Heuristic Approaches for QoS-Based Service Selection

Diana Comes, Harun Baraki, Roland Reichle, Michael Zapf, and Kurt Geihs

Distributed Systems Group, University of Kassel,
Wilhelmshöher Allee 73, 34121 Kassel, Germany
{comes,baraki,reichle,zapf,geihs}@vs.uni-kassel.de

Abstract. In a Service Oriented Architecture (SOA) business processes are commonly implemented as orchestrations of web services, using the Web Services Business Process Execution Language (WS-BPEL). Business processes not only have to provide the required functionality, they also need to comply with certain Quality-of-Service (QoS) constraints which are part of a service-level agreement between the service provider and the client. Different service providers may offer services with the same functionality but different QoS properties, and clients can select from a large number of service offerings. However, choosing an optimal collection of services for the composition is known to be an NP-hard problem.

We present two different approaches for the selection of services within orchestrations required to satisfy certain QoS requirements. We developed two algorithms, OPTIM_HWeight and OPTIM_PRO, which perform a heuristic search on the candidate services. The OPTIM_HWeight algorithm is based on weight factors and the OPTIM_PRO algorithm is based on priority factors. We evaluate and compare the two algorithms with each other and also with a genetic algorithm.

1 Introduction

Web Services constitute the most spreaded technology that overcomes interoperability issues between applications from different partners using the Internet as the underlined infrastructure. In order to be trusted by their clients, services have to guarantee the fulfillment of the required functionality and also of the expected non-functional properties, known as Quality of Service (QoS). Service clients and providers come to an agreement where providers assure that the service complies with the promised service levels. Web Services from different partners can be composed to a service orchestration and realize a complex business process. The de facto standard for executing business processes is the *Web Services Business Process Execution Language* (WS-BPEL) [7] which specifies how service orchestrations are built. It describes how the interaction between web services takes place in order to realize a business process. The business workflow is described via activities triggered in the order defined within the BPEL description file. The language offers standard activities for specifying control structures (like *while, repeatUntil, foreach* activities), conditional structures (like *if, switch*) and basic activities like *invoke* for calling a web service or the *assign* activity for assigning values to variables. A process may start with the arrival of a message (like with a *receive* or a *pick* activity). It may send an answer back to the requestor with a *reply* activity. Activities may be grouped inside a *sequence* (sequential invocation) or a *flow* (parallel invocation) structure.

While WS-BPEL specifies the workflow and functionality of a business process, Quality of Service is not part of the BPEL specification and so needs to be treated separately. In this paper we address QoS properties like response time, availability, reliability, and cost. The malfunction of one single service might cause the failure of the entire process. Since enterprises offer services having the same functionality but at different QoS levels, the QoS properties become the key differentiator between multiple services. Therefore, other service offerings need to be searched in the service registry and bound to the service process dynamically at runtime, which is where the selection algorithm comes into play. The goal of the selection algorithm is to find an optimal choice of services that realize the service orchestration. The orchestration needs to satisfy certain QoS constraints and to optimize an objective function which depends on the QoS properties of the involved services. However, as soon as we have to optimize for several independent QoS properties, finding the optimal combination from all possible service candidates that might realize the service orchestration leads to an NP-hard problem [11,9].

In this paper, we propose two heuristic algorithms for the selection of services in service orchestrations. The *OPTIM_HWeight* algorithm is based on weight factors and the *OPTIM_PRO* algorithm considers priority factors for performing the service search. The OPTIM_HWeight is an extension of the OPTIM_S algorithm utilizing a specific heuristic function $f_{HWeight}$. The OPTIM_S algorithm can perform a heuristic search, a local, and a global (brute-force) search by only modifying its parameters. We evaluate and compare the algorithms OPTIM_HWeight and OPTIM_PRO with each other and also with the genetic algorithm proposed by [2] which we call GA_CAN. Our experiments revealed that our OPTIM_PRO and OPTIM_HWeight algorithms perform better than the genetic algorithm GA_CAN and reach optimization values at least as good as GA_CAN. Our OPTIM_PRO algorithm is the fastest of the presented algorithms.

The paper is structured as follows: In Section 2 we describe the selection algorithms OPTIM_S, OPTIM_HWeight, OPTIM_PRO and GA_CAN. Section 3 describes the experiments we run in order to compare the algorithms with each other and measure their performance and results. In Section 4 we compare our heuristic approaches to the related works.

2 Service Selection Algorithms

A service orchestration is a composition of multiple services that are required in order to execute the orchestration. An *abstract service* represents the functionality of the desired service and we assume that there are several *concrete services* *(candidates)* that provide this functionality but have different QoS properties. The QoS of the entire process is computed from the QoS of the services that build up the service composition. Finding the optimal solution means selecting those services that satisfy the QoS requirements, including the QoS constraints, and optimizing an objective function for the entire orchestration. Assuming that we have n abstract services and each abstract service may have m concrete service realizations, we get a total of m^n possible combinations. As we have to assume that all QoS dimensions are independent, the whole optimization problem turns out to be NP-hard [11,9].

We define the set S_a that contains the set of abstract services, the set S_c containing the concrete services and the set QD of QoS dimensions. The set V represents the set of service variants, meaning the possible combinations of service candidates. The vector $Q = (a, l, r, c) \in \mathbb{R}^4$ contains the QoS values of the QoS dimensions *availability* (a), *reliability* (l), *response time* (r), and *cost* (c) computed for the service variants $v \subseteq V$. As an example we consider the following QoS requirements for the orchestration:

$$f_{obj}(Q) = \frac{k_1 \cdot a + k_2 \cdot l}{k_3 \cdot r + k_4 \cdot c} \tag{1}$$

$$\text{maximize } f_{obj}(Q)\ \forall v \subseteq \mathbb{V} \text{ realizing the service orchestration} \tag{2}$$

$$a > b_1,\ l > b_2,\ r < b_3,\ c < b_4, \text{where } b_i \in \mathbb{R} \tag{3}$$

The factors k_i, $i = 1 \ldots 4$ represent the weights for the $q \in QD$ variables depending on the user's preferences and b_i represent the bounds for the q variables. Within our approaches we address also non-linear objective functions, aggregation functions and constraints.

We developed the heuristic algorithms OPTIM_HWeight and OPTIM_PRO for the service selection problem. First, we will describe the tasks which are common for all our algorithms: (a) QoS aggregation and constraint checking, (b) creating the BPEL tree, and then we will present the specifics of the algorithms.

(a) QoS aggregation and constraint checking

The QoS of the service compositions depends on the QoS values of the services that build up the composition. The QoS value for a complex node (with children) is computed by the aggregation functions shown in table 1 (taken from [2]). The table is not complete but it provides examples for the structured activities *sequence*, *switch*, *flow* and *loop*. The aggregation formulas for the *switch* activity take into account the execution probabilities of the different branches to calculate the expected value for the quality dimensions. Still, the QoS constraints need to be fulfilled for every execution path, so for the *switch* activity we added an extra column (the last one) in the table where we consider the worst case of the QoS values over all of the branches of the *switch*. When we check the QoS constraints we use the aggregation formulas of the table (columns 2-4) and for the switch we take the formulas from the last column (*switch worst*). The QoS values of the candidate services are normalized to map the values onto the $[0, 1]$ interval. This is done with the formula adopted from [3]:

Table 1. Aggregation Functions

QoS Dimension	sequence	flow	loop	switch	switch worst
response time(r)	$\sum_{i=1}^{n} r_i$	$\max_{i \in 1..n} \{r_i\}$	$k \cdot r$	$\sum_{i=1}^{n} p_i \cdot r_i$	$\max_{i \in 1..n} \{r_i\}$
cost (c)	$\sum_{i=1}^{n} c_i$	$\sum_{i=1}^{n} c_i$	$k \cdot c$	$\sum_{i=1}^{n} p_i \cdot c_i$	$\max_{i \in 1..n} \{c_i\}$
availability (a)	$\prod_{i=1}^{n} a_i$	$\prod_{i=1}^{n} a_i$	a^k	$\sum_{i=1}^{n} p_i \cdot a_i$	$\min_{i \in 1..n} \{a_i\}$
reliability (l)	$\prod_{i=1}^{n} l_i$	$\prod_{i=1}^{n} l_i$	l^k	$\sum_{i=1}^{n} p_i \cdot l_i$	$\min_{i \in 1..n} \{l_i\}$

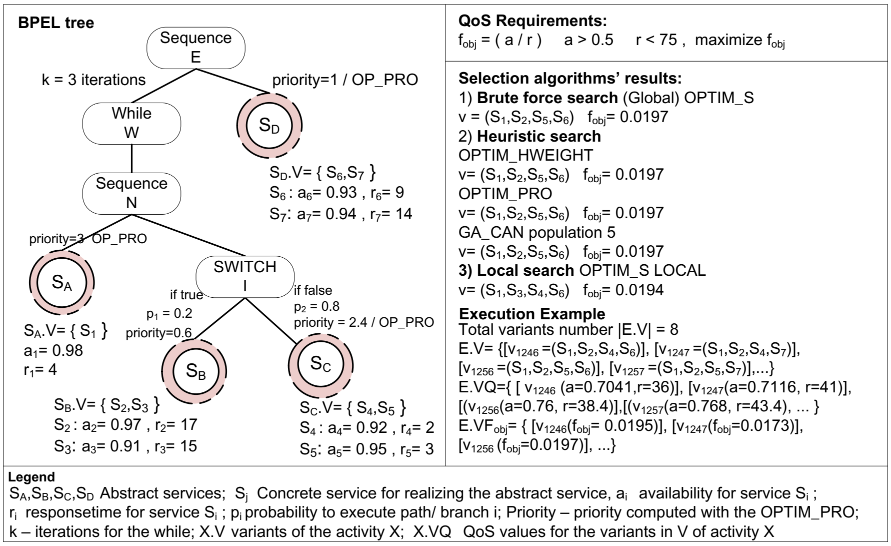
$$q' = \begin{cases} \frac{q - q^{min}}{q^{max} - q^{min}} & \text{if} \quad q^{max} - q^{min} \neq 0; \\ 1 & \text{if} \quad q^{max} - q^{min} = 0. \end{cases} \tag{4}$$

(b) Creating the tree

Both algorithms start with *creating a tree* out of the BPEL description file. The example in Figure 1 only serves to illustrate how our algorithms find a selection for a composition with four abstract services S_A, S_B, S_C, and S_D which can be realized by different concrete services. It does not show a realistic BPEL tree as this would be too complex here. The nodes of the BPEL tree contain the activities from the BPEL description file which are relevant for the execution of the BPEL process. It does not contain nodes for `partnerLinks`, for instance. We define S, the set of *simple element types* that contains the BPEL activities (e.g. `invoke`, `assign`) which we represent as leaves in the tree. The set C of *complex element types* contain the BPEL activities (e.g. `sequence`, `while`, `switch`) which we represent as inner nodes. Each node of the tree has the same structure and contains:

– the type $node.elem \in S \cup C$ of the node, the references *node.parent* to the parent node and *node.children* to the child nodes,
– the set of variants *node.V* containing combinations of the service candidates (e.g. $V = \{[v_1 = (S_2, S_4)], [v_2 = (S_2, S_5)]\}$),
– the set of QoS dimensions *node.QD* (e.g. $node.QD = \{a, l, c, r\}$),
– the set of QoS values *node.vQ* of variant v, where $v \in node.V$,
– the set *node.VQ* for all the variants $v \in node.V$ (e.g. $VQ = \{[v_1(c = 7, r = 3)], [v_2(c = 9, r = 4)]\}$), the objective values $node.VF_{obj}$ computed for all the variants $v \in node.V$ having the QoS values *node.VQ*, (The objective value of variant v is vF_{obj} and is equivalent to $f_{obj}(vQ)$, analogously we define the heuristic value $f_{Heu}(vQ)$).

After being created, the BPEL tree has to be initialized with probabilities p and iterations k obtained from the runtime monitoring of the BPEL process during multiple executions. The probability p_i appears for conditional activities like *switch* and corresponds to the probability for executing the branch i of the activity during an execution. The nodes representing a loop (e.g. *repeatUntil*,



QoS Requirements:
$f_{obj} = (a / r)$ $a > 0.5$ $r < 75$, maximize f_{obj}

Selection algorithms' results:
1) **Brute force search** (Global) OPTIM_S
$v = (S_1,S_2,S_5,S_6)$ $f_{obj}= 0.0197$
2) **Heuristic search**
OPTIM_HWEIGHT
$v= (S_1,S_2,S_5,S_6)$ $f_{obj}= 0.0197$
OPTIM_PRO
$v= (S_1,S_2,S_5,S_6)$ $f_{obj}= 0.0197$
GA_CAN population 5
$v= (S_1,S_2,S_5,S_6)$ $f_{obj}= 0.0197$
3) Local search OPTIM_S LOCAL
$v= (S_1,S_3,S_4,S_6)$ $f_{obj}= 0.0194$

Execution Example
Total variants number |E.V| = 8
E.V= {[v_{1246} =(S_1,S_2,S_4,S_6)], [v_{1247} =(S_1,S_2,S_4,S_7)],
[v_{1256} =(S_1,S_2,S_5,S_6)], [v_{1257} =(S_1,S_2,S_5,S_7)],...}
E.VQ={ [v_{1246} (a=0.7041,r=36)], [v_{1247}(a=0.7116, r=41)],
[(v_{1256}(a=0.76, r=38.4)],[(v_{1257}(a=0.768, r=43.4), ... }
E.VF_{obj}= { [v_{1246}(f_{obj}= 0.0195)], [v_{1247}(f_{obj}=0.0173)],
[v_{1256} (f_{obj}=0.0197)], ...}

Legend
S_A,S_B,S_C,S_D Abstract services; S_j Concrete service for realizing the abstract service, a_i availability for service S_i ;
r_i responsetime for service S_i ; p_i probability to execute path/ branch i; Priority – priority computed with the OPTIM_PRO;
k – iterations for the while; X.V variants of the activity X; X.VQ QoS values for the variants in V of activity X

Fig. 1. Service selection example

while) receive an iteration number k that represents the average number of iterations monitored for that loop.

2.1 OPTIM_S Algorithm

Since our *OPTIM_HWeight algorithm* is an extension of our *OPTIM_S algorithm* we will first describe the OPTIM_S algorithm. We developed the *OPTIM_S algorithm* with the intention to allow for easy adaptation to the runtime environment, depending on the number of services that are available at runtime. The OPTIM_S algorithm permits different types of search (local, global, and heuristic search) for the service selection by only changing its parameters. Thus the algorithm allows for adjusting between shorter computation time of the algorithm and better solution quality. The choice of the service selection algorithm should depend on the number of services searched and available at runtime. For example, when the selection targets only few abstract services (like in a sub-orchestration) a brute-force search is sufficient. In contrast, when the search is performed for the entire process which contains many services, a suitable optimization algorithm is needed. The choice of the selection algorithm may differ between set-up time and runtime. At runtime, a quick and effective solution is usually preferred to an optimal but slow strategy. All these requirements have been considered in the development of our selection algorithms. As inputs, the algorithm receives the BPEL tree, the QoS constraints, the objective function f_{obj} to be minimized or maximized, the maximum number of selected variants (nr_v) for a node, and optionally a heuristic function f_{Heu} used within the

```
Algorithm 1: OPTIM_S( tree, qosConstraints, f_obj, nr_v, optional f_Heu)

Input: tree- the BPEL tree ; qosConstraints – the QoS constraints, f_obj -objective function, f_Heu heuristic function, nr_v the maximal
number of variants selected for a node
Output: the selected services
Begin
  initializeTree( tree) // initialize the tree nodes with probabilities (p) and iterations (k)
  init(C,Sa) // initialize the set C with complex nodes and Sa with the abstract services
  repeat  // traverse the tree from bottom to top
      node ← treeTraverseBottomToTop(tree)
      if node.elem in Sa  // Sa- is an abstract service, a call to a service
          node.V ← node.getConcreteServices()
      else
        if node.elem in  C // node is a complex element with children
            children ← node.children
            childrenV  ← { }
            foreach child in children
                child.VQ ← aggregateQoS(child.V)
                // sort the variants with f_Heu and cut those that are too many
                child.V ← sortAndCut(child.V, child.VQ, f_Heu, nr_v)
                childrenV  ← childrenV U child.V
            endforeach
            node.V ← combine(childrenV)
            //compute QoS for each variant and eliminate those that don't meet QoS
            foreach v in node.V
              node.vQCons ←aggregateQoSCons(v) //aggregate with formulas for constraints check
              if  not (checkQoSConstraints(node.v, node.vQCons, qosConstraints))
                  node.V ←node.V \ v
              endif
            endforeach
        endif
      endif
  until node.parent = null
  node.VQ ← aggregateQoS(node.V)
  node.VFobj  ←  computeFobj(node.V, node.VQ, f_obj)
  node.V ← sort( node.V, node.VFobj)  // sort the variants by the value of fobj
return node.V[1]
End
```

Fig. 2. OPTIM_S Algorithm

selection process. The output of the algorithm will be the service selection fulfilling the QoS constraints and with the best found value for the objective function.

The basic idea of the algorithm is that for each of the nodes in the tree $(node.V)$ starting with the leafs, we select only a subset of the variants of the children of this node so that the size $|node.V|$ of the variants set is at most nr_v. For each variant the QoS is computed by using the aggregation functions in Table 1. With this QoS value (vQ) the heuristic value is computed by applying the heuristic function $(f_{Heu}(vQ))$. The variants are sorted according to this heuristic value and those variants with better values are selected and propagated to the parent node. An example for a heuristic function is the objective function itself.

The different search types *heuristic*, *local*, and *global search* can be switched by modifying the parameters of the OPTIM_S algorithm nr_v and f_{Heu}. The *global search* is a simple brute force search without any heuristic function. The algorithm is invoked by calling it with $nr\_v = \infty$ (in Java we take `Integer.MAX_VALUE`). After sorting the variants on the root node with f_{obj}, the variant $node.V[1]$ will be the global optimum. This kind of search is suited when the search is performed on a small number of services. The *heuristic search* is triggered by calling the algorithm with $\infty > nr\_v \geq 2$ and a given heuristic function. The discovered solution is not necessarily the optimal solution, but a good heuristic could provide a near-optimal or even an optimal solution. The *local search* is triggered by calling the algorithm with $nr\_v = 1$ so that only one variant is selected at each node. While this is the fastest way, the solution quality is expected to be worse than from the heuristic search.

We describe the algorithm on the basis of the pseudocode (see Fig. 2). The service selection starts with traversing the tree from the bottom to its root node (line 5). We distinguish between nodes that represent a service call and complex nodes. If the node represents a call to a service, the variants of the node *node.V* are initialized with the concrete services (lines 6-7). The number nr_v represents the maximum number of variants that are selected for each node. This allows us to restrict the size of the search space considerably. For each of the non-leaf nodes in the tree, the variants are selected from the variants of the child nodes (lines 9-16) so that the number of combinations of the child variants is at most nr_v. The heuristic function helps us to select those candidates which are likely to perform better in the process (line 15). At selection time the child variants are already sorted by their heuristic values computed with f_{Heu}. Furthermore, the selected child variants are combined with their siblings (see the *combine* method in line 18) on their parent node. The QoS of the variants is computed (line 21) using the formulas in Table 1. The variants are checked against the QoS constraints and those which do not fulfill the constraints are eliminated (lines 22-23). In the final step, the variants of the root node are sorted with regard to the objective function and the variant on top of the sorted list (*node.V[1]*) represents the service selection that has won the evaluation.

2.2 OPTIM_HWeight Algorithm

The OPTIM_HWeight algorithm makes use of the OPTIM_S algorithm, providing a specific heuristic function $f_{HWeight}$ which is used to rank and sort the candidate services/variants. OPTIM_HWeight is a probabilistic iterative algorithm with the heuristic function $f_{HWeight}$ at its heart. By virtue of this function, the candidate variants are sorted at each node according to their *influence* on the overall process, ascending from the leaves to the root, and only the best rated variants are kept. The heuristic function $f_{HWeight}$ is defined for an arbitrary node N as follows:

$$\begin{aligned} f_{HWeight}(\boldsymbol{W_N}, q_N^c, q_N^s) &= \boldsymbol{W_N} \cdot (\boldsymbol{q_N^c} - \boldsymbol{q_N^s}) = \nabla f_{obj} \cdot (\boldsymbol{q_N^c} - \boldsymbol{q_N^s}) \\ &= \left(\frac{\partial f_{obj}}{\partial q_1^N}, \frac{\partial f_{obj}}{\partial q_2^N}, \ldots, \frac{\partial f_{obj}}{\partial q_n^N} \right)^T \cdot (\boldsymbol{q_N^c} - \boldsymbol{q_N^s}) \qquad (5) \end{aligned}$$

with $\cdot$ being the scalar product, $\boldsymbol{q_N^s}$ the QoS vector of the current variant selection, $\boldsymbol{q_N^c}$ the QoS vector of the candidate variant, and the q_i^N being the QoS value of the node (aggregated value for complex nodes) of the i^{th} QoS dimension. We denote the difference vector $\boldsymbol{q_{dif}} = \boldsymbol{q_N^c} - \boldsymbol{q_N^s}$. We take the gradient ∇f_{obj} computed at node N as the weight vector $\boldsymbol{W_N}$ considering the QoS values of the

```
Algorithm 2: OPTIM_HWeight(tree, qosConstraints, $f_{obj}$, nr_v, n_iter, n_steps)
1: Procedure computeWeight(tree, variant) //computes the weights vector w for the heuristic $f_{HWeight}$
2: begin
3:  tree.InitToLastLeaf();
4:  repeat  //Aggregate QoS for variant variant from the bottom to the top of the tree
5:     node = treeTraverseBottomToTop(tree)
6:     node.v = node.getSubVariant(variant)//get the sub-variant v of the node that coresponds to variant
7:     node.vQ = aggregateQoS(node.v) //aggregate QoS values for v
8:   until (node.parent == null)
9:  repeat//propagate the QoS values from the top to the bottom of the tree and compute the weights
10:      node = treeTraverseTopToBottom(tree)
11:      foreach q in QD
12:          if(node == tree.Root)
13:               node.Weight(q) = partialDerivative($f_{obj}$, q, node.vq);
14:            else
15:               node.Weight(q) = node.parent.Weight(q) * partialDerivative(Agg(node.parent, q), node.siblings.vq, node.vq));
16:          endif
17:      endforeach;
18:  until (node == tree.LastLeaf)
19:  return tree.Weight;
20: end;
    Procedure OPTIM_HWeight ( tree, qosConstraints, $f_{obj}$, nr_v, n_iter, n_steps)
21: begin //initialize multiple random variants, optimize with OPTIM_S, and select the best one
22:  init(v_best)
23:  for i =  1 to n_iter do
24:     v_0 = randomVariants(tree,Sc) //select a random variant from the set of concrete services
25:     tree.Weight = computeWeight(tree, v_0)
26:     v_prev = v_0;
27:     for j=1 to n_steps do//iterative improvement of weights, iterative steps of the gradient ascent
28:         $f_{HWeight}$ = createFHWeight(tree.Weight, v_prev) //create the heuristic function $f_{HWeight}$
29:         v_s = OPTIM_S(tree, qosConstraints, $f_{obj}$, nr_v, $f_{HWeight}$)
30:         tree.Weight = computeWeight(tree,v_s)
31:         v_prev =v_s
32:     endfor
33:     if v_sFobj > v_bestFobj  //from the found Variants select the one that optimizes fobj
34:         v_best = v_s
35:     endif
36:  endfor
37: return v_best
38: end
```

Fig. 3. OPTIM_HWeight Algorithm

current selection. The result of the $f_{HWeight}$ function is used to deliver a score for the candidate service selection versus the current service selection; a higher value is ranked higher. The idea of this approach is essentially a *gradient ascent*. Given the current point represented by q_N^s, i.e. the QoS vector of the current selection, we calculate the gradient at that location. We try to find another variant which has "moved" from the point q_N^s in the direction of the gradient. As we have only discrete locations in our search space (the service selection variants) we can only choose another point with a minimal error. For this purpose, we compute the heuristic $f_{HWeight}$ as the scalar product between $\boldsymbol{W_N}$ and $\boldsymbol{q_{dif}}$. For the iteration we use the newly found point in the search space and retry (for n_steps).

In order to compute the derivatives for the gradient we have to consider that the objective function is a chain of aggregations, so we need to use the chain rule for partial differentiation. This can be explained by the fact that the QoS of a node in the tree is an aggregation of the QoS of its child nodes, and the QoS of each child node is again an aggregation of its child nodes, etc. For instance in Fig. 1, the tree has a root node E which has a child node W, and node W has a child node N which has a child S_A (abstract service) being a leaf. For the first quality dimension q_1 we may have the weight factor w_{S_A} computed at node S_A:

$$\begin{aligned} w_{S_A} &= \frac{\partial f_{obj}}{\partial q_1^{S_A}} = \frac{\partial f_{obj}(ag_E^{q_1}(ag_W^{q_1}(ag_N^{q_1}(q_1^{S_A}))))}{\partial q_1^{S_A}} = \frac{\partial f_{obj}}{\partial ag_E^{q_1}} \frac{\partial ag_E^{q_1}(ag_W^{q_1}(ag_N^{q_1}(q_1^{S_A})))}{\partial q_1^{S_A}} \\ &= \ldots = \frac{\partial f_{obj}}{\partial ag_E^{q_1}} \frac{\partial ag_E^{q_1}}{\partial ag_W^{q_1}} \frac{\partial ag_W^{q_1}}{\partial ag_N^{q_1}} \frac{\partial ag_N^{q_1}(q_1^{S_A})}{\partial q_1^{S_A}} \end{aligned} \tag{6}$$

where $ag_X^{q_1}$ denotes the aggregation function for node X with regard to the QoS dimension q_1. The partial derivatives can be efficiently computed when traversing the tree top-down because we only need to reuse the last computed value at parent node and multiply the inner derivative for the current node.

In the following we explain the pseudocode of the algorithm (see Fig. 3). The weight factors of $f_{HWeight}$ differ dependent on the nodes in the tree where the heuristic is evaluated. At each node in the tree we need the partial derivatives for each QoS dimension. The weight vector $\boldsymbol{W_N}$ is computed (procedure *computeWeight*, lines 1-20) by aggregating the QoS values of a random variant v from the bottom of the tree to the top. Through backpropagation of the influence of the QoS dimensions from top to the bottom of the tree we compute the weight factors $\boldsymbol{W_N}$ for $f_{HWeight}$. This is done by calculating the partial derivatives, which requires the aggregated QoS values of the current selection as input.

$OPTIM\_HWeight$ performs two iterative processes, the external loop (lines 23-36, with n_iter iterations) and the internal loop (lines 27-32, with n_steps iterations). The internal loop can be interpreted in two different ways: (1) it implicitly performs a gradient ascent with regard to the objective function (2) it iteratively improves the weight factors $\boldsymbol{W_N}$ of the heuristic function. Starting from an initial random variant v_0, the $\boldsymbol{W_N}$ vector is computed (procedure *computeWeight*) through bottom-up aggregation of the QoS of v_0 in the tree and backpropagation of the influence of the QoS dimensions from top to the leafs of the tree. Knowing the weight factors $\boldsymbol{W_N}$ and the QoS of v_0 the heuristic function $f_{HWeight}$ can be created (line 28). The function $f_{HWeight}$ is used inside the $OPTIM\_S$ algorithm (in the *sortAndCut* procedure, see $OPTIM\_S$ line 15) and calculates the scalar product between $\boldsymbol{W_N}$ and the difference vector $\boldsymbol{q_{dif}} = \boldsymbol{q_N^c} - \boldsymbol{q_N^{v\_0}}$, i.e. the difference of the QoS vector of the candidate variant at the actual node and the QoS vector of the current selection v_0. Now the OPTIM_S algorithm is called to find the (desired optimal) variant (v_s line 29) that optimizes the objective function. This selected variant (v_s) is considered in the next iteration step as starting variant to make a further improvement to the weights $\boldsymbol{W_N}$ and perform a further step of the gradient ascent. The computation of $\boldsymbol{W_N}$ and of the selection variant v_s starts again and the iterations continue until the n_steps iteration steps have been performed. In the external loop (line 23 - the for loop) all the steps described above are repeated n_iter times with different random variants as starting points for the inner loop. From the found variants during multiple iterations (i) the one with the best value for optimizing the objective function (lines 33-34) is finally selected.

Algorithm 3: OPTIM_PRO (tree, qosConstraints, f_{obj}, n_iter)

```
1:Begin
2: initializeTree( tree, p, k) // initialize the tree nodes with probabilities (p) and iterations(k)
3: init( root.v, C, Sa, vList)
4: foreach node.elem in Sa // node is an abstract service
5:     node.priority ← computePriority(tree) //compute priority = k * p
6:     SaSet ← SaSet U node
7: endforeach
8: SaSet ← sortByPriority(SaSet , 'descendent')
9: i ← 0
10: NEXT while (i < n_iter)
11:     i ← i + 1;
12:     foreach sa in SaSet
13:        c ← 0; roottmp ← root
14:        foreach sc in sa.V // sc is a concrete service that realizes the abstract service sa
15:           c ← c + 1
16:           roottmp.v(sa) ← sc //sc replaces the old candidate service of sa, in the root copy variant
17:           if (checkQoSConstraintsAggregate2 (roottmp.vQ, qosConstraints))
18:                 roottmp.vQ ← aggregateQoS1 (roottmp.v)
19:                 roottmp.vFobj ← computeFobj (roottmp.vQ)
20:                 if (roottmp.vFobj > root.vFobj) OR ((i==1) AND(c==1))
21:                    //optimizing obj function OR first iteration, first candidate service
22:                     root ← roottmp
23:                 endif
24:              endif //else select the variant with the minimal distance to fulfill constraints
25:        endforeach
26:        if (root.v in vList )
27:              root.v ← chooseForAllRandomServices() //root.v receives random service candidates
28:              continue NEXT
29:           else
30:              vList ← vList U root.v   //save the root variant in the root variants list, vlist
31:        endif
32:     endforeach
33: endwhile
34: sortByFobj(vList) //sort the variants list by their objective values
35: return vList[1] //return the best variant
36: End
```

Fig. 4. OPTIM_PRO Algorithm

2.3 OPTIM_PRO Algorithm

The OPTIM_PRO heuristic algorithm calculates *priority factors* and uses the objective function to sort the variants. It is described in pseudocode in Figure 4. During monitoring of the execution, the nodes in the tree receive an iteration number k and a probability p as explained previously. Each of the nodes that represent an abstract service (Sa, lines 4-7) receive a priority as a product of k and p and the node is added to the set of abstract services $SaSet$. The *priority factor* states that those nodes which are executed more often should receive a higher priority. The algorithm proceeds with sorting the nodes from the $SaSet$ (line 8) in descendent order of the computed priorities such as the nodes with higher priorities are processed first. OPTIM_PRO is an iterative algorithm which improves the found variant (the objective value) of the root node with each iteration (lines 10-33) step. After selecting a variant for the root node in the first iteration, this variant is going to be improved during the next iterations. A copy of the root is created (*roottmp*, line 13) in order to check if the currently selected service candidate (*sc*) is an improvement to the objective function. In the root copy variant, the currently selected service candidate replaces the old service candidate. With this new service candidate value, the

QoS value of the root copy variant is aggregated ($roottmp.vQ$), checked against the constraints, and the objective function is computed ($roottmp.vFobj$). If the objective function yields a better value the root receives the value of its copy ($roottmp$), otherwise it remains the same. The variants that no longer can be improved are saved into the list *vlist*. When this is the case, the root variant receives random candidate services and the iterative process continues in the same way as described above. After reaching the maximal iteration number (n_iter), the iterative process stops. The list *vlist* that contains the found variants is sorted due to their objective values. The first element in the list is returned as being the best variant that was found for optimizing the objective function.

2.4 GA_CAN Algorithm

In order to compare our heuristic algorithms we implemented a genetic algorithm as proposed by Canfora *et al.* [2]. Genetic algorithms are inspired by biology and use meta-heuristics in optimization problems. The reason why we chose this algorithm is because it can also be applied to non-linear functions and constraints, which is also our target. For more details we recommend [2]. The genome represents the service variants that realize the service orchestration and is encoded as an array. The length of the genome is equal to the number of abstract services. Each element within the array contains a reference to the list of the concrete candidate services that may realize the abstract service. The initial population is built with random individuals. The fitness of the individuals represents their utility as a solution and is computed using the fitness function defined in equation 7. It corresponds to the sum of the objective function calculated on the genome and the weighted distance $D(g)$ (multiplied with the penalty factor k_5) resulting from constraints satisfaction checking. This means that those individuals that do not fulfill the constraints are penalized with distance $D(g)$. Assuming that there are h missed constraints, the distance $D(g)$ is defined as the sum of all the deviations from each of the missed constraints.

$$f_{fit}(g) = f_{obj}(g) + k_5 \cdot D(g) \quad \text{with} \quad D(g) = \sum_{i=1}^{h} dev_i \tag{7}$$

We build multiple generations over the population in an iterative way by applying the mutation and crossover operators. Through the mutation operator, the candidate services are varied randomly and an arbitrary concrete service is selected to realize the abstract service. The crossover operator combines service variants of different individuals. The algorithm stops when during multiple generations there is no improvement to the fitness function value.

3 Evaluation

Several experiments have been performed in order to compare our algorithms OPTIM_HWeight and OPTIM_PRO with the GA_CAN algorithm proposed

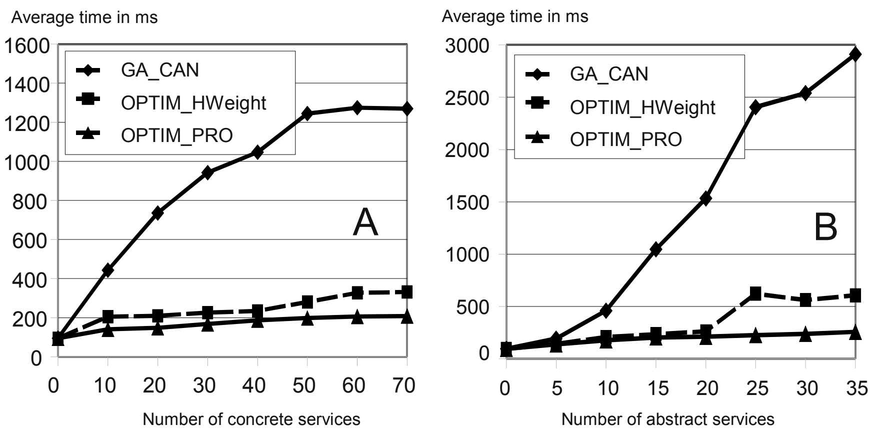


Fig. 5. Computation time OPTIM_HWeight, OPTIM_PRO and GA_CAN

in [2]. We have evaluated the three algorithms with regard to the required computation time and the optimization of the objective function. All tests have been performed on a Lenovo R60, 1.83 GHz, 2 GB RAM with Windows XP SP3 and JSDK 1.6.

As baseline for our experiments, we have randomly generated 10 different BPEL trees for each test case, with different structures and dimensions. The tree structures have been created in such a way that they contain the relevant BPEL activities, like *while*, *if*, *invoke*, *sequence* and *flow* with adjustable probabilities.

For the GA_CAN algorithm we set the mutation probability to 0.01 and the crossover probability to 0.7. The OPTIM_HWeight algorithm was triggered with $nr\_v = 12$ and $n\_iter = 12$. Since all the algorithms are probabilistic, we executed the algorithms 10 times (for each of the 10 BPEL trees, having 100 test runs in total) and took the average value. The results of our measurements are presented in Figures 5 and 6.

In our experiments A and B (Fig. 5, Fig. 6 Table A and B) we compared the computation time of OPTIM_HWeight and OPTIM_PRO with the computation time of the GA_CAN algorithm (with a population of 100) for reaching approximately the same value (difference less than 0.01 %) for the objective function.

In experiment A (Fig. 5, Fig. 6 Table A) we used a fixed number of abstract services (15) and measured how an increasing number of concrete services (from 10 to 70 per abstract service) influences the computation time. The results show that OPTIM_PRO is the fastest algorithm, requiring on average about 19% of the time of GA_CAN for reaching approximately the same optimization of f_{obj}. We observed that in average, our OPTIM_HWeight algorithm requires only about 28% of the time of the GA_CAN.

Experiment B is similar to experiment A, but this time we increased the abstract services from 0 to 35 while keeping the number of concrete services

Table A

Nr. Con. Serv	GA_CAN (G) Avg. timeG P100,[ms]	HWeight (H) Avg. timeH [ms]	Avg. timeH/ timeG	O_PRO (P) Avg. timeP [ms]	Avg. timeP/ timeG
10	443,10	205,10	46,29%	140,80	31,78%
20	736,10	210,10	28,54%	148,50	20,17%
30	942,20	225,80	23,97%	167,80	17,81%
40	1046,50	234,80	22,44%	186,90	17,86%
50	1244,00	280,10	22,52%	198,10	15,92%
60	1274,10	327,30	25,69%	206,50	16,21%
70	1269,00	331,30	26,11%	208,20	16,41%

Table C

Nr. Abs. Serv	Fobj_H/ Fobj_G	Fobj_P/ Fobj_G
5	100,00%	100,00%
10	101,22%	101,22%
15	100,01%	100,01%
20	103,29%	103,36%
25	110,78%	110,94%
30	107,31%	107,31%
35	107,10%	107,31%

Table B

Nr. Abs. Serv	GA_CAN Avg. timeG [ms]	HWeight (H) Avg. timeH [ms]	Avg. timeH/ timeG	O_PRO (P) Avg. timeP [ms]	Avg. timeP/ timeG
5	192,10	142,60	74,23%	136,80	71,21%
10	459,30	206,30	44,92%	176,10	38,34%
15	1046,50	234,80	22,44%	203,60	19,46%
20	1532,80	261,20	17,04%	210,40	13,73%
25	2404,70	621,80	25,86%	226,00	9,40%
30	2539,25	561,42	22,11%	238,30	9,38%
35	2909,92	606,25	20,83%	257,20	8,84%

Fig. 6. Algorithms evaluation: GA_CAN (G), OPTIM_PRO (P), OPT_HWeight (H)

constantly at 40. Experiment B (Fig. 5, Fig. 6 Table B) shows again that OPTIM_PRO is the fastest algorithm and needed in average about 24% of the GA_CAN time while the OPTIM_HWeight needed about 32% of the GA_CAN time.

In experiment C (Fig. 6 Table C) we evaluated how well the objective function was optimized by the different algorithms. We computed the value of f_{obj} reached by OPTIM_HWeight, OPTIM_PRO and GA_CAN, where the computation time limit for all of them was set to 4 seconds. The population size of GA_CAN during the evaluation was varied between 100 and 600 and the best result was chosen. The evaluation shows that our OPTIM_HWeight and OPTIM_PRO provides an optimization value at least as good as GA_CAN. With an increasing number of abstract services (the number of concrete services/possible realizations per abstract service is constantly 100) our algorithms provide even better optimization results than GA_CAN (e.g. above 25 abstract services, f_{obj} is about 7% better). Thus, according to our evaluation, the more possible combinations exist, the better are the optimization results of OPTIM_HWeight and OPTIM_PRO in comparison to GA_CAN.

4 Related Work

Zeng et al. describe in [3] a "QoS-Aware Middleware for Web Service Composition". For the service selection the authors describe two approaches for local and global optimization. In their global planning approach, the authors propose an Integer Programming (IP) solution, assuming that the objective function, the constraints, and the aggregation functions are linear. The multiplicative aggregation functions for availability and success rate are linearized by applying the logarithm. Our goal was to also consider non-linear objective functions and aggregation functions within both our heuristic algorithms. Due to the bad runtime performance of IP, their approach quickly becomes unfeasible when confronted with an increasing number of tasks and concrete services. An improvement of IP in respect to runtime performance is achieved in [6] by using relaxed integer programming and a backtracking algorithm. They have the same restrictions like [3] and address only sequential web service compositions and linear objective functions. In our approach we consider sequential and also parallel execution of activities and non-linear objective functions. In particular, OPTIM_HWeight utilizes gradient ascent and we perform the optimization on a tree.

Canfora et al. [2] propose a genetic approach for the selection problem. Since the authors consider also non-linear objective functions, we implemented their approach (within the GA_CAN algorithm) to compare it with our heuristic algorithms. The runtime behaviour was discussed in the previous section. We used the same aggregation functions as Canfora did. The authors select the genome with a fitness function containing a penalty factor. This penalty factor does not guarantee that the individuals will fulfill the QoS constraints for all possible execution paths. By checking the QoS constraints considering the worst case of QoS values throughout multiple branches, our algorithms ensure that the QoS constraints are met for all of the execution paths. In addition, our OPTIM_HWeight and OPTIM_PRO need less time than GA_CAN to find the same result or even a better one. An amendment of the genetic approach is introduced in [10] where the improvement is achieved by the usage of hybridization. The neighborhood of each individual of the genetic algorithm is explored iteratively to replace the actual individual with the best or the almost best neighbor. Still, this reduces the diversification of the population or the number of generations if the computation time shall not be increased. Thus, the authors come to the conclusion that for bigger problem instances the basic genetic algorithms perform better.

In [5] different heuristics are evaluated, including approaches which consider global constraints and obtain almost best possible QoS like the pattern-wise selection or the discarding subsets approach. Nevertheless the runtime performance can make them unsuitable for a growing number of tasks. Other heuristics like greedy selection or the bottom-up approximation proposed in [5] result in a loss of QoS up to 5% but lead to an acceptable runtime performance. In this context, especially our OPTIM_HWeight and OPTIM_PRO algorithms can provide an improvement in relation to shorter runtimes than *pattern-wise selection* and *discarding subsets*, and delivers a better QoS than greedy selection and bottom-up approximation. Besides, the user can configure the runtime-to-QoS ratio by setting few parameters like number of steps of OPTIM_HWeight or *nr_v* for OPTIM_S.

5 Conclusion

We presented two heuristic algorithms as solutions to the service selection problem in Web service orchestrations: the OPTIM_HWeight algorithm based on weighting factors inspired by gradient ascent approaches and the OPTIM_PRO algorithm utilizing priority factors. Both algorithms are iterative improving the found solution by every iteration step, and provide an easy way to trade computation time against the quality of the solution by merely changing their parameters. As our target was to optimize non-linear objective functions, we compared OPTIM_HWeight and OPTIM_PRO with the genetic algorithm GA_CAN proposed by Canfora [2]. Our experiments revealed that our OPTIM_PRO and OPTIM_HWeight are faster than GA_CAN (in average they needed about 22%, respectively 30% of the time of GA_CAN) and even achieve better values for the objective function (in our experiments up to 7% better) than GA_CAN in cases with a high number of combinations. The OPTIM_PRO algorithm turned out to be the fastest algorithm. In our future work we will also consider a combination of both algorithms, like having the solution of OPTIM_PRO as starting point for OPTIM_HWeight.

References

1. Bleul, S., Comes, D., Geihs, K.: Automatic Service Brokering in Service oriented Architectures, Homepage, `http://www.vs.uni-kassel.de/research/addo/`
2. Canfora, G., Penta, M., Esposito, R., Villani, M.L.: An approach for QoS-aware service composition based on genetic algorithms. In: Proceedings of the 2005 conference on Genetic and evolutionary computation, ACM, Washington (2005)
3. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-Aware Middleware for Web Services Composition. In: IEEE Transactions on Software Engineering, pp. 311–327. IEEE Press, Los Alamitos (2004)
4. Comes, D., Bleul, S., Weise, T., Geihs, K.: A Flexible Approach for Business Processes Monitoring. In: Senivongse, T., Oliveira, R. (eds.) DAIS 2009. LNCS, vol. 5523, pp. 116–128. Springer, Heidelberg (2009)
5. Jaeger, M., Múhl, G., Golze, S.: QoS-aware Composition of Web Services: An Evaluation of Selection Algorithms. In: International Symposium on Distributed Objects and Applications (DOA 2005), Springer, Heidelberg (2005)
6. Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for QoS-aware Web Service Composition. In: IEEE International Conference on Web Services (ICWS 2006), IEEE Computer Society, Los Alamitos (2006)
7. Web Services Business Process Execution Language Version 2.0, OASIS standard (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
8. Oracle BPEL Process Manager (2008), `http://www.oracle.com/technology/products/ias/bpel/index.html`
9. Baligand, F., Rivierre, N., Ledoux, T.: A Declarative Approach for QoS-Aware Web Service Compositions. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, Springer, Heidelberg (2007)
10. Parejo, J., Fernandez, A., Cortes, P., QoS-Aware Services, A.: composition using Tabu Search and Hybrid Genetic Algorithms. Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos 2(1) (2008)
11. Garey, M., Johnson, D.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W.H. Freeman, New York (1979)