Web Service Composition by Optimizing Composition-Segment Candidates

Fang-Yuan Zuo, Ze-Han Shen, Shi-Liang Fan, and Yu-Bin Yang(✉)

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
yangyubin@nju.edu.cn

Abstract. Web service composition has been increasingly challenging in recent years due to the escalating number of services and the diversity of task objectives. Despite many researches have already addressed the optimization of multiple Quality of Service (QoS) attributes, most of the currently available methods have to build a large web service dependency graph, which may incur excessive memory consumption and extreme inefficiency. To address these issues, we present a novel web service composition method by optimizing composition-segment candidates. Firstly, we formalize the web service composition problem as a Mixed-Integer Linear Programming (MILP) model and introduce some effective techniques for complex cases, and then a standard solver can be applied to this model. Afterwards, a candidate optimization method is proposed to solve the MILP model efficiently, which runs sharply fast without building a web service dependency graph. Experimental results on both Web Service Challenge 2009's datasets and substantial datasets randomly generated show that the proposed method outperforms the state-of-art while achieving a much ideal tradeoff among all the objectives with better performance.

Keywords: Web service composition · Optimization · MILP

1 Introduction

In service-oriented environments, many complex applications can be described as a series of processes invoking services selected at runtime. Thus, the web service composition problem has been widely studied [1,2]. Generally, many researchers aim at optimizing a single global QoS [3] by searching for a solution in a huge web service dependency graph [4]. When there are more than two objectives, i.e., QoS attributes considered, these methods usually fail to output a satisfactory solution. These QoS attributes are usually conflicted with each other, which makes it difficult to find a solution optimal for all the QoS attributes. Another shortcoming of these methods is that seeking a near-optimal solution in a huge dependency graph consumes much time and memory.

To address these issues, we aim at finding an ideal tradeoff among all the objectives without building a web service dependency graph. We formalize the

Q. Wang et al. (Eds.): SCC 2020, LNCS 12409, pp. 20–34, 2020.
https://doi.org/10.1007/978-3-030-59592-0_2

web service composition problem as a MILP model and apply a standard solver to it for a near-optimal solution. Furthermore, we present a candidate optimization method for a better tradeoff, which does not require any web service dependency graph. The main contributions of this paper are summarized as follows.

- We formalize the web service composition problem as a novel MILP model, which transforms min-max constraints into linear constraints by introducing integer variables.
- A standard solver is applied to the MILP model and outputs a near-optimal solution in most cases, and some effective techniques are introduced for complex cases.
- A candidate optimization method is proposed to solve the MILP model efficiently and obtains a composition with better tradeoff among all the objectives.

To validate the methods proposed in this paper, we carry out extensive experiments on both WSC-2009's datasets and randomly generated datasets.

The rest of this paper is organized as follows. Section 2 introduces the background and some related work. Section 3 formalizes the web service composition problem into a MILP model and provides some practically useful techniques. Section 4 proposes a candidate optimization method with no need to build a web service dependency graph. Section 5 presents the experimental results, and Sect. 6 provides the final remarks.

2 Background and Related Work

2.1 Background

The formal definition of web service is shown as follows.

Definition 1. *Giving a set of concepts C (the size of C is $|C| = m$), we define a Web Service ("service" for short) as a tuple $s_i = \{I_i, O_i, R_i, T_i\}$, where $I_i = \{i_1, \ldots, i_p\}$ is the subscript set of inputs required to invoke the web service s_i and $O_i = \{o_1, \ldots, o_q\}$ is the subscript set of outputs generated by invoking service s_i. Each element $c_j, j \in I_i \cup O_i$ is a semantic concept belonging to the set C, namely, $\{c_j | j \in I_i\} \subseteq C$ and $\{c_j | j \in O_i\} \subseteq C$. R_i and T_i are the nonfunctional attributes which are the measures for judging how well the service s_i serves the user.*

Obviously, services are not independent to each other. Relevant services can be combined by connecting matched inputs and outputs to construct compositions.

Lemma 1. *Giving an output c_o of a service s_i, as well as an input c_i of another service s_j, if c_o and c_i are equivalent concepts or c_o is a sub-concept of c_i, c_o matches c_i.*

Each service has its own QoS, which contributes to the global QoS of a composition. The definition of QoS of a web service composition is dependent on the structure of composition. There are two main kinds of structures, named *sequential structure* and *parallel structure*. The first one means the services are invoked in order, while the second one means they are invoked synchronously.

Definition 2. *A composition containing the set of services* $S = \{s_1, \ldots, s_n\}$ *is defined as* Ω*. If the services are chained in sequence, the composition is expressed as* $\Omega^{\rightarrow} = s_1 \rightarrow \ldots \rightarrow s_n$*; if in parallel, it is expressed as* $\Omega^{\parallel} = s_1 \parallel \ldots \parallel s_n$*. The set of services involved in* Ω *is defined as* $Servs(\Omega) = S$*. Moreover, the length of a composition* Ω *is defined as* $Len(\Omega) = |S|$*, namely, the number of services in* Ω*. Taking the response time as an example, we compute the global QoS of* Ω *as follow.*

$$\left.\begin{array}{r} RT(\Omega^{\rightarrow}) = \sum_{i=1}^{n} RT(s_i), s_i \in S \\ RT(\Omega^{\parallel}) = \max_{1 \leq i \leq n} RT(s_i), s_i \in S \end{array}\right\} \quad (1)$$

where $RT(\Omega)$ *represents the global response time of the composition and* $RT(s)$ *represents the same of services* s*. Another QoS attribute is throughput, which can be defined as follows:*

$$\left.\begin{array}{r} TP(\Omega^{\rightarrow}) = \min_{1 \leq i \leq n} TP(s_i), s_i \in S \\ TP(\Omega^{\parallel}) = \min_{1 \leq i \leq n} TP(s_i), s_i \in S \end{array}\right\} \quad (2)$$

where $TP(\Omega)$ *and* $TP(s)$ *represent the global throughput and the service throughput similarly.*

Based on the above concepts, Multi-Objective Web Service Composition can be described as Definition 3.

Definition 3. *Giving a web services set* S*, a concepts set* C *and a given composition request* $R = \{In_R, Out_R\}$*, we define Multi-Objective Web Service Composition as finding a composition* Ω *which archives an ideal tradeoff among* $Len(\Omega), RT(\Omega)$ *and* $TP(\Omega)$*.*

2.2 Related Work

In this subsection, we introduce some related works about single objective and multi-objective web service composition. Meanwhile, we point out their main drawbacks at last.

2.2.1 Single Objective Web Service Composition

For the single objective web service composition, the most popular objective is the number of services in the final composition.

A heuristic A^* search algorithm was proposed in [5] for web service composition, which used A^* search algorithm in a dependency graph. Noting that some useless services might exist in the final composition, Xia et al. [6] proposed an algorithm to remove the useless service, which was useful to reduce the number

of services. Fan et al. [4] transformed the web service composition problem into a dynamic knapsack problem and applied dynamic programming technique on it, which obtained a solution containing a small number of services.

Single objectives web service composition fails to meet the requirements in many applications. Therefore, many researchers pay more attention to multi-objective service composition of which goal is to find a proper composition achieving an ideal tradeoff among all the objectives.

2.2.2 Multi-objective Web Service Composition

Graphs are natural and intuitive ways to express the complex interaction relations between entities. The web service dependency graph is useful to illustrate the multi-objective web service composition problem. In Fig. 1, a web service composition problem is shown as a layered directed graph. The composition request is $R = \{\{in_1, in_2, in_3\}, \{out_1, out_2, out_3, out_4\}\}$. Each rectangle in the graph represents a web service. The response time and throughput of a web service are shown in the above and below, respectively. Each circle represents an input or an output of a service. In addition, the edges connecting circles and rectangles denote the matching relations between them. Two dummy service S_i for the inputs and S_o for the outputs are added in the graph, whose response time and throughputs are 0 ms and $+\infty$ inv/s respectively.

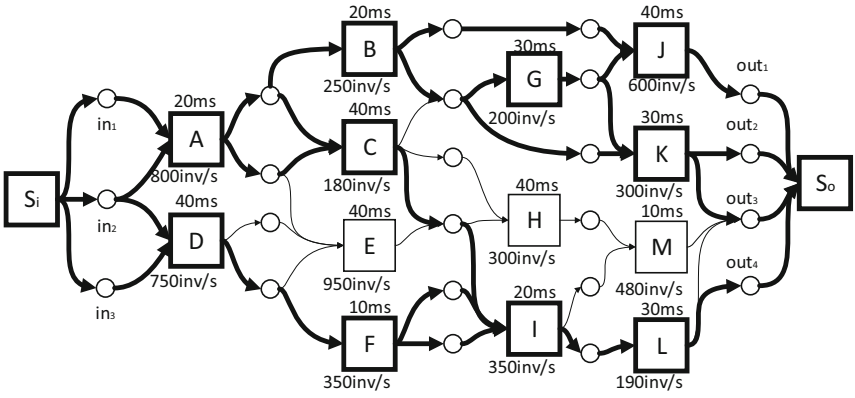


Fig. 1. An example of a service dependency graph.

As shown in the Fig. 1, there are many compositions with different QoS and numbers of services satisfying the request R. The composition highlighted in the graph $\Omega = S_0 \rightarrow (A||D) \rightarrow (B||C||F) \rightarrow (G||I) \rightarrow (J||K||L) \rightarrow S_o$ is the optimal with response time of 110 ms. In addition, the throughput of Ω is 180 inv/s, which is not optimal. Moreover, another composition $\Omega' = S_i \rightarrow (A||D) \rightarrow (B||E||F) \rightarrow (G||I) \rightarrow (J||K||L) \rightarrow S_o$, has a response time of 130 ms, a throughput of 190 inv/s and the same length of 12. On the one hand, the response time of Ω is shorter in comparison with the one of Ω'. On the another hand, the throughput of Ω is less than Ω'. Although both Ω and Ω' are Pareto

optimal solutions, we prefer the former since $TP(\Omega')$ changes little from $TP(\Omega)$ (180 inv/s to 190 inv/s), while the response time of Ω has been greatly improved (130 ms versus 110 ms).

To deal with the above problems, Zeng et al. [7] directly transformed the multi-objective service composition into single-objective optimization and used traditional techniques to solve it. Furthermore, some researchers applied a systematic search algorithm like Dijkstra's algorithm with the same single-objective function [8], which generated many solutions and recorded the best one until no more concepts could be generated. Another important objective is the number of services in the resulting composition, which is necessary to consider for conducting services composition. Fan et al. [9] used a Knapsack-Variant algorithm with transforming multi-objectives into one loss objective computed dynamically. However, these methods have to build a huge dependency graph explicitly, which leads to a long composition time, especially in an enormous number of services situation.

3 MILP Formalization of Web Service Composition

In this section, the problem of web service composition is formalized as a MILP model. Once a MILP model of web service composition is obtained, some standard solvers such as groubi [10], can be applied to it and output a well enough composition.

3.1 Notations and Variables

Given a composition request $R = \{In_R, Out_R\}$, two dummy services s_0, s_{n+1} named the input service and the output service, are added to the model, which represent the input and output of the request respectively. Some related notations are defined in Table 1. The constants R_{min} and R_{max} are minimum and maximum response time and so do T_{min} and T_{max} for throughput.

Table 1. Some notations in this paper

Name	Notation	Description
Service set	S	$S = \{s_0, \ldots, s_{n+1}\}$
Concept set	C	$C = \{c_1, \ldots, c_m\}$
Service input	I_i	The input set of s_i is $\{c_j \mid j \in I_i\}$, $I_0 = \varnothing, I_{n+1} = Out_R$
Service output	O_i	The output set of s_i is $\{c_j \mid j \in O_i\}$, $O_0 = In_R, O_{n+1} = \varnothing$
Response time	R_i	The response time of s_i and $R_0 = R_{n+1} = R_{min}$
Throughput	T_i	The throughput of s_i, $T_0 = T_{n+1} = T_{max}$ specially

For a formal description, we introduce some variables optimized by standard solver in Table 2. In the composition context of this paper, the term *response time* is treated as *generated time* of a concept or *invoked time* of a service.

Table 2. The variables in MILP model

Notation	Range	Description
x_i	$\{0,1\}$	$x_i = 1$ means s_i is selected
y_j	$\{0,1\}$	$y_j = 1$ means c_i is generated
sr_i	$[0,+\infty)$	The time when s_i has been invoked
r_j	$[0,+\infty)$	The time when c_j is generated at first time
st_i	$[0,+\infty)$	The throughput of s_i in the composition
t_j	$[0,+\infty)$	The throughput of c_j in the composition

3.2 Criteria

Taking response time, throughput, and number of services into consideration, we can formalize the criteria of this MILP model as follows:

$$\max_{x,y,sr,r,st,t} st_{n+1} - \alpha \sum_{i=1}^{n} x_i - \beta sr_{n+1} \tag{3}$$

where α and β are weights of different single objectives, and they can be assigned flexibly to adapt to the preference of user.

3.3 Constraints

Without building a huge dependency graph, we add some constraints to the proposed MILP model, which guarantees that a solution of the MILP model is also a valid web service composition.

3.3.1 Input and Output Constraints

For the input and output services, they must be invoked:

$$x_0 = x_{n+1} = 1 \tag{4}$$

One service can be invoked until its whole input concepts have been generated. A concept cannot be generated unless at least one service whose output set contains it has been invoked.

$$|I_i| x_i \leq \sum_{j \in I_i} y_j, \ i = 0, \ldots, n+1 \tag{5}$$

$$y_j \leq \sum_{i \in \{k | j \in O_k\}} x_i, \ j = 1, \ldots, m \tag{6}$$

If sets I_i in (5) and $\{k|j \in O_k\}$ in (6) are empty sets, the right sides of them are treated as zero.

3.3.2 Response Time Constraints

In the MILP model, we pay attention to the criteria consisting of three parts. The first part of criteria is to minimize the invoked time sr_{n+1} of output service s_{n+1}, so only the lower bound need to be given. For each service, the constraint of response time is shown as follows:

$$\left.\begin{array}{l} sr_i \geq (1 - x_i)R_{max} \\ sr_i \geq R_i \\ sr_i \geq R_i + r_j, j \in I_i \end{array}\right\} \quad i = 0, 1, \ldots, n+1 \tag{7}$$

The first inequality in (7) makes the invoked time sr_i reach the maximum response time $R_{\max}$ while s_i is not selected. The third inequality makes the response time sr_i satisfy the definition of response time in Definition 2 when s_i is selected. A special case is that the set I_i is an empty set, such as I_0, in which the third equation makes no sense (no constraint). To handle this case correctly, we introduce the second inequality in which sr_i is greater than or equal to its original response time R_i. For example, the response time sr_0 of input service s_0 equals to R_0.

The generated time constraints of each concept are defined as follows.

$$r_j = \begin{cases} R_{\max} & \text{if } \{k|j \in O_k\} = \varnothing \\ \min_{i \in \{k|j \in O_k\}} sr_i & \text{otherwise} \end{cases} \quad j = 1, \ldots, m \tag{8}$$

However, it's esoteric that the minimum part in (8) can be transformed into a linear constraint [11]. We introduce variables $l_{ji} \in [0, +\infty), z_{ji} \in \{0, 1\}$ for each $r_j, i \in \{k|j \in O_k\}$, which ensure the equivalence between the minimum part of (8) and (9).

$$\left.\begin{array}{lll} r_j & \leq sr_i, & \forall i \in \{k|j \in O_k\} \\ r_j & \geq sr_i - l_{ji}, & \forall i \in \{k|j \in O_k\} \\ l_i & \leq (1 - z_{ji})R_{max}, & \forall i \in \{k|j \in O_k\} \\ \sum_{i \in \{k|j \in O_k\}} z_{ji} = 1 & & \end{array}\right\}, \quad j = 1, 2, \ldots, m \tag{9}$$

3.3.3 Throughput Constraints

Similarly, we only need to give an upper bound for throughput, since the criteria focus on the maximum throughput of output service. The throughput of a service depends on the throughputs of its input concepts and its own throughput, more precisely, on the minimum of them. If one service is not selected, we let its throughput to be T_{min} reasonably.

$$\left.\begin{array}{l} st_i \leq T_i x_i + (1 - x_i)T_{min} \\ st_i \leq t_j, \forall j \in I_i \end{array}\right\}, \quad i = 0, 1, \ldots, n+1 \tag{10}$$

Intuitively, the throughput of a concept is the maximum throughputs of all services which can generate the concept.

$$t_j = \begin{cases} T_{\min} & \text{if } \{k|j \in O_k\} = \varnothing \\ \max_{\forall i \in \{k|j \in O_k\}} st_i & \text{otherwise} \end{cases} \quad j = 1, \ldots, m \tag{11}$$

As same as the constraints of service generated time, the maximum part in (11) can be transformed into a linear constraint by introducing variables $g_{ji} \in [0, +\infty), u_{ji} \in \{0, 1\}$.

$$\left.\begin{array}{lll} t_j & \geq st_i, & \forall i \in \{k | j \in O_k\} \\ t_j & \leq sr_i + g_{ji}, & \forall i \in \{k | j \in O_k\} \\ g_i & \leq (1 - u_{ji}) T_{max}, & \forall i \in \{k | j \in O_k\} \\ \sum_{i \in \{k | j \in O_k\}} u_{ji} = 1 & & \end{array}\right\}, \quad j = 1, \ldots, m \tag{12}$$

3.4 Practical Techniques for MILP Model

QoS-aware web service composition can be seen as an NP-hard problem, for which there are no effective algorithms [12]. In practical terms, the above MILP model equivalent to the original problem works not well in some cases. For this reason, some effective techniques are applied to improve the performance of the MILP model.

3.4.1 Throughput Constraints Simplification

The vital part of (3) is the throughput of output service, while the throughputs of other services are inconsequential. We notice that the throughput of output service in a composition is the minimum throughput of all the selected services. Consequently, we can obtain the final correct throughput of output service with the following steps.

- Let the throughputs of selected services (expect s_{n+1}) to be their original throughputs.
- Let other throughputs to be T_{max}.
- Take the minimum throughput of all services as the throughput of output service.

The formalized description (13) can replace (10), (11) and (12), which reduces many constraints and variables. The second minimum equation can be transformed into linear constraints with the similar method used in (9).

$$\left.\begin{array}{rl} st_i & = T_i x_i + (1 - x_i) T_{max}, i = 0, 1, \ldots, n \\ st_{n+1} & = \min_{i=0}^{n} st_i \end{array}\right\} \tag{13}$$

3.4.2 Response Time Constraints Approximation

However, there are numerous integer variables introduced in the response time constraints, which causes a serious performance problem while applying a standard solver.

$$\left.\begin{array}{rl} sr_i & = R_i x_i + (1 - x_i) R_{max}, i = 0, 1, \ldots, n \\ sr_{n+1} & = \sum_{i=0}^{n} sr_i \end{array}\right\} \tag{14}$$

An efficient method is to replace the response time of output service with the sum of the response time of all chosen services, and the detail is described

in (14). It does greatly shorten the execution time while holding well enough criteria, even though the approximations of response time constraints are not completely accurate.

In summary, the MILP model with these techniques can be solved efficiently without building a huge and complex dependency graph. Extensive experiments applying groubi [10] solver are presented in Sect. 5. We notice that there is an obvious gap between the MILP method and other methods, which means we can still make great progress. Therefore, we propose a more effective and efficient mechanism in the next section.

4 Composition-Segment Candidate Optimization

In this section, a mechanism of optimizing composition-segments candidates is proposed to improve the performance of the MILP model. We define four kinds of segment candidates in Definition 4, and the core idea of this mechanism is to improve the score segment candidate in current composition with three other kinds of segment candidates.

Definition 4. *Composition-Segment Candidate ("segment candidate" for short) of a service is defined as a local composition whose last service is exactly the service. For a concept, its composition-segment candidate can generate it. Similar to the criteria (3) of MILP model, the score of a composition-segment Ω_s is defined as:*

$$Score(\Omega_s) = TP(\Omega_s) - \alpha * Len(\Omega_s) - \beta * RT(\Omega_s) \tag{15}$$

For each service and concept, we maintain four kinds of segment candidates—S_s, N_s, R_s and T_s, which hold the best current segment candidates of different objectives—score, length, response time and throughput respectively.

4.1 Generating Composition-Segment Candidates

To generate segment candidates, we construct the current output map M_c firstly, of which the keys are services or concepts and the values are lists of segment candidates.

Table 3. Segment candidates related to service I with $\alpha = 100, \beta = 20$

Candidate no	Composition-segment	Len	RT (ms)	TP (inv/s)	Score
1	$S_i \rightarrow A \rightarrow C$	3	60	180	−1320
2	$S_i \rightarrow A \vert\vert D \rightarrow E$	4	80	750	−1250
3	$S_i \rightarrow D \rightarrow F$	3	50	350	−950
4	$S_i \rightarrow A \vert\vert D \rightarrow C \vert\vert F \rightarrow I$	6	80	180	−2020
5	$S_i \rightarrow A \vert\vert D \rightarrow E \vert\vert F \rightarrow I$	6	100	350	−2250

Algorithm 1 takes service s_i and map M_c as inputs and checks whether service s_i can be invoked at line 2. Then, it initializes the list P_s with four sets of precursors and adds all precursors to these sets respectively. The following step is to create four candidates in order and assign their precursor sets with P_s respectively. Finally, the method *update_attribute* calculates their score, length and QoS, and we append the four candidates to list P_c.

Algorithm 1: Generating Composition-Segment Candidates

```
   Input: s_i, M_c
   Output: P_c
1  P_c ← []
2  if I_i ⊆ M_c.keys then
3      P_s ← [set(), set(), set(), set()]
4      for concept c ∈ I_i do
5          for segment i, s ∈ M_c[c] do
6              P_s[i].add(s)
7      for segments set p_s ∈ P_s do
8          s ← SegmentCandidate(s_i)
9          s.pre ← p_s
10         update_attribute(s)
11         P_c.append(s)
12 return P_c
```

Taking the generating process of service I in Fig. 1 as an example, we list some segment candidates related with service I in Table 3 and shows the detailed process in Fig. 2. Service I has three input concepts i_1, i_2, i_3, and their segment candidates are listed in the left (the green cell denotes score segment candidate). For each kind of segment candidate, the newly generated candidate combines the corresponding candidates of its inputs respectively. For example, the score segment candidate (Candidate 5 in Table 3) of I consists of Candidate 2—the score segment candidate of i_1, and Candidate 3—the score segment candidate of i_2 and i_3.

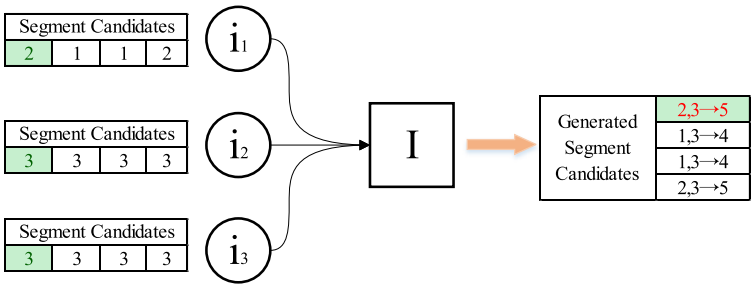


Fig. 2. Segment candidates generation

4.2 Optimizing Composition-Segment Candidates

After generating the segment candidates, the next step is to optimize the score segment candidate with other kinds of segment candidates. We firstly analyze the bottlenecks of score segment candidate. Algorithm 2 shows the process of analyzing bottlenecks of score segment candidate S_s.

Algorithm 2: Analyzing Bottlenecks of Candidate

Input: S_s
Output: L_b
1 $L_b \leftarrow [null, null, null]$
2 $B_l \leftarrow 0, B_r \leftarrow R_{min}, B_t \leftarrow T_{max}$
3 **for** segment $s \in S_s$.pre **do**
4 **if** $Len(s) > B_l$ **then**
5 $B_l = \text{Len}(s), L_b[0] = s.\text{concept}$
6 **if** $RT(s) > B_r$ **then**
7 $B_r = \text{RT}(s), L_b[1] = s.\text{concept}$
8 **if** $TP(s) < B_t$ **then**
9 $B_t = \text{TP}(s), L_b[2] = s.\text{concept}$
10 **return** L_b

For a score segment candidate S_s, Algorithm 2 finds its three kinds of bottlenecks. The operations from line 4 to 6 find the precursor with maximum length and record the corresponding concept in L_b. Similar operations are performed with response time bottleneck. On the contrary, the throughput bottleneck gets the minimal throughput concept of them.

Algorithm 3: Improving Bottlenecks of Candidate

Input: M_c, P_c
Output: P_c
1 **while** ***True*** **do**
2 $S_s \leftarrow P_c[0], S'_s \leftarrow S_s, L_b \leftarrow bottleneck\_analyze(S'_s)$
3 $score\_pre\_set \leftarrow S'_s$.pre, $b_1, b_2, b_3 = L_b$
4 $score\_pre\_set[b_1] \leftarrow M_c[b_1][1]$
5 $score\_pre\_set[b_2] \leftarrow M_c[b_2][2]$
6 $score\_pre\_set[b_3] \leftarrow M_c[b_2][3]$
7 S'_s.pre $\leftarrow score\_pre\_set$
8 update_attribute(S'_s)
9 **if** $S'_s.score > S_s.score$ **then**
10 $P_c[0] \leftarrow S'_s$
11 **else**
12 break
13 **return** P_c

Algorithm 3 improves the score segment candidate S_s in list P_c. Taking current output map M_c and candidates list P_c as the inputs, we use Algorithm 2

to get bottlenecks L_b, and then replace bottleneck candidates with the currently best candidates in M_c to improve S_s. Finally, the near-optimal score segment candidate S_s is generated by repeating the two foregoing steps until the score of S_s isn't able to be greater.

As shown in Fig. 2, the optimal score segment candidate is Candidate 4 instead of Candidate 5 (whose color is red). By calling Algorithm 2, we can obtain bottlenecks $L_b = [i_1, i_1, i_2]$. Then, Algorithm 3 handles each bottleneck of L_b in a same way. Taking the first element i_1 in L_b as an example, we replace the score segment candidate(Candidate 2) of i_1 with its length segment candidate(Candidate 1) in S_s, which reduce its length. Finally, the score segment candidate of I becomes Candidate 4.

4.3 Greedy Selection

Having generated four candidates P_c of service s_i and optimized the score candidate S_s in P_c, we compare each kind of candidate in P_c with the corresponding one in previous list $M_c[s_i]$ respectively and store the better ones. If the map M_c does not contain s_i, we insert the key-value pair (s_i, P_c) into M_c directly. For each output concept of s_i, we create four kinds of segment candidates and assign their precursors with the corresponding service candidates in P_c. Then, we perform similar operations to reserve the better ones. After greedy selection, Candidate 4 is reserved as the final score segment candidate of service I in Fig. 2.

By repeating the three above steps until the output map M_c is not changing, the score segment candidate of output service s_{n+1} appears, and the final composition is achieved.

5 Experimental Results

Extensive experiments have been carried out to evaluate the performance of our proposed methods. To make the conclusion more convincing, we evaluate our methods on two different groups of datasets.

Table 4. The characteristics of datasets

Datasets	D-01	D-02	D-03	D-04	D-05	R-01	R-02	R-03	R-04	R-05
#Service	572	4129	8138	8301	15211	1000	3000	5000	7000	9000
RT.opt (ms)	500	1690	760	1470	4070	1430	975	805	1225	1420
TP.opt (inv/s)	15000	6000	4000	4000	4000	1000	2500	1500	2000	2500
Len.opt	5	20	10	40	30	7	12	12	14	16

5.1 Datasets

To evaluate the performance of the proposed composition mechanisms, we conducted a group of experiments using five public repositories from the Web Service Challenge 2009 and five randomly generated datasets. As shown in Table 4, the

group of datasets of the WSC 2009 ranges from 572 to 15211 services. We evaluate further the performance of our algorithms with another group of datasets[1]. And the optimal values (RT.opt, TP.opt, Len.opt) of single objectives for each dataset are shown in it, which are computed by the memory-based algorithm.

5.2 Performance Analysis

To validate our approaches, we compare them with three different the-state-of-arts in the same experimental environment. For each dataset, we mainly show the solicitude for the global QoS of generated solution (*RT* for response time and *TP* for throughput), the length of composition (*Len*) and the execution time of method (*Time* including the time of building service dependency graphs).

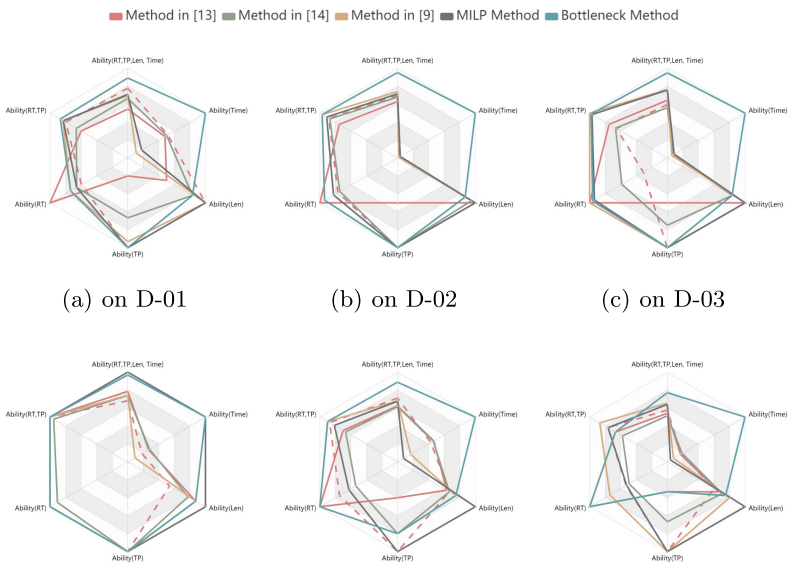
Table 5. Detailed comparisons with other methods

Datasets		D-01	D-02	D-03	D-04	D-05	R-01	R-02	R-03	R-04	R-05
Method in [13]	RT (ms)	500	1690	760	1470	4070	1430	975	805	1225	1420
	TP (inv/s)	3000	3000	2000	2000	1000	1000	1000	500	1000	500
	Len	10	20	10	42	33	8	19	18	21	19
	Time (ms)	73	1324	3591	10121	14925	26	161	531	1023	2066
	RT (ms)	840	2200	2450	4150	4990	1430	1305	1520	2095	1975
	TP (inv/s)	15000	6000	4000	2000	4000	1000	2500	1500	2000	2500
	Len	5	20	10	44	32	13	18	20	30	19
	Time (ms)	68	1373	3736	9283	12717	38	175	503	992	2053
Method in [14]	RT (ms)	760	2270	1300	2140	5340	1580	1815	1640	1840	2300
	TP (inv/s)	10000	6000	3000	1000	4000	1000	2000	1000	2000	1500
	Len	6	21	12	47	36	9	18	17	19	20
	Time (ms)	70	1252	3795	9813	14544	25	163	473	845	2096
Method in [9]	RT (ms)	680	1800	760	1600	4260	1430	975	1090	1225	1605
	TP (inv/s)	14000	6000	4000	3500	4000	1000	2000	1500	2000	2500
	Len	5	20	10	43	33	8	16	15	17	18
	Time (ms)	317	1684	3713	10651	13223	76	443	1136	1804	1613
MILP Method ($\alpha = 1, \beta = 0.2$)	RT (ms)	760	2050	810	3560	4130	1430	1560	1535	1620	2210
	TP (inv/s)	15000	6000	4000	4000	4000	1000	2500	1500	2000	2500
	Len	5	20	10	62	30	7	12	12	15	16
	Time (ms)	196	1113	2138	3558	4723	245	964	2259	3058	4828
Candidate Optimization Method ($\alpha = 10, \beta = 7$)	RT (ms)	680	1800	790	1470	4260	1430	975	805	1225	1420
	TP (inv/s)	15000	6000	4000	2000	4000	1000	2000	500	2000	2500
	Len	6	23	12	45	41	8	16	16	15	18
	Time (ms)	**35**	**98**	**72**	**459**	**274**	**7**	**75**	**90**	**101**	**230**

As shown in Table 5, [13] can generate two different solutions (one with the optimal response time and another with the optimal throughput). The method in [9] makes an excellent tradeoff of three attributes. Obviously, the execution time of compositions generated by methods [13], [14] and [9] is so long that some of them are longer than ten seconds. Moreover, our candidate optimization method runs not only fast but also archives ideal tradeoffs.

[1] https://wiki.citius.usc.es/inv:downloadable_results:ws-random-qos.

To measure the performance intuitively, we define $Ability(RT) = \frac{RT.opt}{RT}$, $Ability(TP) = \frac{TP}{TP.opt}$, $Ability(Len) = \frac{Len.opt}{Len}$ and $Ability(Time) = \frac{min(Time)}{Time}$. Moreover, we have $Ability(RT, TP) = [Ability(RT) + Ability(TP)]/2$, and the whole performance $Ability(RT, TP, Len, Time)$ is defined in the same manner. As shown in Fig. 3, the candidate optimization method has an outstanding $Ability(Time)$ and outperforms other methods in $Ability(RT, TP, Len, Time)$.



(a) on D-01 (b) on D-02 (c) on D-03

(d) on R-01 (e) on R-02 (f) on R-03

Fig. 3. Radar charts to compare the performance of five methods on several datasets.

6 Conclusions

In this paper, we formalize the multi-objective web service composition problem as a MILP model and propose a candidate optimization method to solve the model effectively and efficiently. A large number of experiments show that our candidate optimization method runs sharply fast while performing better than the state-of-the-art on QoS and number of services. Both the MILP method and the candidate optimization method save much running time with no need to build a service dependency graph.

Acknowledgment. This work is funded by the National Natural Science Foundation of China (No. 61673204), and the Fundamental Research Funds for the Central Universities (No. 14380046).

References

1. Jiang, W., Zhang, C., Huang, Z., Chen, M., Hu, S., Liu, Z.: Qsynth: a tool for QoS-aware automatic service composition. In: 2010 IEEE International Conference on Web Services, pp. 42–49. IEEE (2010)
2. Wagner, F., Ishikawa, F., Honiden, S.: QoS-aware automatic service composition by applying functional clustering. In: 2011 IEEE International Conference on Web Services, pp. 89–96. IEEE (2011)
3. Strunk, A.: QoS-aware service composition: a survey. In: 2010 Eighth IEEE European Conference on Web Services, pp. 67–74. IEEE (2010)
4. Fan, S.-L., Yang, Y.-B., Wang, X.-X.: Efficient web service composition via knapsack-variant algorithm. In: Ferreira, J.E., Spanoudakis, G., Ma, Y., Zhang, L.-J. (eds.) SCC 2018. LNCS, vol. 10969, pp. 51–66. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94376-3_4
5. Rodriguez-Mier, P., Mucientes, M., Lama, M.: Automatic web service composition with a heuristic-based search algorithm. In: 2011 IEEE International Conference on Web Services (ICWS), pp. 81–88. IEEE (2011)
6. Chen, M., Yan, Y.: Redundant service removal in QoS-aware service composition. In: 2012 IEEE 19th International Conference on Web Services, pp. 431–439. IEEE (2012)
7. Zeng, L., Benatallah, B., Ngu, A.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Softw. Eng. **30**(5), 311–327 (2004)
8. Yan, Y., Chen, M., Yang, Y.: Anytime QoS optimization over the PlanGraph for web service composition. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, pp. 1968–1975. ACM (2012)
9. Fan, S.-L., Ding, F., Guo, C.-H., Yang, Y.-B.: Supervised web service composition integrating multi-objective QoS optimization and service quantity minimization. In: Jin, H., Wang, Q., Zhang, L.-J. (eds.) ICWS 2018. LNCS, vol. 10966, pp. 215–230. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94289-6_14
10. Gurobi Optimization, LLC: Gurobi optimizer reference manual (2019)
11. Bertsimas, D., Mazumder, R., et al.: Least quantile regression via modern optimization. Ann. Stat. **42**(6), 2494–2525 (2014)
12. Jatoth, C., Gangadharan, G., Buyya, R.: Computational intelligence based QoS-aware web service composition: a systematic literature review. IEEE Trans. Serv. Comput. **10**(3), 475–492 (2015)
13. Xia, Y.M., Yang, Y.B.: Web service composition integrating QoS optimization and redundancy removal. In: 2013 IEEE 20th International Conference on Web Services, pp. 203–210. IEEE (2013)
14. Chattopadhyay, S., Banerjee, A., Banerjee, N.: A scalable and approximate mechanism for web service composition. In: 2015 IEEE International Conference on Web Services (ICWS), pp. 9–16. IEEE (2015)