutation and Crossover

The procedures for performing the mutation and crossover operations are shown in Algorithm 5. The general idea behind the *mutation* procedure is to modify a part of the original graph *G*, but maintain the rest of the graph unchanged. In order to do so, a node *n* is initially selected as the mutation point, provided that it is not an *end* or a *condition* node. If this node is the *start* node, an entirely new candidate graph is constructed; otherwise, all nodes whose input satisfaction depends upon node *n* are removed from *G*, and so are any subsequent splits of that branch. The construction of this partially-built graph is then finished by invoking the *buildBranch* procedure and providing the original *TaskNode* (*n*'s objective) and appropriate data structures to it. The *allowedAncestors*, the *candList*, and *allInputs* are calculated based on the remaining nodes of *G*. The mutation operator was designed in this way so that it allows for variations of the original candidate, at the same time maintaining the correctness of the connections between services.

**Algorithm 4:** Procedure for finishing construction of a branch, splitting it further in case another condition exists.

```

1: Procedure connectTaskNode(TaskNode,
   connResult, G, allowedAncestors, candList, candMap)
2:   if TaskNode is ConditionalNode then
3:     TaskGoal.probs ← connResult[1].probs;
4:     G.nodes ← G.nodes ∪ {TaskNode};
5:     G.edges ← G.edges ∪ {connResult[1] → TaskNode};
6:     allowedAncestors ←
       allowedAncestors ∪ {TaskNode};
7:     connections ← {};
8:     if candMap is null then
9:       candList ← candList ∪
         findCands(TaskNode, allowedAncestors,
           relevant);
10:    else
11:      candList ← candList ∪
        {node | (TaskNode, node) ∈ candMap};
12:    allInputs ← {};
13:    ifChild ← TaskNode.ifChild;
14:    if ifChild is OutputNode then
15:      allInputs ← {x.outputs | x ∈
        allowedAncestors};
16:    buildBranch(ifChild, candList, allInputs,
      G, relevant, allowedAncestors, candMap);
17:    elseChild ← TaskNode.elseChild;
18:    if elseChild is OutputNode then
19:      allInputs ← {x.outputs | x ∈
        allowedAncestors};
20:    buildBranch(elseChild, candList, allInputs,
      G, relevant, allowedAncestors, candMap);
21:  else
22:    ancestors ← {x.outputs | x ∈ G ∧ x ∈
      allowedAncestors};
23:    connections ← {x → TaskNode | x ∈
      minimal(ancestors)};
24:    G.nodes ← G.nodes ∪ {TaskNode};
25:    G.edges ← G.edges ∪ connections;

```

In the case of *crossover*, the general idea is to reuse connection patterns from two existing candidates  $G_1$  and  $G_2$  in order to create a new child candidate that combines elements from these two parents. In order to do so, the original connections of  $G_1$  and  $G_2$  are abstracted into a map called *candMap*. This map can be queried to determine all the connections (from both parents) that can be made starting from a given node  $x$ . After having assembled this map, the *buildGraph* procedure is invoked to create a child candidate. The difference is that the addition of candidates to the *candList* is done by querying the *candMap* to determine which services could be reached from the current node according to the connection patterns in the original parents. One of the advantages of this crossover implementation is that it allows for an operation that reuses connection information from both parents. Additionally, this operation can be executed using the already existing graph-building algorithm with minimal changes.

### 3.3 Fitness function

The fitness function is used for optimising solutions according to their overall QoS, and was based on the function shown in [3]. This function measures the overall quality of a composition candidate by performing a weighted sum of the overall QoS attributes of a given candidate:

$$fitness_i = w_1 A_i + w_2 R_i + w_3 (1 - T_i) + w_4 (1 - C_i) \quad (1)$$

**Algorithm 5:** Procedures for performing mutation and crossover on graph candidates [14].

```

1: Procedure mutation(G, InputNode, relevant)
2:   n ← selectNode(G);
3:   if n is start then
4:     return buildGraph(InputNode, relevant, null);
5:   else
6:     TaskNode ← n.objective;
7:     removeNodes(n);
8:     allInputs ← {};
9:     candList ← {};
10:    allowedAncestors ← {};
11:    foreach node ∈ G.nodes do
12:      allowedAncestors ←
        allowedAncestors ∪ {node};
13:    foreach node ∈ G.nodes do
14:      candList ← candList ∪
        findCands(node, allowedAncestors, relevant);
15:    if TaskNode is OutputNode then
16:      allInputs ← {x.outputs | x ∈
        allowedAncestors};
17:    return buildBranch(TaskNode, candList, allInputs,
      G, relevant, allowedAncestors, null);
18: Procedure crossover(G1, G2, InputNode, relevant)
19:   candMap ← {(x, y) | x → y ∈ G1.edges};
20:   candMap ← candMap ∪ {(x, y) | x → y ∈ G2.edges};
21:   return buildGraph(InputNode, relevant, candMap);

```

where  $\sum_{k=1}^4 w_k = 1$

This function produces values in the range [0,1], where a fitness of 1 means the best quality. Because this is a maximising function, the Time  $T$  and cost  $C$  are offset by 1 in the formula, so that higher scores correspond to better qualities for these attributes as well. The overall quality attributes  $A$ ,  $R$ ,  $T$ , and  $C$  of a composition are normalised, with the upper bound of  $T$  and  $C$  multiplies by the total number of services in the repository [7].

## 4. EXPERIMENT DESIGN

Experiments were conducted to compare the performance of our approach to that of another composition technique that uses a GP tree representation to encode branching constraints [3]. This tree-based approach follows the input-as-root representation discussed in Section 2. The datasets employed in this comparison were based on those proposed in [3], since they contain composition requests that require one conditional constraint. The initial sets were extended to make the problem more complex by replicating each original service in the repository ten times. The replicated services were then assigned randomly generated QoS values that were within the original ranges for each quality attribute. A new dataset, also based on those presented in [3], was created to measure the performance of our technique when addressing more complex composition requests. More specifically, this dataset was based on dataset 1, with two changes: firstly, manually-generated services whose composition results in a solution with fitness 1 (i.e. an artificial optimum) were added; secondly, the composition request was modified to require three conditional constraints instead of one, as before. Because the tree approach cannot handle more than one conditional constraint [3], it has not been tested with this dataset.

## 4.1 Parameters

Tests were executed on a personal computer with an Intel Core i7-4770 CPU (3.4 GHz), and 8GB RAM. In total, four different tests were executed, three of them being comparisons and the fourth being an experiment on the behaviour of our approach only. The parameters for all approaches were based on those proposed by Koza [5]. 30 independent runs were conducted for each approach, with a population size of 500 during 51 generations. The crossover probability was set to 0.8, and mutation and reproduction probabilities were set to 0.1 each. No elitism was used, and tournament selection with a tournament size of 2 was employed. Finally, all fitness function weights were set to 0.25, indicating that all quality attributes are considered equally important by the user requesting the composition.

## 5. RESULTS

The results of the comparison between the two composition techniques using the datasets from [3] are shown in Table 1, where the first column lists the dataset used and its number of atomic services, the second column displays the mean time with standard deviation for executing our graph-based approach, the third column shows the mean fitness and standard deviation of the best solution found by our graph-based approach, and the fourth and fifth columns show the corresponding time and fitness values for the tree-based approach. Time values, including their standard deviations, are rounded to 1 decimal point of precision; fitness values and their standard deviations are rounded to 2 decimal points. Wilcoxon signed-ranked tests at 95% confidence level were run where possible to ascertain whether the differences in time and fitness values for a given dataset are significant, with  $\downarrow$  denoting significantly lower values and  $\uparrow$  denoting significantly higher values.

Set (size)	Graph-based		Tree-based	
	Avg. time (s)	Avg. fitness	Avg. time (s)	Avg. fitness
1(1738)	11.2 $\pm$ 1.5 $\downarrow$	0.76 $\pm$ 0.02	235.2 $\pm$ 52.8	0.85 $\pm$ 0.01 $\uparrow$
2(6138)	35.0 $\pm$ 3.3 $\downarrow$	0.67 $\pm$ 0.01 $\uparrow$	609.3 $\pm$ 112.7	0.65 $\pm$ 0.00
3(6644)	19.0 $\pm$ 1.0 $\downarrow$	0.72 $\pm$ 0.01	2264.5 $\pm$ 296.2	0.74 $\pm$ 0.02 $\uparrow$
4(11451)	49.1 $\pm$ 1.6 $\downarrow$	0.56 $\pm$ 0.01	900.6 $\pm$ 138.2	0.77 $\pm$ 0.08 $\uparrow$
5(11990)	34.9 $\pm$ 1.3 $\downarrow$	0.81 $\pm$ 0.02	2680.7 $\pm$ 217.8	0.83 $\pm$ 0.01 $\uparrow$
6(24178)	140.7 $\pm$ 21.8 $\downarrow$	0.77 $\pm$ 0.02 $\uparrow$	19772.2 $\pm$ 2142.7	0.76 $\pm$ 0.02
7(45243)	345.4 $\pm$ 55.5 $\downarrow$	0.79 $\pm$ 0.02	24467.1 $\pm$ 5482.4	0.90 $\pm$ 0.03 $\uparrow$
8(89309)	522.1 $\pm$ 94.5 $\downarrow$	0.82 $\pm$ 0.00	51850.3 $\pm$ 5768.2	0.82 $\pm$ 0.04

Table 1: Comparison results.

As expected, the execution times of our graph-based approach are significantly lower than those of the tree-based approach for all datasets. Surprisingly, the performance gains of the graph-based method are more pronounced as the size of the dataset grows, culminating into a difference of two orders of magnitude for the largest dataset. These results are extremely encouraging, because they demonstrate that representing solutions directly as a DAG facilitates the enforce-

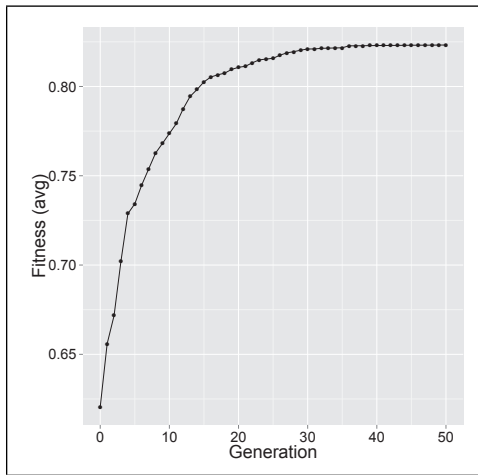
ment of correct output-input connections between services, which in turn translates to lower execution costs. With regards to the quality of solutions, results show that the fitness of the tree-based solutions is slightly higher for the majority of datasets, but occasionally the quality of the solutions produced using the tree-based approach is superior (datasets 1, 4, and 7). Despite this, the graph-based approach produces solutions that are significantly better for datasets 2 and 6. These results indicate a trade-off between the techniques, depending on whether the objective is to produce solutions using a lower execution time or to focus on the quality of these solutions instead. However, the rate of improvement for these two aspects should also be taken into consideration when comparing techniques. Namely, while the tree-based approach may result in solutions with a quality gain of up to 20% (for dataset 4), the execution time required by the graph-based approach may be as little as 1% (for datasets 5 to 8) of that required by the tree-based approach. Thus, our proposed graph-based approach is a viable Web service composition alternative, particularly when working in a dynamic environment where the quality of services changes over time [2].

A convergence test was also conducted using the variation of dataset 1 described earlier. Since the task associated with this new dataset requires the creation of a composition that takes multiple conditional constraints into account, it can only be tested against our graph-based approach (recall that the tree-based approach cannot handle more than one conditional constraint [3]). Therefore, the objective of this test is twofold: to demonstrate that the graph-based approach can indeed handle composition requests with multiple conditional constraints, and to examine the convergence of this approach. The results of this test are shown in Figure 5, where the mean fitness values calculated over 30 independent runs have been plotted for each generation. The artificial optimum is known to correspond to the value 1 when evaluated using the fitness function, which means that solutions with a value close to 1 are likely to be close to this optimum. The plotted mean values clearly show that the most significant fitness improvements occur between generations 0 and 30, with the remaining generations performing smaller improvements that eventually lead to the convergence of the population. Interestingly, the fitness values between generations 40 and 50 remain constant at 0.88, without the slightest variation. We hypothesise that this behaviour is caused by the genetic operators used in the graph-based approach, since the mutation and crossover patterns dictate the trajectory of the overall search. This suggests that improvements to these operators could be performed to enhance the search strategies when using a DAG representation. Nevertheless, these results show that the graph-based approach can successfully handle composition tasks with multiple conditional constructs, meanwhile still generating solutions with a reasonably high quality.

## 6. CONCLUSIONS

This work has discussed an Evolutionary Computing approach to Web service composition with conditional constraints that represents solution candidates as Directed Acyclic Graphs, as opposed to the previously explored tree-based representation. The increased flexibility of the graph representation requires specialised versions of mutation and crossover operators to in order to maintain the functional





**Figure 5: Average fitness per generation with artificial optimum.**

correctness of the solutions, and algorithms for accomplishing this were proposed. The graph-based approach was compared to an existing tree-based composition method, with results showing that the graph-based approach executes significantly faster than the tree-based approach for all datasets, even reaching a difference of two orders of magnitude for the largest dataset. The resulting solution qualities for both approaches, on the other hand, were found to be generally slightly higher when using the tree-based approach. Another experiment that tests the ability of the graph-based approach to handle composition tasks with multiple conditional constraints was also conducted, at the same time analysing the convergence behaviour of the approach. This test has shown that the graph-based approach can successfully produce solutions of a reasonably high quality for the more complex composition request, even though it could not reach the known global optimum. The convergence results potentially indicate that the genetic operators employed by the graph-based approach can be improved. Consequently, future work in this area should explore alternative designs for the genetic operators used in the evolution process, likely resulting in improved fitness levels for the solutions produced.

## 7. REFERENCES

- [1] N. Brown, B. McKay, F. Gilardoni, and J. Gasteiger. A graph-based genetic algorithm and its application to the multiobjective evolution of median molecules. *Journal of chemical information and computer sciences*, 44(3):1079–1087, 2004.
- [2] K. M. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A fault taxonomy for web service composition. In *Service-oriented computing-ICSOC 2007 Workshops*, pages 363–375. Springer, 2007.
- [3] A. S. da Silva, H. Ma, and M. Zhang. A GP approach to QoS-aware Web service composition including conditional constraints. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pages 2113–2120. IEEE, 2015.
- [4] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web services architecture. *IBM systems Journal*, 41(2):170–177, 2002.
- [5] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.
- [6] H. Ma, K.-D. Schewe, and Q. Wang. An abstract model for service provision, search and composition. In *Proceedings of the 2009 IEEE Asia-Pacific Services Computing Conference (APSCC)*, pages 95–102. IEEE, 2009.
- [7] H. Ma, A. Wang, and M. Zhang. A hybrid approach using genetic programming and greedy search for QoS-aware Web service composition. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XVIII*, pages 180–205. Springer, 2015.
- [8] S. Mabou, K. Hirasawa, and J. Hu. A graph-based evolutionary algorithm: genetic network programming (GNP) and its extension using reinforcement learning. *Evolutionary Computation*, 15(3):369–398, 2007.
- [9] D. Menasce. QoS issues in Web services. *Internet Computing, IEEE*, 6(6):72–75, 2002.
- [10] N. Milanovic and M. Malek. Current solutions for Web service composition. *IEEE Internet Computing*, 8(6):51–59, 2004.
- [11] C. A. Nicolaou, J. Apostolakis, and C. S. Pattichis. De novo drug design using multiobjective evolutionary graphs. *Journal of Chemical Information and Modeling*, 49(2):295–307, 2009.
- [12] R. Perrey and M. Lycett. Service-oriented architecture. In *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*, pages 116–119. IEEE, 2003.
- [13] P. Rodriguez-Mier, M. Mucientes, M. Lama, and M. I. Couto. Composition of Web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010.
- [14] A. Sawczuk da Silva, H. Ma, and M. Zhang. Graphevol: A graph evolution technique for Web service composition. In *Database and Expert Systems Applications, volume 9262 of Lecture Notes in Computer Science*, pages 134–142. Springer International Publishing, 2015.
- [15] S. Sohrabi, N. Prokoshyna, and S. A. McIlraith. Web service composition via the customization of golog programs with user preferences. In *Conceptual Modeling: Foundations and Applications*, pages 319–334. Springer, 2009.
- [16] L. Wang, J. Shen, and J. Yong. A survey on bio-inspired algorithms for Web service composition. In *Computer Supported Cooperative Work in Design (CSCWD), IEEE 16th International Conference on*, pages 569–574. IEEE, 2012.
- [17] P. Wang, Z. Ding, C. Jiang, and M. Zhou. Automated Web service composition supporting conditional branch structures. *Enterp. Inf. Syst.*, 8(1):121–146, Jan. 2014.
- [18] P. Wohed, W. M. van der Aalst, M. Dumas, and A. H. Ter Hofstede. Analysis of Web services composition languages: The case of BPEL4WS. In *Conceptual Modeling-ER 2003*, pages 200–215. Springer, 2003.