A Particle Swarm Optimisation-Based Indirect Approach to Web Service Composition

Alexandre Sawczuk da Silva, Yi Mei, Hui Ma, Mengjie Zhang

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

Abstract. Web service composition has been an active research topic in the past years, as the prospect of having a system that automatically creates complex applications from building blocks (Web services) given some parameters is quite attractive to users. Existing approaches to automated composition rely on modifying and optimising solution structures directly, a complex process that requires several constraints to be considered before each alteration. In this work, two variations of a Particle Swarm Optimisation (PSO) with indirect composition representations are explored, using each particle to hold a service queue which is then decoded into a composition solution graph before fitness evaluation. These approaches are compared to a previously proposed graph-based direct representation method, and experiment results show that the one of the proposed PSO approaches produces solutions with greater or equivalent quality while requiring a lower execution time. Bar graphs showing the frequency with which each atomic service appears in a solution are also produced, indicating that the PSO-based approaches explore a larger area of the search space than the direct representation method.

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 [9]. The advantage of these services lies in their modularity: they provide specific functionality that can be seamlessly integrated into larger applications, thus leading to code reuse and preventing time-consuming reimplementation [7]. Indeed, the self-contained 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* [7]. As the number of candidates in the repository grows and as composition tasks become more complex, however, performing the selection and integration of services manually becomes increasingly difficult [14]. 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 [10]. To overcome these challenges, researchers have been investigating the development of techniques to perform *automated Web service composition* [17]. 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.

One important group of techniques to Web service composition focuses on creating a correct composition workflow, where atomic services are connected in a way that can be executed at runtime [20]. Another group focuses on optimising the quality of compositions, assuming that an abstract workflow is already known for the composition and then employing optimisation techniques to bind the best possible set of concrete services to this existing structure [29,1]. A third group of approaches attempts to address both of these concerns simultaneously, creating a correct workflow at the same time that it optimises the quality of the services included in the composition [23]. However, simultaneously accomplishing these two tasks increases the complexity of these approaches, since the optimisation must also respect a number of interrelated constraints [26].

An alternative way to manage the complexity of addressing the simultaneous creation and optimisation of composition workflows is to optimise a set of heuristics that are then used for the construction of solutions, instead of directly optimising the actual compositions. This allows heuristics to be freely optimised, and constraints to be enforced once those heuristics are employed in the construction of a final solution. In this paper, two variations of a Particle Swarm Optimisation (PSO) [8] Web service composition approach that employs the strategy explained above are presented. More specifically, PSO is used to optimise the order of a queue of atomic services that is then provided to a graph-building algorithm. This algorithm uses the optimised queue to constructs composition solutions that abide by correctness constraints. The hypothesis is that this indirect optimisation approach will produce solutions that are equivalent to or better than direct optimisation approaches, while at the same time proposing a simpler composition approach.

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 3 describes the initial PSO approach proposed in this paper; section 4 presents the layered PSO approach proposed in this work; section 5 describes the experiments conducted to test the performance of the novel PSO approaches; section 6 presents the results of these experiments; section 8 concludes the paper.

2 Background

2.1 Problem Description and Example

The fundamental idea of Web service composition is to combine a set of modules, selected from a large pool of candidates, into a structure that accomplishes a more complex task. These modules are known as *Web services*, and require a set of *input* values in order to be executed, produce a set of *output* values, and have an associated set of *quality attributes* describing their non-functional aspects. The composition process must ensure that the Web services selected are compatible, and that the overall quality of the resulting solution is as high as possible. The fundamental elements in the composition problem are a *service repository* containing the services, and a *composition request* 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*, where each function is responsible for either maximising or minimising an aspect of the overall composition quality. There are three fundamental *constraints* required in a composition solution: firstly, the inputs of each service in the composition must be a subset of the outputs provided by their predecessor services, meaning that the inputs of each service are fully satisfied; secondly, the outputs of the starting node of a composition must be the composition requests overall inputs; thirdly, the inputs of the ending node of a composition must be the composition requests overall outputs.

A classic Web service composition example the travel problem, which has been extensively described in the literature [24,25,21]. In this scenario, the goal is to create a composite system that can book flights and accommodation at a given destination city according to the preferences specified by a user. More specifically, these overall composition inputs include the departure date and the destination city, and the overall composition outputs include reservation outcomes such as issued tickes and receipts. The services that make up the composition are selected from a pool of hotel and flight booking services that offer the desired elements of functionality with varying levels of quality, and structured into a workflow. Figure 1 depicts an example of a composition solution for booking travel transport and accommodation. When using this system the customer provides the appropriate required input (i.e. their personal information, the origin and destination cities, the departure date, and the duration of the stay) and retrieves the desired output (i.e. issued tickets with a specific arrival date at the destination city).

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 [16]. Based on the properties selected in previous works [11,30], 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

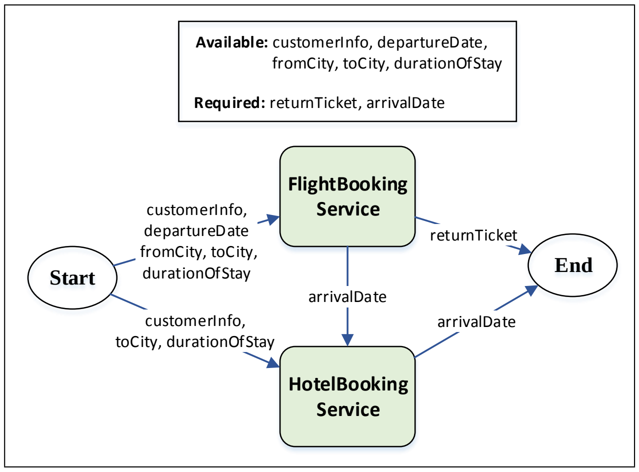
Available: customerInfo, departureDate, fromCity, toCity, durationOfStay
Required: returnTicket, arrivalDate
FlightBooking Service
HotelBooking Service
Start
End
customerInfo, departureDate fromCity, toCity, durationOfStay
customerInfo, toCity, durationOfStay
returnTicket
arrivalDate
arrivalDate

Fig. 1. Example of a solution to a Web service composition task [23].

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 [31]. This work considers two composition constructs, sequence and parallel, that are recognised by Web service composition languages such as BPEL4WS[30,6]. These two constructs are described as follows:

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

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

2.3 Particle Swarm Optimisation

PSO is an optimisation algorithm based on the behaviour of social animals, such as a school of flying birds [22]. The core idea of PSO is to employ a swarm of particles that can communicate with each other to explore a search space and

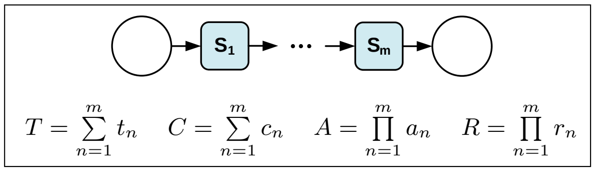
S_1
S_m
$T = \sum_{n=1}^{m} t_n \quad C = \sum_{n=1}^{m} c_n \quad A = \prod_{n=1}^{m} a_n \quad R = \prod_{n=1}^{m} r_n$

Fig. 2. Sequence construct and calculation of its QoS properties [30].

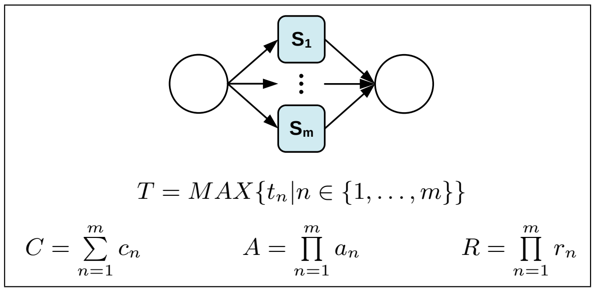
S_1
S_m
$T = MAX\{t_n | n \in \{1, \dots, m\}\}$
$C = \sum_{n=1}^{m} c_n \quad A = \prod_{n=1}^{m} a_n \quad R = \prod_{n=1}^{m} r_n$

Fig. 3. Parallel construct and calculation of its QoS properties [30].

identify the best possible solution. Each particle holds a *position* vector which specifies its current location in the solution space, and a *velocity* vector that specifies the direction in which the particle is moving. Each particle also keeps track of the best solution location it has visited so far (i.e. its *pBest*), and the whole swarm keeps track of the best overall location found so far (i.e. the *gBest*). As explained in [8], the original PSO process is divided into a number of steps, shown in Algorithm 1.

Algorithm 1. Steps of the PSO optimisation technique.

```
1. Randomly initialise each particle in the swarm.
while max. iterations not met do
    forall the particles in the swarm do
        2. Calculate the particle's fitness.
        if fitness value better than pBest then
            3a. Assign current fitness as new pBest.
        else
            3b. Keep previous pBest.
    4. Assign best particle's pBest value to gBest, if better than gBest.
    5. Calculate the velocity of each particle according to the equation:
        $v_{id} = v_{id} + c_1 * rand() * (p_{id} - x_{id}) + c_2 * rand() * (p_{gd} - x_{id})$
    6. Update the position of each particle according to the equation:
        $x_{id} = x_{id} + v_{id}$
```

The first step is to randomly initialise the position and velocity vectors of all particles in the swarm. Then, the following steps are repeated for the specified number of iterations: for each particle, the fitness value associated with its current location is calculated, and if it is better than the previous *pBest*, then the *pBest* is updated with it. Once the fitness has been calculated for all particles, we check whether the best particle's *pBest* is better than the current *gBest*, and we update *gBest* if that is the case. We then update the velocity and position of each particle according to the equations listed in steps 5 and 6, respectively [8]. The equation for updating the velocity modifies each dimension in the velocity vector (v_{id}) by using the current local best (p_{id}), the current global best (p_{gd}), the current particle position (x_{id}), two user-defined coefficients (c_1 and c_2), and random variables ($rand()$); the equation for updating the positions uses the newly calculated velocity (v_{id}) to update each dimension of the position vector (x_{id}).

2.4 Related Work

A wide variety of Evolutionary Computing (EC) approaches has been applied to the problem of Web service composition, as evidenced by literature surveys of the field [28,19]. One of the earliest works in this area [5] applies genetic algorithms to optimise the overall Quality of Service (QoS) of a composition. The composition process in this work is *semi-automated*, meaning that a workflow of abstract services has already been provided. In semi-automated compositions, the 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 [29,1,15,32], but similarly to genetic algorithm they focus exclusively on semi-automated composition.

Another approach [20] employs Genetic Programming (GP) to perform fully automated Web service composition, representing solutions as a 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, instead 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 [30,23]. 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 [30], 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 [23], 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 do not consider the idea of optimising solutions indirectly, and instead perform operations to the solution workflows directly.

3 PSO-Based Composition Approach

The aim of the PSO-based composition approach proposed in this paper is to optimise the queue of services used by a graph-building algorithm, which in turn determines the structure of the resulting composition. The rationale behind the creation of this method is that the indirect optimisation of the service queue allows for a simpler optimisation process than modifying composition solutions directly, while at the same time reducing the risk of creating an overly constrained search space. As Algorithm 2 shows, this approach follows the usual PSO steps, though with some particularities. 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 (step 1). 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 (step 3) and used as the input for the algorithm (step 4), which decodes a corresponding solution graph from it. Finally, the particle's fitness can be calculated from this corresponding solution graph (step 5). Further discussion on each of these highlighted steps is carried out in the subsections below.

3.1 Particle Mapping (Step 1)

As explained before, each particle in this approach represents a queue of candidate services, i.e. a queue of those services that can potentially be used to construct a solution that satisfies the composition request. These services are mapped to particles in the swarm following the principle depicted in Figure 4. Namely, each service is mapped to a different index of the particle's position vector in a consistent manner, so that a given index value corresponds to the same service across the entire swarm. This means that the number of dimensions in a particle is determined dynamically, and it corresponds to the number of services in the repository. Each cell in the the position vectors of all particles are randomly initialised with values between 0 and 1 (inclusive).

3.2 Creation of Service Queue (Step 3)

Before constructing a composition graph based on the information contained in a particle, it is necessary to construct a service queue using the particle's

Algorithm 2. Steps of the PSO-based Web service composition technique.

```
1. Map each candidate service to an index in the particle's position vector.
2. Randomly initialise each particle in the swarm.
while max. iterations not met do
    forall the particles in the swarm do
        3. Create queue of services using the particle's position vector.
        4. Build the corresponding composition graph using the queue.
        5. Calculate the fitness of the resulting graph.
        if fitness value better than pBest then
            6a. Assign current fitness as new pBest.
        else
            6b. Keep previous pBest.
    7. Assign best particle's pBest value to gBest, if better than gBest.
    8. Calculate the velocity of each particle according to the equation:
        $v_{id} = v_{id} + c_1 * rand() * (p_{id} - x_{id}) + c_2 * rand() * (p_{gd} - x_{id})$
    9. Update the position of each particle according to the equation:
        $x_{id} = x_{id} + v_{id}$
```

position vector. As shown in Figure 5, 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.

3.3 Graph-Building Algorithm (Step 4)

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 [3]. 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.

As shown in Algorithm 3, 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. We begin by assigning I as the set of outputs produced by the graph's *start* node, and O as the set of input nodes required by the graph's *end* node. Then, *start* is added to the graph G, its outputs are added to a set (*availOutputs*) that records all available outputs from the nodes currently in the graph, and an *index* variable is created to track positions in the *queue* (holding 0, which indicates the queue head). After this setup stage, the following steps are repeated until *availOutputs* can

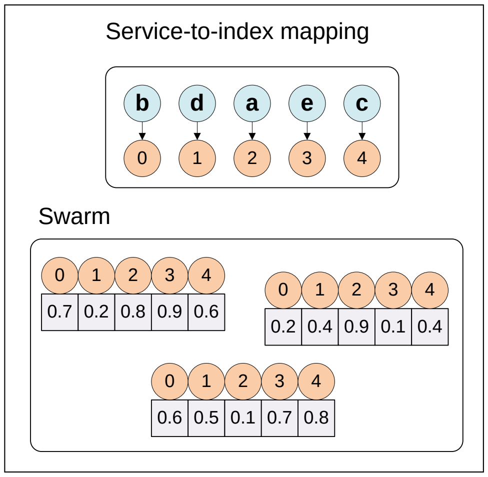

Fig. 4. Mapping of services to particles, and random initialisation.

be used to fulfil all of the inputs of *end*: the node at the current *queue* position (*index*) is retrieved as a candidate (*cand*); if all of its inputs can be fulfilled, *cand* is connected to the graph using $connectNode$ – which works by identifying existing nodes in the graph whose output fulfils the input of *cand*, and creating connecting edges from these existing nodes to *cand* –, its outputs are added to *availOutputs*, it is removed from the queue, and *index* is reset to 0; otherwise, the candidate in the next queue position is considered. Once all *end* node inputs can be fulfilled, *end* is connected to the graph, any dangling nodes are removed, and G is returned. In summary, by using this algorithm the structure of the resulting graph changes depending on the service ordering within the queue provided as input.

3.4 Fitness Calculation (Step 5)

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. The QoS attributes are combined using a weighted sum, 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

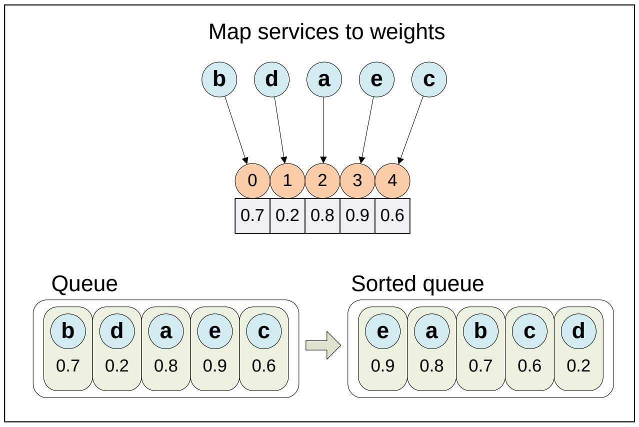

Fig. 5. Generating a service queue from a particle.

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. This is done by identifying the minimum and maximum values for each QoS attribute within the the dataset, then using the following formula (applied individually for each of the four quality attributes):

$$normalise(value) = \begin{cases} \frac{value - min}{max - min} & \text{if } max - min \neq 0. \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

4 Layered PSO-based Composition Approach

A variation to the PSO-based approach proposed in the previous section, which considers the use of service layers, was also developed. As shown in Algorithm 4, this approach follows the same steps used in the non-layered method, with three fundamental differences: firstly, the mapping of candidates to an index in the particle's position vector includes the division of the services that can potentially be included into a composition for the given task into 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 in the subsections below.

4.1 Layer Identification and Particle Mapping (Step 1)

The unique feature of the PSO approach proposed in this section is that it identifies the *composition layer* to which a service belongs. Before the optimisation

Algorithm 3. Generating a composition graph from a queue.

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

process begins, the service repository is run through a discovery process [27] that identifies the services that could be possibly used in the composition. As shown in Algorithm 5, 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 inputs so far (C_{search}), and uses them to discover additional layers: if a previously undiscovered service has all of its inputs satisfied by C_{search}, then it is added to the current layer and its outputs are added as available inputs in C_{search}. 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 accord-

Algorithm 4. Steps of the layered PSO-based Web service composition technique.

```
1. Identify layers and map each candidate service to an index in the
particle's position vector.
2. Randomly initialise each particle in the swarm.
while max. iterations not met do
    forall the particles in the swarm do
        3. Decode and calculate quality of corresponding solution layer by
        layer.
        if fitness value better than pBest then
            4a. Assign current fitness as new pBest.
        else
            4b. Keep previous pBest.
    5. Assign best particle's pBest value to gBest, if better than gBest.
    6. Calculate the velocity of each particle according to the equation:
        $v_{id} = v_{id} + c_1 * rand() * (p_{id} - x_{id}) + c_2 * rand() * (p_{gd} - x_{id})$
    7. Update the position of each particle according to the equation:
        $x_{id} = x_{id} + v_{id}$
    8. Create final solution graph based using gBest's weights.
```

ing to the layers. This segmentation facilitates the solution decoding process discussed in the following subsection.

4.2 Solution Decoding and Fitness Calculation (Step 3)

The solution decoding step employed in the layered PSO approach is fundamentally different from that of the simple PSO approach. 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 6, 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 $(nextToSatisfy)$, initialising it to contain all the inputs required by the end node. Then, the algorithm is executed from the last layer (layer $i = |L|$) towards the first layer (layer $i = 1$), each time performing the same series of steps. Firstly, all the inputs in $nextToSatisfy$ that correspond to services in

Algorithm 5. Discovering relevant service composition layers [27].

```
Input  : $I$, $O$, $R$
Output: service layers $L$
1: $C_{search} \leftarrow I$;
2: $L \leftarrow \{\}$;
3: $i \leftarrow i$;
4: $S_{found} \leftarrow DiscoverService()$;
5: while $|S_{found}| > 0$ do
6:     $L.add(S_{found}, i)$;
7:     $i \leftarrow i + 1$;
8:     $C_{search} \leftarrow C_{search} \cup C_{output}$ of $S_{found}$;
9:     $S_{found} \leftarrow DiscoverService()$;
10: if $C_{search} \supseteq O$ then
11:     return $L$;
12: else
13:     Report no solution;
```

the current layer are identified as *toSatisfy*. Then, all previous sorted layers (excluding the current layer i) are merged into a single service queue that is then used to fulfil the entries in *toSatisfy*. 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 *nextToSatisfy* 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. It is important to note that if an input in *toSatisfy* cannot be fulfilled by any of the services in the merged queue (i.e. $nextS = null$), this input will be fulfilled by the composition's given values. After all layers are satisfied, the fitness is calculated using the same fitness calculation approach as the previous PSO and the algorithm returns the result.

4.3 Construction of Final Graph (Step 8)

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 7. 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 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.

ALGORITHM 6. Algorithm for decoding solutions and calculating their fitness.

```
Input   : $end$, sorted layers $L$
Output: fitness $f$
1:  $solution \leftarrow \{\}$;
2:  $cost \leftarrow 0$;
3:  $availability \leftarrow 1$;
4:  $reliability \leftarrow 1$;
5:  $nextToSatisfy \leftarrow \{\}$;
6:  forall the $inputs$ of $end$ do
7:      $nextToSatisfy \leftarrow nextToSatisfy \cup \{(input, end.time, |L|)\}$;
8:  $i \leftarrow |L|$;
9:  for $i = |L|$ to $i = 1$ do
10:     $toSatisfy \leftarrow$ filterByLayer$(nextToSatisfy, i)$;
11:     $nextToSatisfy \leftarrow nextToSatisfy - toSatisfy$;
12:     $mergedL \leftarrow$ mergeLayers$(L, i)$;
13:     while $|toSatisfy| > 0$ do
14:         $nextS \leftarrow mergedL.next()$;
15:         if $nextS =$ null then
16:             $nextToSatisfy \leftarrow nextToSatisfy \cup toSatisfy$;
17:             $toSatisfy \leftarrow \{\}$;
18:         else
19:             $nLayer \leftarrow nextS.layer$;
20:             $satisfied \leftarrow$ getInputsSatisfied$(toSatisfy, nextS)$;
21:             if $|satisfied| > 0$ then
22:                 $solution \leftarrow solution \cup \{nextS\}$;
23:                 $cost \leftarrow cost + nextS.QoS.time$;
24:                 $availability \leftarrow availability \times nextS.QoS.availability$;
25:                 $reliability \leftarrow reliability \times nextS.QoS.reliability$;
26:             $time \leftarrow nextS.QoS.time$;
27:             $toSatisfy \leftarrow toSatisfy - satisfied$;
28:             $highestTime \leftarrow$ findHighestTime$(satisfied)$;
29:             forall the $inputs$ of $nextS$ do
30:                 $nextToSatisfy \leftarrow$
                    $nextToSatisfy \cup \{(input, time + highestTime, nLayer)\}$;
31: $time \leftarrow$ findHighestTime$(nextToSatisfy)$;
32: $f \leftarrow$ calculateFitness$(cost, time, availability, reliability)$;
33: return $f$;
```

Algorithm 7. Algorithm for building final graph solution.

```
Input  : $start$, $end$, sorted layers $L$
Output: final graph $G$
1:  $G.nodes \leftarrow \{end\}$;
2:  $G.edges \leftarrow \{\}$;
3:  $nextToSatisfy \leftarrow \{\}$;
4:  forall the $inputs$ of $end$ do
5:      $nextToSatisfy \leftarrow nextToSatisfy \cup \{(input, end, |L|)\}$;
6:  $i \leftarrow |L|$;
7:  for $i = |L|$ to $i = 1$ do
8:      $toSatisfy \leftarrow$ filterByLayer$(nextToSatisfy, i)$;
9:      $nextToSatisfy \leftarrow nextToSatisfy - toSatisfy$;
10:     $mergedL \leftarrow$ mergeLayers$(L, i)$;
11:     while $|toSatisfy| > 0$ do
12:         $nextS \leftarrow mergedL.next()$;
13:         if $nextS =$ null then
14:             $nextToSatisfy \leftarrow nextToSatisfy \cup toSatisfy$;
15:             $toSatisfy \leftarrow \{\}$;
16:         else
17:             $nLayer \leftarrow nextS.layer$;
18:             $satisfied \leftarrow$ getInputsSatisfied$(toSatisfy, nextS)$;
19:             if $|satisfied| > 0$ then
20:                 $G.nodes \leftarrow G.nodes \cup \{nextS\}$;
21:                 $G.edges \leftarrow G.edges \cup$ createEdges$(nextS, satisfied)$;
22:                 $toSatisfy \leftarrow toSatisfy - satisfied$;
23:                 forall the $inputs$ of $nextS$ do
24:                     $nextToSatisfy \leftarrow$
                        $nextToSatisfy \cup \{(input, nextS, nLayer)\}$;
25: $G.nodes \leftarrow G.nodes \cup \{start\}$;
26: $G.edges \leftarrow G.edges \cup$ createEdges$(start, nextToSatisfy)$;
27: return $G$;
```

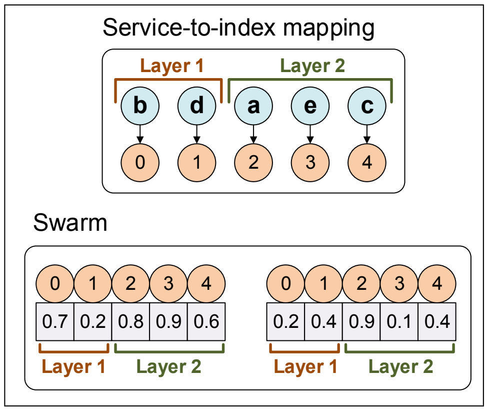

Fig. 6. Mapping of services to particles according to layers.

5 Experiment Design

Experiments were conducted to evaluate the performance of the PSO-based indirect composition approaches in comparison to a graph-based direct composition approach [23], according to three criteria. The first criterion is execution time, with the hypothesis that the graph-based approach will take more time to execute than the PSO-based approaches; the second criterion is best solution fitness, with the hypothesis that the solutions provided by the PSO-based approaches will match or surpass the quality of those produced by the graph-based approach; the third criterion is search space coverage, with the hypothesis that the PSO-based approaches will consider a greater variety of solutions during the search process than the graph-based approach. If these hypotheses are shown to be true, the implication is that the indirect PSO-based strategy is a suitable Web service composition alternative when prioritising the fast exploration of the search space.

5.1 Parameters

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 [2] and WSC-2009 [12] were used to compare the graph-based and PSO-based approaches, with 30 independent runs for each approach using each dataset. The choice of the parameters used for executing each approach, shown in Table 1, was based on common settings from the literature [13,8], with the two PSO approaches using the same settings.

6 Results

Results showing the execution time and solution fitness of the graph-based and the simple PSO-based approach are displayed in Table 2, where the first column

Table 1. Parameters used for experiments.

Graph-based		**PSO-based**	
Population size	500	Swarm size	30
Generations	51	Iterations	100
Fitness weights	0.25 (all)	Fitness weights	0.25 (all)
Mutation prob.	0.8	c_1	1.49618
Crossover prob.	0.1	c_2	1.49618
Reproduction prob.	0.1	w	0.7298
Tournament size	2		

indicates the dataset used, the second and third columns contain the mean execution time and mean fitness (respectively) of the PSO-based approach, and the fourth and fifth columns contain the time and fitness of the graph-based approach. All values are shown in 2dp, and means are accompanied by the standard deviation. A Wilcoxon signed-rank test at .95 confidence interval was conducted to ascertain whether the differences between the results for the two approaches are statistically significant, and the symbols $\uparrow$ and $\downarrow$ are used to indicate significantly larger and significantly smaller values, respectively. The results of the comparison between graph-based and the layered PSO-based approach are shown in table 3, which shares the layout described above.

Table 2. Execution time and fitness results for simple PSO and graph-based composition approaches.

Dataset	**Simple PSO**		**Graph-based**	
	Time (s)	**Fitness**	**Time (s)**	**Fitness**
WSC-08-1	2.1 ± 0.5	$0.49 \pm 1.18 \times 10^{-3}$	3.2 ± 0.4	$0.49 \pm 5.17 \times 10^{-5}$
WSC-08-2	4.1 ± 2.1	$0.59 \pm 1.30 \times 10^{-2}$	$2.6 \pm 0.4 \downarrow$	0.60 ± 0.00
WSC-08-3	24.7 ± 5.9	$0.49 \pm 2.89 \times 10^{-4} \uparrow$	$14.3 \pm 1.1 \downarrow$	$0.49 \pm 1.45 \times 10^{-4}$
WSC-08-4	16.4 ± 6.5	$0.51 \pm 2.56 \times 10^{-3}$	$6.1 \pm 0.6 \downarrow$	$0.51 \pm 1.24 \times 10^{-3}$
WSC-08-5	29.1 ± 8.2	$0.50 \pm 1.39 \times 10^{-4}$	$10.1 \pm 1.3 \downarrow$	$0.50 \pm 4.14 \times 10^{-5}$
WSC-08-6	195.6 ± 45.7	$0.50 \pm 1.24 \times 10^{-4} \uparrow$	$21.9 \pm 1.5 \downarrow$	$0.50 \pm 2.29 \times 10^{-5}$
WSC-08-7	202.4 ± 61.0	$0.50 \pm 3.81 \times 10^{-5} \uparrow$	$52.6 \pm 3.7 \downarrow$	$0.50 \pm 1.91 \times 10^{-5}$
WSC-08-8	539.0 ± 145.0	$0.50 \pm 9.19 \times 10^{-6}$	$75.2 \pm 13.8 \downarrow$	$0.50 \pm 2.19 \times 10^{-6}$
WSC-09-1	3.8 ± 1.3	$0.56 \pm 1.25 \times 10^{-2}$	3.2 ± 0.5	$0.57 \pm 9.91 \times 10^{-3}$
WSC-09-2	168.6 ± 38.8	$0.50 \pm 2.55 \times 10^{-5}$	$18.1 \pm 1.6 \downarrow$	$0.50 \pm 8.04 \times 10^{-6}$
WSC-09-3	260.0 ± 99.3	$0.51 \pm 2.19 \times 10^{-3}$	23.3 ± 0.6	$0.51 \pm 1.21 \times 10^{-3}$
WSC-09-4	1378.4 ± 577.5	$0.50 \pm 4.73 \times 10^{-5} \uparrow$	$65.1 \pm 2.9 \downarrow$	$0.50 \pm 1.03 \times 10^{-5}$
WSC-09-5	2124.0 ± 580.0	$0.50 \pm 1.05 \times 10^{-5}$	$151.1 \pm 17.8 \downarrow$	$0.50 \pm 5.31 \times 10^{-6} \uparrow$

6.1 Bar Charts

In order to evaluate the third comparison criterion, additional data were collected to reveal underlying patterns regarding the search behaviour of the two

Table 3. Execution time and fitness results for layered PSO and graph-based composition approaches.

Dataset	Layered PSO		Graph-based	
	Time (s)	**Fitness**	**Time (s)**	**Fitness**
WSC-08-1	0.3 ± 0.1	$0.49 \pm 1.19 \times 10^{-3}$	3.2 ± 0.4	$0.49 \pm 5.17 \times 10^{-5}$
WSC-08-2	$0.4 \pm 0.1 \downarrow$	$0.59 \pm 1.40 \times 10^{-2}$	2.6 ± 0.4	0.60 ± 0.00
WSC-08-3	$0.9 \pm 0.1 \downarrow$	$0.49 \pm 2.01 \times 10^{-4} \uparrow$	14.3 ± 1.1	$0.49 \pm 1.45 \times 10^{-4}$
WSC-08-4	0.5 ± 0.1	$0.51 \pm 7.20 \times 10^{-4}$	6.1 ± 0.6	$0.51 \pm 1.24 \times 10^{-3}$
WSC-08-5	$0.9 \pm 0.1 \downarrow$	$0.50 \pm 9.93 \times 10^{-5} \uparrow$	10.1 ± 1.3	$0.50 \pm 4.14 \times 10^{-5}$
WSC-08-6	$3.8 \pm 0.2 \downarrow$	$0.50 \pm 1.24 \times 10^{-4} \uparrow$	21.9 ± 1.5	$0.50 \pm 2.29 \times 10^{-5}$
WSC-08-7	$3.2 \pm 0.4 \downarrow$	$0.50 \pm 4.03 \times 10^{-5} \uparrow$	52.6 ± 3.7	$0.50 \pm 1.91 \times 10^{-5}$
WSC-08-8	$7.4 \pm 0.6 \downarrow$	$0.50 \pm 3.03 \times 10^{-5}$	75.2 ± 13.8	$0.50 \pm 2.19 \times 10^{-6}$
WSC-09-1	$0.4 \pm 0.1 \downarrow$	$0.57 \pm 1.73 \times 10^{-2}$	3.2 ± 0.5	$0.57 \pm 9.91 \times 10^{-3}$
WSC-09-2	$3.2 \pm 0.3 \downarrow$	$0.50 \pm 5.26 \times 10^{-5} \uparrow$	18.1 ± 1.6	$0.50 \pm 8.04 \times 10^{-6}$
WSC-09-3	$5.0 \pm 1.1 \downarrow$	$0.51 \pm 2.93 \times 10^{-3}$	23.3 ± 0.6	$0.51 \pm 1.21 \times 10^{-3}$
WSC-09-4	$21.0 \pm 3.3 \downarrow$	$0.50 \pm 5.08 \times 10^{-5} \uparrow$	65.1 ± 2.9	$0.50 \pm 1.03 \times 10^{-5}$
WSC-09-5	$11.9 \pm 2.1 \downarrow$	$0.50 \pm 1.30 \times 10^{-5}$	151.1 ± 17.8	$0.50 \pm 5.31 \times 10^{-6}$

techniques. More specifically, the frequency with which each atomic service appears in all solutions throughout the search/evolutionary process was tallied and displayed as a bar chart, sorted by frequency, with 0 indicating that a service appears in 0% of the solutions and 1 indicating 100% of them. This technique was based on the analysis performed in works in the area of feature selection [18,4], and it allows us to verify how often certain areas of the search space are visited and whether the two techniques visit certain atomic services with comparable frequencies. A high-level view of the bar charts produced by each technique is shown in Figure 7 for WSC-2008 datasets, and in Figure 8 for WSC 2009.

7 Analysis of Results

The fitness results presented in Tables 2 and 3 generally show that the solution fitness produced by the PSO-based approaches 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 for the simple PSO; 08-3, 08-5, 08-6, 08-7, 09-2, 09-4 for the layered PSO), whereas this is only the case for the graph-based approach when using dataset 09-5 (in comparison to the simple PSO). Thus, these results indicate that the PSO-based approaches are preferrable when the focus of the composition process is on the quality of the resulting solutions. This conclusion is supported by the bar graphs produced when running each dataset, which show that both PSO approaches explore the inclusion of different atomic services in a more "spread" way than the graph-based approach for all datasets. Namely, the frequencies with which services appear in graph-based solutions are either very high or very low, indicating that this approach converges to a small area of the search space relatively quickly. In the PSO-based approaches, on the other hand, the atomic

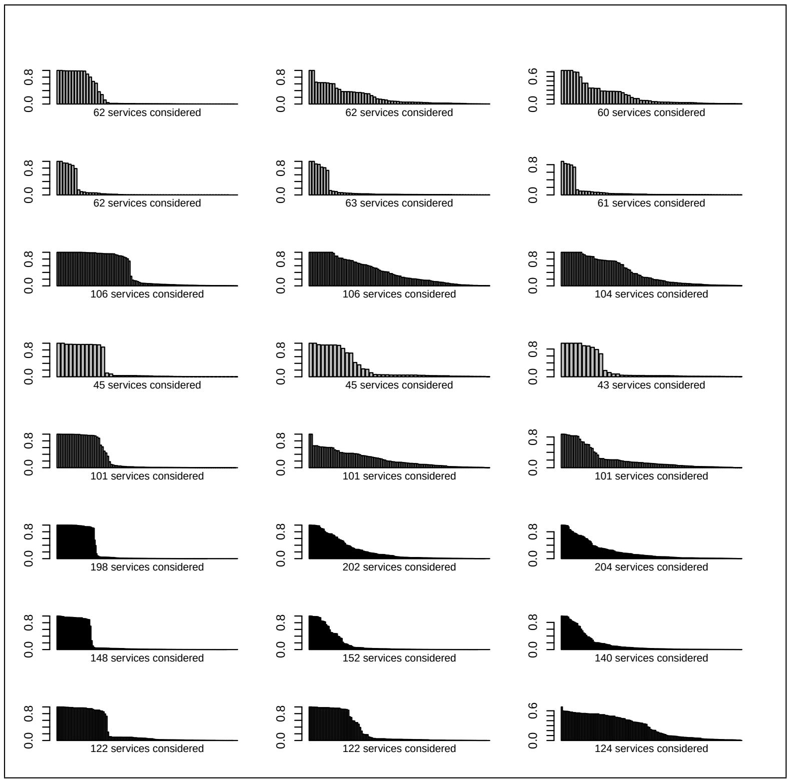

Fig. 7. Bar charts showing the frequency with which atomic services appear in solutions, for WSC 2008 datasets. Rows: datasets 1 to 8, from top to bottom. Columns: graph-based approach, simple PSO approach, and layered PSO approach, from left to right.

service frequencies lower gradually, showing that a larger area of the search space was considered.

Finally, with regards to the execution time of the two techniques, an interesting pattern is observed: while the time required by the simple PSO-based approach is consistently higher than that of the graph-based approach, the time required by the layered PSO-based approach 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 approach 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 approaches once each with WSC2008-8 (using the

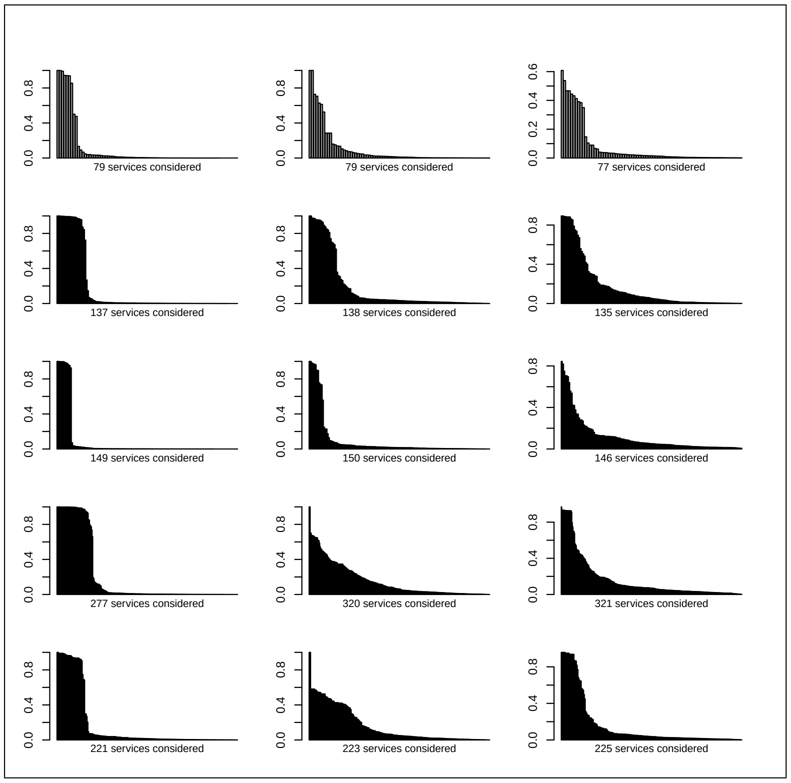

Fig. 8. Bar charts showing the frequency with which atomic services appear in solutions, for WSC 2009 datasets. Rows: datasets 1 to 5, from top to bottom. Columns: graph-based approach, simple PSO approach, and layered PSO approach, from left to right.

same settings as before) and counting how many times each of those approaches 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 Tables 2 and 3 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 amount of times the decoding algorithms check for potential connections during a run.

8 Conclusion

This work introduced two PSO-based QoS-aware Web service composition approaches 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 approaches 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 approaches 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, even though the layered PSO has a shorter execution time than the graph-based approach. Additionally, tracking of the appearance of atomic services in solutions throughout the run showed that both PSO methods explore the search space more effectively than their graph-based counterpart. 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 particle encodings.

References

1. Amiri, M.A., Serajzadeh, H.: Effective web service composition using particle swarm optimization algorithm. In: Telecommunications (IST), 2012 Sixth International Symposium on. pp. 1190–1194. IEEE (2012)
2. Bansal, A., Blake, M.B., Kona, S., Bleul, S., Weise, T., Jaeger, M.C.: Wsc-08: continuing the web services challenge. In: E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on. pp. 351–354. IEEE (2008)
3. Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial Intelligence 90(1), 281–300 (1997)
4. Butler-Yeoman, T., Xue, B., Zhang, M.: Particle swarm optimisation for feature selection: A hybrid filter-wrapper approach. In: Evolutionary Computation (CEC), 2015 IEEE Congress on. pp. 2428–2435 (May 2015)
5. 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)
6. 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)
7. 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)
8. 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)
9. Gottschalk, K., Graham, S., Kreger, H., Snell, J.: Introduction to web services architecture. IBM systems Journal 41(2), 170–177 (2002)
10. 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)
11. Jaeger, M.C., Mühl, G.: Qos-based selection of services: The implementation of a genetic algorithm. In: Communication in Distributed Systems (KiVS), 2007 ITG-GI Conference. pp. 1–12. VDE (2007)

12. 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)
13. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)
14. Lécué, F., Léger, A.: A formal model for semantic web service composition. In: The Semantic Web-ISWC 2006, pp. 385–398. Springer (2006)
15. 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)
16. Menasce, D.: QoS issues in web services. Internet Computing, IEEE 6(6), 72–75 (2002)
17. Milanovic, N., Malek, M.: Current solutions for web service composition. IEEE Internet Computing 8(6), 51–59 (2004)
18. Nguyen, H., Xue, B., Liu, I., Andreae, P., Zhang, M.: Gaussian transformation based representation in particle swarm optimisation for feature selection. In: Mora, A.M., Squillero, G. (eds.) Applications of Evolutionary Computation, Lecture Notes in Computer Science, vol. 9028, pp. 541–553. Springer International Publishing (2015)
19. 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)
20. 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)
21. 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)
22. Shi, Y., Eberhart, R.: A modified particle swarm optimizer. In: Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on. pp. 69–73. IEEE (1998)
23. 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)
24. Srivastava, B., Koehler, J.: Web service composition-current solutions and open problems. In: ICAPS 2003 workshop on Planning for Web Services. vol. 35, pp. 28–35 (2003)
25. 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)
26. Venkatraman, S., Yen, G.G.: A generic framework for constrained optimization using genetic algorithms. Evolutionary Computation, IEEE Transactions on 9(4), 424–435 (2005)
27. 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)
28. Wang, L., Shen, J., Yong, J.: A survey on bio-inspired algorithms for web service composition. In: IEEE 16th International Conference on Computer Supported Co-operative Work in Design (CSCWD). pp. 569–574. IEEE (2012)

29. 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)
30. 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)
31. 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)
32. 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)