Particle Swarm Optimisation with Sequence-Like Indirect Representation for Web Service Composition

No Author Given

No Institute Given

Abstract. Automated Web service composition, which refers to the creation of a complex application from pre-existing building blocks (Web services), has been an active research topic in the past years. The advantage of having an automated composition system is that it allows users to create new applications simply by providing the required parameters, instead of having to manually assemble the services. Existing approaches to automated composition rely on planning techniques or evolutionary computing (EC) to modify and optimise composition solutions *directly* in their tree/graph form, a complex process that requires several constraints to be considered before each alteration. To improve the search efficiency and simplify the checking of constraints, this work proposes an *indirect* Particle Swarm Optimisation (PSO)-based approach. The key idea of the indirect approach is to optimise a service queue which is then decoded into a composition solution by using a planning algorithm. This approach is compared to a previously proposed graph-based direct representation method, and experiment results show that the indirect representation can lead to a greater (or equivalent) quality while requiring a lower execution time. The analysis conducted shows that this is due to the design of the algorithms used for building and evaluating the fitness of solutions.

1 Introduction

Software developers around the world are well acquainted with *Web services*, which may be defined as applications that provide operations and/or data and are accessible via the network using communication protocols [7]. The modular nature of Web services has led users to think of them as building blocks for more complex applications, selected and integrated as needed from a repository of available candidates in a process known as *Web service composition* [5]. As the number of candidates in the repository grows and as composition tasks become more complex, performing the selection and integration of services manually becomes increasingly difficult [12]. Additionally, if the repository contains multiple candidates with equivalent functionality but different quality attributes, then manually choosing the ideal alternative to include in a composition may become infeasible [8]. To overcome these challenges, researchers have been investigating the development of techniques to perform *automated Web service composition*

[15]. By using these techniques, the *composition requestor* would be able to simply specify the inputs and outputs of the desired application, and an *automated composition system* would then correctly select and integrate services into a correct composition solution.

There are normally two tasks to be considered in Web service composition: maintaining the *correctness* of solutions, i.e. ensuring atomic services are connected in a way that can be executed at run time, and optimising solutions according to their overall *Quality of Service*. To accomplish this, there are typically three different kinds of methods: the first group focuses on creating a correct composition [18]; the second group optimises the quality of compositions assuming that an abstract workflow is already known [25]; the third group attempts to address both of these concerns simultaneously, creating a correct workflow and at the same time optimising the quality of the services included in the composition [20]. However, simultaneously accomplishing these two tasks increases the complexity of these approaches, since the optimisation must also respect a number of interrelated constraints [22].

The overall goal of this paper is to investigate an indirect representation to the problem of Web service composition, proposing a Particle Swarm Optimisation (PSO)-based approach [6] that represents each solution candidate as a queue of services. Each queue is decoded into the corresponding composition workflow by using a specific graph-planning algorithm, verifying the correctness of the connections between services. This work accomplishes three objectives:

1. It identifies suitable encodings for representing a queue of services within a PSO particle vector.
2. It proposes decoding algorithms that efficiently create composition workflows from a service queue as efficiently as possible.
3. It compares the indirect approach with a state-of-the-art direct composition approach to verify that there is indeed a performance gain without a loss of solution quality.

The remainder of this paper is organised as follows: section 2 provides the fundamental background on the Web service composition problem, including a literature review; section **??** describes the indirect representation proposed in this paper; section 3 presents two PSO methods that use this indirect representation; section 4 describes the experiments conducted to test the performance of the novel PSO methods; section 5 presents the results of these experiments; section 6 concludes the paper.

2 Background

2.1 Problem Description and Example

The fundamental idea of Web service composition is to combine Web services into a structure that accomplishes a more complex task. A *Web service* $S = (input(x_1, x_2, ..., x_n),\ output(y_1, y_2, ..., y_n),\ QoS(time,\ cost,\ availability,\ reliability))$ requires a set of *inputs*, produces a set of *outputs*, and has an associated set of *quality attributes*. The fundamental elements in the composition problem are a *service repository* $SR = \{S_1, ..., S_m\}$ containing the services, and a *composition request* $R = (input(i_1, i_2, ..., i_n), output(o_1, o_2, ..., o_n))$

which specifies the overall inputs that should be made available when executing the composition as well as the overall outputs the composition should produce. The objective of this problem is to create a service composition with the best possible overall quality attributes, optimised according to a set of *objective functions* $f_1, f_2, ..., f_n$, where $1 \leq i \leq n$ and $i \in QoS$. There are three fundamental *constraints* required in a composition solution: firstly, the inputs of each service must be fully satisfied by predecessor services in the composition ($\forall x : input(x) \supseteq output(pred(x))$); secondly, the outputs of the starting node of a composition must be the composition requests overall inputs ($output(s) = input(R)$); thirdly, the inputs of the ending node of a composition must be the composition requests overall outputs ($input(e) = output(R)$). The travel problem, a well-known Web service composition example which has been extensively described in the literature [21,19], is shown in Figure 1 as a concrete example of this problem description.

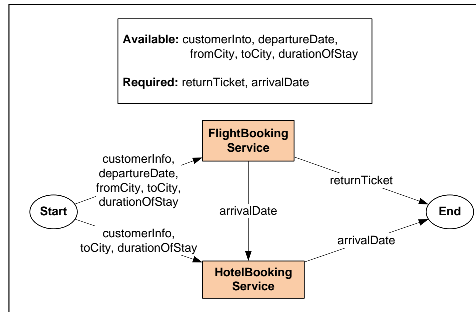


Fig. 1: Example of a solution to a Web service composition task [20].

2.2 Quality of Service and Composition Constructs

When creating compositions, it is necessary to pay attention to the Quality of Service (QoS) properties of each selected service, i.e. a QoS-aware composition approach is needed. There exist many Web service quality properties, from security levels to service throughput [14]. Based on the properties selected in previous works [9,26], in this paper we consider four of them: the probability of a service being available (A) upon request, the probability of a service providing a reliable response to a request (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 producing a reliable response, 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. The configuration of services in a composition is dictated by constructs used in building a workflow showing how services connect to each other [27]. This work considers two composition configurations, sequence and parallel, that are recognised by Web

service composition languages such as BPEL4WS [26,4]. These two constructs are described as follows.

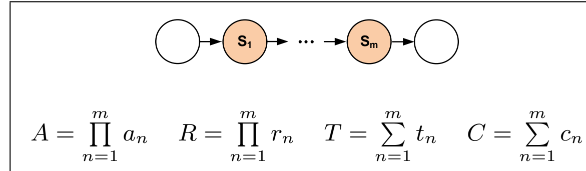


Fig. 2: Sequence construct and formulae for calculating its QoS properties [26].

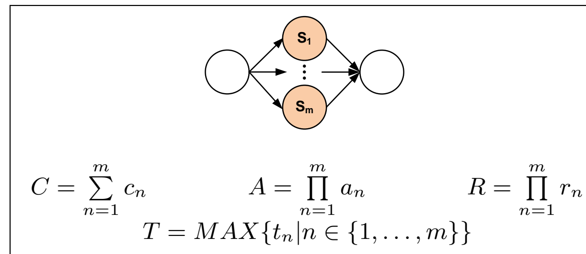


Fig. 3: Parallel construct and formulae for calculating its QoS properties [26].

Sequence construct Services in a composition construct are chained sequentially, so that the outputs of a preceding service fulfil the inputs of the subsequent one, as shown in Figure 2. The total time (T) and total cost (T) for a sequence can be calculated simply by adding the individual time and cost values of the composing services. The availability (A) and reliability (R) of individual services are expressed as probabilities, therefore the total composition values can be calculated by multiplying the individual service values.

Parallel construct The inputs of components in a parallel construct are satisfied independently, which allows these services to be executed simultaneously. The output produced by this construct can then be provided to any subsequent services, as shown in Figure 3. The overall QoS values for the parallel construct are calculated using the same formulae applied to the sequence construct, with the exception of the total time (T), which is simply the time of the component service that takes the longest to execute.

2.3 Related Work

In addition to planning [17] and traditional optimisation strategies [28], a wide variety of Evolutionary Computing (EC) approaches have been applied to the problem of Web service composition [24,16]. One of the earliest works in this area [3] applies genetic algorithms to optimise the overall Quality of Service (QoS) of a composition. Its objective is to select a set of concrete services that fulfil the required functionality of their abstract counterparts, ensuring that the selected

set results in a composition with the best possible quality. Even though this approach takes QoS into account, it is not capable of performing *fully automated* composition, which is when the composition workflow is automatically deduced at the same time that the services to include in the composition are identified. Several works employ particle swarm optimisation (PSO) to solving the problem of service composition [25,13,29], but similarly to genetic algorithms they focus exclusively on semi-automated composition.

Another approach [18] employs Genetic Programming (GP) to perform fully automated Web service composition, representing solutions as trees with candidate services as the leaf nodes and composition constructs as the inner nodes. A context-free grammar is used to generate new individuals at the beginning of the evolutionary process, as well as ensuring structural correctness during the crossover and mutation operations. Despite its favourable experimental results, this approach has the shortcoming of neglecting the Quality of Service of compositions and optimising candidates according to workflow topology measures such as the length of the longest path in the composition and the number of atomic services included.

Finally, some approaches both create the composition workflow and optimise the quality of the overall composition [26,20]. These works accomplish this by relying on variable-size solution representations (trees or directed acyclic graphs) and by measuring the quality of candidate compositions through the fitness function. In [26], the fitness function is responsible for penalising solutions that are not *functionally correct*, i.e. solutions that contain services whose inputs have not been entirely fulfilled; in [20], candidate initialisation and genetic operators are restricted to only produce functionally correct solutions. While these approaches do consider both workflow creation and quality improvement simultaneously, they perform operations to the solution workflows directly, which requires quite complex constraint checks.

3 PSO with Indirect Representation

The core idea explored in this work is to optimise solutions indirectly, using a representation that is then decoded into the final composition. Given that the verification of correctness constraints is very time-consuming, the indirect representations, which are usually simpler than the direct one, are expected to simplify or even remove the computationally expensive tasks of constraint checking and solution repairing, and thus improve the search efficiency. However, the implication of using this approach is that its efficiency and the quality of the solutions produced is affected by two factors: the specific representation used during the optimisation process, and the decoding process used for translating the particular representation into an actual service composition. In the context of an evolutionary computing approach, the use of an indirect representation requires candidates to be decoded before the usual fitness evaluation step.

The representation investigated in this work uses a linear format that is meant to represent a sequence of services, i.e. a service queue. The composition solution is then decoded from this queue by using an algorithm that adds services one by one to the solution according to the queue's ordering. PSO is the technique

chosen for the indirect optimisation, since the linear structure of its particles naturally lends itself to representing a service queue. This approach follows the usual PSO steps [6], though with some particularities shown in Algorithm 1. Firstly, the size of particles is determined based on the number of candidate services being considered for the composition, with each candidate service being mapped to an index of the particle's position vector (each position holds a weight between 0.0 and 1.0, inclusive). Secondly, solutions must be built using a graph-building algorithm before their fitness can be calculated; a queue of services is generated from the particle's position vector and used as the input for the algorithm, which decodes a corresponding solution graph from it. Finally, the particle's fitness can be calculated from this corresponding solution graph. Two PSO variations are presented in this work, and they are discussed separately in the subsections below.

ALGORITHM 1. Steps of the PSO-based Web service composition technique.

```
1:  Randomly initialise each particle in the swarm;
2:  while max. iterations not met do
3:      forall the particles in the swarm do
4:          Create queue of services using the particle's position vector;
5:          Build the corresponding composition graph using the queue;
6:          Calculate the fitness of the resulting graph;
7:          if fitness value better than pBest then
8:              Assign current fitness as new pBest;
9:          else
10:             Keep previous pBest;
11:     Assign best particle's pBest value to gBest, if better than gBest;
12:     Calculate the velocity of each particle according to the equation:
13:         v_id = v_id + c_1 * rand() * (p_id - x_id) + c_2 * rand() * (p_gd - x_id);
14:     Update the position of each particle according to the equation:
15:         x_id = x_id + v_id;
```

3.1 A Simple Forward Decoding PSO

The first PSO method proposed investigates a forward-decoding strategy, as it is potentially the simplest possible implementation to consider. In this method, a simple service queue is constructed using the particle's position vector before it can be translated into a composition graph. As shown in Figure 4, this is done by checking the service-to-index mapping for the particles' position vector. Each service is placed on the queue with an associated weight, which is retrieved by accessing the position vector with the index mapped to that service. This queue is then sorted according to these weights, placing the services with the highest weight at the head of the queue, and those with the lowest weight at the tail. Note that if two or more services have the same weight, then the ordering between them may vary.

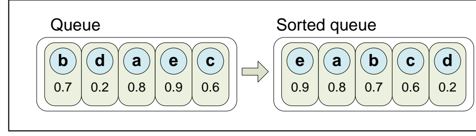


Fig. 4: Generating a service queue from a particle.

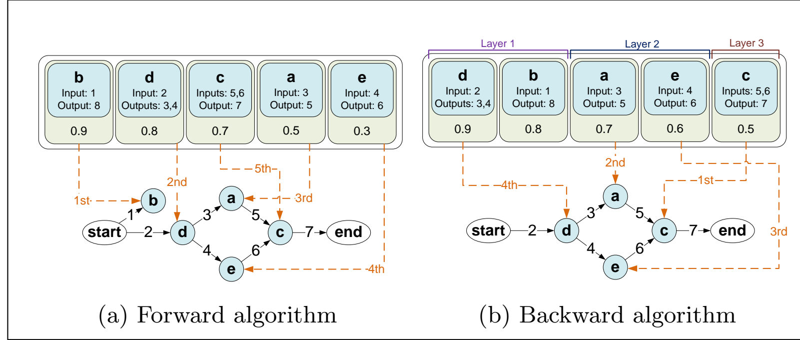
Graph-Building Algorithm By determining the service queue represented in a particle, it is then possible to build a composition graph from that service ordering, based on the Graphplan technique discussed in [2]. The graph is built in a forward way – from the *start* node towards the *end* node – to prevent the formation of cycles, which may lead to the addition of nodes that do not contribute to reaching the *end* (i.e. dangling nodes). To address this, after the graph has been constructed, it is submitted to a function that removes these redundant nodes.

ALGORITHM 2. Generating a composition graph from a queue

Input : I, O, *queue*
Output: composition graph G

1: Create *start* node with outputs I and *end* node with inputs O;
2: Create graph G containing the *start* node;
3: Create set of available outputs containing *start* outputs;
4: **while** *available outputs do not satisfy end inputs* **do**
5:     Get next candidate from queue;
6:     **if** *candidate inputs are satisfied by available outputs* **then**
7:         Connect node to graph;
8:         Remove it from queue and go back to the queue's beginning;
9: Connect end node;
10: Remove dangling nodes from graph G;
11: **return** G;

As shown in Algorithm 2, the values initially required are the composition task inputs (I), task outputs (O), and a *queue* of services as its input; this leads to the creation of a composition graph G. Firstly, the *start* node is added to the graph G, its outputs are added to a set that records all available outputs from the nodes currently in the graph. Then, the following steps are repeated until the available outputs can be used to fulfil all of the inputs of *end*: the next node of the queue is retrieved as a candidate; if all of its inputs can be fulfilled, the candidate is connected to the graph, its outputs are added to the set of available outputs, and it is removed from the queue; otherwise, the candidate in the next queue position is considered. Finally, *end* is connected to the graph, any dangling nodes are removed, and G is returned. This process is illustrated in Figure 5a, where node b is dangling.



(a) Forward algorithm (b) Backward algorithm

Fig. 5: Approaches for decoding a particle solution.

Fitness Calculation The fitness for a candidate graph is calculated using a function that evaluates its overall QoS values, considering the four attributes discussed in subsection 2.2. Note that when using a forward graph-building method, the fitness function can only be calculated once the entire graph has been decoded (as dangling nodes must not be included QoS calculations). The QoS attributes are combined using a commonly used weighted sum [23], according to the following function for a graph i:

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

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

A, C, and R are calculated using each atomic service in the graph according to the formulae shown in Figures 2 an 3; T, on the other hand, is determined by adding the individual times of the services that form the longest path in the graph, from start to end. The time is calculated based on the longest graph path because this allows us to handle both parallel and sequence constructs at the same time. The output of the fitness function is within the range [0, 1], with 1 representing the best possible fitness and 0 representing the worst. To ensure that the final result of the sum is within this range, the values of A, C, R and T must all be normalised between 0 and 1 (for time and cost, the maximum value used in the normalisation is the highest individual service value in the repository multiplied by the total number of services in the repository) [23].

3.2 Layered Backward Decoding PSO

A variation to the PSO-based method proposed in the previous section, which considers the use of service layers, was also developed with the objective of further improving the efficiency of the indirect technique. This variation was implemented because the use of layers allows solutions to be decoded using a backwards algorithm, which is more computationally effective. A *layer* in this context refers to a group of discovered services whose inputs that can be completely fulfilled by a set of outputs (either given or from previous layers). The

same steps used in the non-layered method are also employed here, with three fundamental differences: firstly, before initialising the population candidate services are mapped to an index in the particle's position vector according to the position information from the layers; secondly, the the decoding and evaluation of the solution represented by a particle are performed in a single step by using a new algorithm that does not require the creation of a graph; finally, a graph representing the global best solution is created after the optimisation process has finished running. Each of these steps is explained below.

Layer Identification and Particle Mapping The unique feature of the PSO method proposed here is that it identifies the *composition layer* to which a service belongs. Before the optimisation process begins, the service repository is run through a discovery process [23] that identifies the services that could be possibly used in the composition. As shown in Algorithm 3, this filtering process requires the set of inputs (I) and the set of outputs (O) from the overall composition task, in addition to the service repository (R); given these inputs, it produces a list of candidate service layers that are relevant to the composition (L). The algorithm keeps track of all available outputs so far, and uses them to discover additional layers: if a previously undiscovered service has all of its inputs satisfied by the set of available outputs, then it is added to the current layer and its outputs are added to the set of available outputs. The discovery continues until no additional layers are found, and the final step verifies whether the desired composite output O can in fact be achieved using the services in the repository. Once the composition layers are identified, the particle mapping takes place. As shown in in Figure 6, each service layer is mapped to contiguous particle indices, effectively segmenting particles according to the layers. This segmentation facilitates the solution decoding process to be discussed in the following subsection, as the connectivity information can be reused during the optimisation process.

ALGORITHM 3. Discovering relevant service composition layers [23].

```
    Input  : I, O, R
    Output: service layers L
1:  Initialise output set with I;
2:  Discover services satisfied by output set;
3:  while at least one service discovered do
4:  |   Add services as the next layer in L;
5:  |   Add the outputs of these services to the output set;
6:  |_  Discover additional services satisfied by the updated output set;
7:  if Output set satisfies O then
8:  |   return L;
9:  else
10: |_  Report no solution;
```

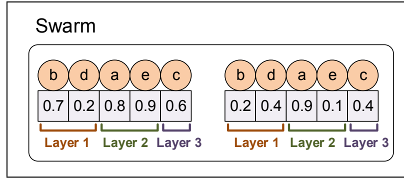


Fig. 6: Mapping of services to particles according to layers.

Solution Decoding and Fitness Calculation The solution decoding step employed in the layered PSO method is fundamentally different from that of the simple PSO method. Since particles are segmented by layers, it becomes possible to build graph solutions backwards (i.e. from the graph's end node towards the graph's start node) without leading to cycles being formed, provided that only services from previous layers are used to fulfil the input of a service in the current layer. Another difference is that the solution decoding process shown here does not produce a graph structure at the end of its execution, and instead calculates the fitness of the solution at the same time that the solution is identified. As discussed earlier, before the decoding process can begin it is necessary to order candidate services according to the weights contained in the particle. Instead of generating a single queue, however, we generate one individual queue for each of the layers mapped to that particle, creating a series of sorted layers (L). These sorted layers are then provided in conjunction with the solution's *end* node as the input to Algorithm 4, which calculates the corresponding fitness (f) to the particle's solution.

The algorithm works by keeping track of all service inputs that need to be satisfied, initialising it to contain all the inputs required by the end node. Then, the algorithm is executed from the last layer towards the first layer, each time performing the same series of steps. Firstly, all the inputs in set to satisfy that correspond to services in the current layer are identified. Then, all previous sorted layers are merged into a single service queue that is then used to fulfil the current inputs. As each service is selected from the queue to satisfy a given input, its QoS values are added to the QoS totals and all of its inputs are placed in the next-to-satisfy set. The *time* aspect of QoS is calculated by keeping track of the longest total time required by the services in previously processed layers, and by updating this total time with each new service addition. After all layers are satisfied, the fitness is calculated using the same fitness calculation as the previous PSO and the algorithm returns the result. A simplified depiction of the order in which this process is carried out is shown in Figure 5b.

Construction of Final Graph As the decoding algorithm described in the previous subsection does not create a directed acyclic graph out of every candidate solution, at the end of the run it is necessary to build a solution graph out of the overall fittest particle by using Algorithm 5. This algorithm has a very similar structure to the decoding one, but instead of calculating QoS values it connects services from earlier layers whose outputs fulfil the inputs of services in

Algorithm 4. Algorithm for decoding solutions and calculating their fitness.

Input : *end*, sorted layers L
Output: fitness f

1: Initialise variables to keep track of QoS:
$cost = 0, availability = 1, reliability = 1$;
2: Set all inputs of *end* as the next to satisfy, associating each input with time 0 and layer $|L| + 1$;
3: **for** *all sorted layers in* L*, from end to first* **do**
4: Identify the inputs from next-to-satisfy set that correspond to services in this layer;
5: **while** *not all of these inputs have been satisfied* **do**
6: Find the next service from the previous layers with the highest weight;
7: **if** *service outputs satisfy at least one input* **then**
8: Update running QoS values with service QoS (add to running cost, multiply with running availability and reliability);
9: Add the inputs of service to the set of next inputs to satisfy, each associated with (service time + highest time from satisfied inputs) and current layer position;
10: Find the total composition time as the highest from the remaining set of next inputs to satisfy;
11: Calculate fitness f using total QoS values;
12: **return** f;

later layers. As before, the algorithm goes through all composition layers, though in this case the final step is to connect the *start* node to any service with inputs that are still unfulfilled. Finally, the composition graph G is returned.

4 Experiment Design

Experiments were conducted to evaluate the performance of the PSO-based indirect composition methods in comparison to a graph-based direct composition approach [20], with the hypothesis that the indirect representation will produce solutions with equivalent quality but requiring shorter execution times. The graph-based approach was chosen for the comparison because of its flexibility, as it can also simultaneously optimise the quality of solutions and ensure their correctness. All experiments were conducted on a personal computer with 8 GB RAM and and an Intel Core i7-4770 CPU (3.4GHz). The datasets and tasks from WSC-2008 [1] and WSC-2009 [10] were used to compare the graph-based and PSO-based methods, with 30 independent runs for each approach using each dataset. The parameters were chosen based on common settings from the literature [11,6]. For both PSO methods, 100 iterations were run for a swarm of 30 particles, having both c_1 and c_2 as 1.49618, w as 0.7298 and all weights in the fitness function was 0.25. For the graph-based approach, a population of size 500 was evolved for 51 generations, with mutation probability of 0.8, both

Algorithm 5. Algorithm for building final graph solution.

Input : *start*, *end*, sorted layers L
Output: final graph G

1: Create graph G containing the end node;
2: Set all inputs of *end* as the next to satisfy, associating each input with the *end* node and layer $|L| + 1$;
3: **forall the** *sorted layers in* L*, from end to first* **do**
4: Identify the inputs from next-to-satisfy set that correspond to services in thWeb servicesis layer;
5: **while** *not all of these inputs have been satisfied* **do**
6: Find the next service from the previous layer with the highest weight;
7: **if** *the outputs of this service satisfy at least one of the inputs for this layer* **then**
8: Add service node to graph;
9: Add edges connecting this service node to the nodes whose inputs it satisfies;
10: Add the inputs of the service node to the set of next inputs to satisfy, each associated with the service node and current layer position;
11: Add *start* node to graph;
12: Add edges connecting *start* node the associated nodes of all remaining inputs in the next-to-satisfy set;
13: **return** G;

crossover and reproduction with a probability of 0.1, tournament selection with a tournament of size 2, and all fitness function weights as 0.25.

5 Results

Results are presented and discussed in the following subsections, where the solution fitness and execution time means are shown accompanied by the standard deviation. A Wilcoxon signed-rank test at 95% confidence level was conducted to ascertain whether the differences between the two PSO methods and the graph-based approach are statistically significant, and the symbols $\uparrow$ and $\downarrow$ are used to indicate values significantly larger than the graph-based approach and significantly smaller than the graph-based approach, respectively.

5.1 Solution Fitness

The fitness results presented in Table 1 generally show that the solution fitness produced by the PSO-based methods is equivalent to that of the graph-based approach. However, it must also be noted that the fitness of PSO solutions is significantly higher for a number of datasets (08-3, 08-6, 08-7, 09-4), whereas this is not the case for the graph-based approach. Thus, these results indicate that the PSO-based methods are preferable when the focus of the composition process is on the quality of the resulting solutions. This difference in quality was investigated by manually comparing the solutions produced by the graph-based

approach to those produced by layered PSO for dataset WSC-08-6, where the difference is most pronounced. For most runs with this dataset, the layered PSO produced many solutions with fitness 0.4980, whereas the graph-based approach did not produce any solutions with fitness higher than 0.4976. When comparing the topology of the solutions, an interesting observation was made: some of the layered PSO solutions included more services (42) and edges (107) than the graph-based solutions (40 services, 107 edges), but still had better overall quality. A hypothesis for this behaviour is that the operators used by the graph-based approach discourage extensive exploitation of specific solution topologies, since its operators are fundamentally based on topological changes; the indirect approach, on the other hand, updates solutions by changing particle weights that may or may not influence the decoded solution topology. This flexibility potentially allows larger structures to be exploited by the layered PSO for longer, until an area with promising quality is found.

Table 1: Mean solution fitness results for the two PSO-based methods and the graph-based composition approach.

Dataset (# servs.)	**Simple PSO**	**Layered PSO**	**Graph-based**
WSC-08-1 (158)	0.4928 ± 0.00118	0.4928 ± 0.00119	0.4916 ± 0.0000517
WSC-08-2 (558)	0.5949 ± 0.0130	0.5936 ± 0.0140	0.5993 ± 0.00
WSC-08-3 (608)	0.4894 ± 0.000289 ↑	0.4902 ± 0.000201 ↑	0.4879 ± 0.000145
WSC-08-4 (1041)	0.5120 ± 0.00256	0.5141 ± 0.000720	0.5088 ± 0.00124
WSC-08-5 (1090)	0.4971 ± 0.000139	0.4971 ± 0.0000993 ↑	0.4969 ± 0.0000414
WSC-08-6 (2198)	0.4979 ± 0.000124 ↑	0.4980 ± 0.000124 ↑	0.4976 ± 0.0000229
WSC-08-7 (4113)	0.4993 ± 0.0000381 ↑	0.4993 ± 0.0000403 ↑	0.4991 ± 0.0000191
WSC-08-8 (8119)	0.4994 ± 0.00000919	0.4994 ± 0.0000303	0.4994 ± 0.00000219
WSC-09-1 (572)	0.5630 ± 0.0125	0.5727 ± 0.0173	0.5664 ± 0.00991
WSC-09-2 (4129)	0.4993 ± 0.0000255	0.4993 ± 0.0000526	0.4993 ± 0.00000804
WSC-09-3 (8138)	0.5064 ± 0.00219	0.5058 ± 0.00293	0.5060 ± 0.00121
WSC-09-4 (8301)	0.4993 ± 0.0000473 ↑	0.4994 ± 0.0000508 ↑	0.4992 ± 0.0000103
WSC-09-5 (15211)	0.4996 ± 0.0000105	0.4996 ± 0.0000130	0.4996 ± 0.00000531

5.2 Execution Time

With regard to the execution time of the two techniques, shown in Table 2, an interesting pattern is observed: while the time required by the simple PSO-based method is consistently higher than that of the graph-based approach, the time required by the layered PSO-based method is consistently lower. This is the case for two reasons: firstly, the layered PSO decodes solutions backwards (from end to start), meaning it does not explore paths that do not ultimately connect the beginning and the end of a composition; secondly, during the decoding process the layered PSO goes through the services in the particle roughly $|layers| \times |services|$ in the worst case, whereas the worst case for the simple PSO is $|services|!$. These two key differences cause the layered method to check for significantly less potential service connections, which accounts for the time difference. A simple test was carried out to confirm this supposition, running the simple and layered PSO methods once each with WSC2008-8 (using the same settings as before) and counting how many times each of those methods checked for potential service connections during the particle decoding process.

The simple PSO count was 362,045,030 after finishing the run, while the layered PSO count was 2,126,349. From the results in Table 2, we see that the layered-to-simple execution time ratio for WSC2008-8 is roughly 1:70, while the layered-to-simple count ratio is roughly 1:170. The fact that these two execution aspects have similarly high ratios lends credence to the conjecture that the time difference is correlated with the number of times the decoding algorithms check for potential connections during a run.

Table 2: Mean execution time in seconds for the two PSO-based methods and the graph-based composition approach.

Dataset (# servs.)	**Simple PSO**	**Layered PSO**	**Graph-based**
WSC-08-1 (158)	2.1 ± 0.5	0.3 ± 0.1	3.2 ± 0.4
WSC-08-2 (558)	$4.1 \pm 2.1 \uparrow$	$0.4 \pm 0.1 \downarrow$	2.6 ± 0.4
WSC-08-3 (608)	$24.7 \pm 5.9 \uparrow$	$0.9 \pm 0.1 \downarrow$	14.3 ± 1.1
WSC-08-4 (1041)	$16.4 \pm 6.5 \uparrow$	0.5 ± 0.1	6.1 ± 0.6
WSC-08-5 (1090)	$29.1 \pm 8.2 \uparrow$	$0.9 \pm 0.1 \downarrow$	10.1 ± 1.3
WSC-08-6 (2198)	$195.6 \pm 45.7 \uparrow$	$3.8 \pm 0.2 \downarrow$	21.9 ± 1.5
WSC-08-7 (4113)	$202.4 \pm 61.0 \uparrow$	$3.2 \pm 0.4 \downarrow$	52.6 ± 3.7
WSC-08-8 (8119)	$539.0 \pm 145.0 \uparrow$	$7.4 \pm 0.6 \downarrow$	75.2 ± 13.8
WSC-09-1 (572)	3.8 ± 1.3	$0.4 \pm 0.1 \downarrow$	3.2 ± 0.5
WSC-09-2 (4129)	$168.6 \pm 38.8 \uparrow$	$3.2 \pm 0.3 \downarrow$	18.1 ± 1.6
WSC-09-3 (8138)	260.0 ± 99.3	$5.0 \pm 1.1 \downarrow$	23.3 ± 0.6
WSC-09-4 (8301)	$1378.4 \pm 577.5 \uparrow$	$21.0 \pm 3.3 \downarrow$	65.1 ± 2.9
WSC-09-5 (15211)	$2124.0 \pm 580.0 \uparrow$	$11.9 \pm 2.1 \downarrow$	151.1 ± 17.8

5.3 Discussion

The results displayed in this section show that the indirect approach performs well with a PSO implementation, though it was designed to work in a general sense. Thus, we theorise that other solution representations, decoding algorithms, and optimisation techniques in an indirect context could also yield superior results to those produced by direct approaches. Fundamentally, the indirect approach facilitates the checking of correctness constraints in a solution by restricting it to the decoding step, as opposed to incorporating it into the search process. This separation reduces the risk of overly constraining the search process, since invalid solutions can be easily prevented when using a robust decoding algorithm. The potential disadvantage of the indirect approach is that the chosen solution representation may lead to a very large search space, therefore care is needed when making this design decision.

6 Conclusions

This work introduced two PSO-based QoS-aware Web service composition methods that rely on an indirect solution representation, as opposed to the direct representations used by current works in the area. The key idea of these methods is to optimise a queue of candidate atomic services, identifying the sequence that leads to the construction of a composition with the highest possible quality. In order to evaluate the quality of a candidate, the queue is fed into algorithms that decode the underlying solution and calculate its fitness. These PSO-based

methods were compared to a graph-based approach with direct solution representation, with results showing that the quality of the solutions produced by PSO generally matches or surpasses those produced by the graph-based approach but with shorter execution times. Additionally, the indirect representation successfully handles the complexities faced when simultaneously optimising the quality and ensuring the correctness of service compositions. Future works in this area should investigate ways of further improving and simplifying the decoding algorithms used during the PSO fitness calculation step, as well as considering alternative methods such as genetic algorithms.

References

1. 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)
2. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial Intelligence 90(1), 281–300 (1997)
3. Canfora, G., Di Penta, M., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. In: Proceedings of the 7th annual conference on Genetic and evolutionary computation. pp. 1069–1075. ACM (2005)
4. 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)
5. Dustdar, S., Papazoglou, M.P.: Services and service composition–an introduction (services und service komposition–eine einführung). IT - Information Technology (vormals it+ ti) 50(2/2008), 86–92 (2008)
6. Eberhart, R.C., Shi, Y.: Particle swarm optimization: developments, applications and resources. In: Evolutionary Computation, 2001. Proceedings of the 2001 Congress on. vol. 1, pp. 81–86. IEEE (2001)
7. Gottschalk, K., Graham, S., Kreger, H., Snell, J.: Introduction to web services architecture. IBM systems Journal 41(2), 170–177 (2002)
8. Grønmo, R., Jaeger, M.C.: Model-driven semantic web service composition. In: Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific. pp. 8–pp. IEEE (2005)
9. 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)
10. Kona, S., Bansal, A., Blake, M.B., Bleul, S., Weise, T.: Wsc-2009: a quality of service-oriented web services challenge. In: Commerce and Enterprise Computing, 2009. CEC'09. IEEE Conference on. pp. 487–490. IEEE (2009)
11. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)
12. Lécué, F., Léger, A.: A formal model for semantic web service composition. In: The Semantic Web-ISWC 2006, pp. 385–398. Springer (2006)
13. Ludwig, S., et al.: Applying particle swarm optimization to quality-of-service-driven web service composition. In: Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on. pp. 613–620. IEEE (2012)

14. Menasce, D.: QoS issues in web services. Internet Computing, IEEE 6(6), 72–75 (2002)
15. Milanovic, N., Malek, M.: Current solutions for web service composition. IEEE Internet Computing 8(6), 51–59 (2004)
16. Pejman, E., Rastegari, Y., Esfahani, P.M., Salajegheh, A.: Web service composition methods: A survey. In: Proceedings of the International MultiConference of Engineers and Computer Scientists. vol. 1 (2012)
17. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: Artificial Intelligence: Methodology, Systems, and Applications, pp. 106–115. Springer (2004)
18. 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)
19. Sheng, Q.Z., Qiao, X., Vasilakos, A.V., Szabo, C., Bourne, S., Xu, X.: Web services composition: A decades overview. Information Sciences 280, 218–238 (2014)
20. da Silva, A., Ma, H., Zhang, M.: Graphevol: A graph evolution technique for web service composition. In: Chen, Q., Hameurlain, A., Toumani, F., Wagner, R., Decker, H. (eds.) Database and Expert Systems Applications, Lecture Notes in Computer Science, vol. 9262, pp. 134–142. Springer International Publishing (2015)
21. Tang, M., Ai, L.: A hybrid genetic algorithm for the optimal constrained web service selection problem in web service composition. In: Evolutionary Computation (CEC), 2010 IEEE Congress on. pp. 1–8. IEEE (2010)
22. Venkatraman, S., Yen, G.G.: A generic framework for constrained optimization using genetic algorithms. Evolutionary Computation, IEEE Transactions on 9(4), 424–435 (2005)
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: IEEE 16th International Conference on Computer Supported Cooperative Work in Design (CSCWD). pp. 569–574. IEEE (2012)
25. Wang, W., Sun, Q., Zhao, X., Yang, F.: An improved particle swarm optimization algorithm for qos-aware web service selection in service oriented communication. International Journal of Computational Intelligence Systems 3(sup01), 18–30 (2010)
26. Yu, Y., Ma, H., Zhang, M.: An adaptive genetic programming approach to QoS-aware web services composition. In: IEEE Congress on Evolutionary Computation (CEC). pp. 1740–1747. IEEE (2013)
27. 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)
28. 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)
29. Zhao, X., Song, B., Huang, P., Wen, Z., Weng, J., Fan, Y.: An improved discrete immune optimization algorithm based on pso for qos-driven web service composition. Applied Soft Computing 12(8), 2208–2216 (2012)