Efficient Web Service Composition via Knapsack-Variant Algorithm

Shiliang Fan\(^1\) and Yubin Yang\(^1,\)\(^,*\)

\(^1\)State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
dyys1f@gmail.com, yangyubin@nju.edu.cn

Abstract. Since the birth of web service composition, minimizing the number of web services of the resulting composition while satisfying the user request has been a significant perspective of research. With the increase of the number of services released across the Internet, seeking efficient algorithms for this research is an urgent need. In this paper we present an efficient mechanism to solve the problem of web service composition. For the given request, a service dependency graph is firstly generated with the relevant services picked from an external repository. Then, each search step on the graph is transformed into a dynamic knapsack problem by mapping services to items whose volume and cost is changeable, after which a knapsack-variant algorithm is applied to solve each problem after transformation. Once the last search step is completed, the minimal composition that satisfies the request can be obtained. Experiments on eight public datasets proposed for the Web Service Challenge 2008 shows that the proposed mechanism outperforms the state-of-the-art ones by generating solutions containing the same or smaller number of services with much higher efficiency.

Keywords: Web Service Composition, Minima, Efficient, Knapsack.

1 Introduction

Web services are platform-independent applications which are released, discovered and invoked over the web using open standards such as UDDI [1], SOAP [2], and WSDL [3]. As software modules published on servers and consumed across the Internet, web services transmit their communication data over the network expeditiously, which leads to the loosely-coupled nature. The nature of loose coupling pave the way for easier and wider-ranging integration and interoperability among systems, making the technology of web services extensively used in enterprises. It is as plain as a pikestaff that web services have been the support technology for the swift development of IT-based services economy.

However, the complex business requirements cannot be fulfilled by a single service in most cases. But meanwhile, with the sharp increase of the number of web services, the composition of web services provides a way to solve the problem. Web service composition is the process of building a more complex,
functional workflow via combining a collection of single services together, to satisfy the inputs and outputs given by users. There are mainly two kinds of approaches for the web service composition problem. Some researches transform the composition problem into a planning one by mapping services to actions [4], [5], [6], which is known as AI-based technique. Others construct a graph to express the relationship of services and aim to extract a reachable path from the graph [7], [8], which is called graph-based technique.

There exists a phenomenon that a growing number of services own similar or identical functionality. Therefore, it is simple to known that, for a given request, the composition process of massive services may generate many possible solutions with different number of services. Minimizing the number of services of the resulting composition while satisfying the user request is significant for brokers, customers, and providers [9]. Standing in the shoes of brokers, a composition result with fewer services could make it easier for the work of maintenance and management. From the customers’ point of view, a smaller composition ordinarily means the less payment for those services invoked; On the other side, decreasing the number of services involved in the composition may highly increase the success rate of acquiring the wanted responses to the requests made by customers. As for service providers, solutions with fewer services could save resources and cost for the same task.

Up to now, there are several studies on the web service composition taking the optimization of the number of services into consideration. Nevertheless, in face of the repositories that contains a substantial amount of services, existing researches take too long time to obtain the optimal solution on account of the huge search space. Thus these approaches aren’t efficient enough to be applied to large-scale and real-time environments. In this paper, we aim at presenting a mechanism to efficiently solve the web service composition problem. The main contribution are:

- An equivalent transformation approach that transforms search steps on the service dependency graph into dynamic knapsack problems by mapping services to items with changeable volume and cost.
- A knapsack-variant algorithm that guarantees to efficiently generate the composition with minimal number of services by means of solving each dynamic knapsack problem.
- An optimization strategy to reduce the spatial complexity of the knapsack-variant algorithm.

Furthermore, a full validation on eight datasets of Web Service Challenge 2008 has been done. Experimental results show that our mechanism performs better than the state-of-the-arts both in term of quality and efficiency.

The rest of this paper is organized as follows. Section 2 reviews some related work, Section 3 describes the background and formalizes the web service composition problem, then illustrates the motivation of this research. Section 4 introduces the proposed mechanism, Section 5 shows the experimental results, and Section 6 provides some final remarks.
2 Related Work

Effectively combining minimal number of services distributed over the web to build enterprise-class services that satisfy given requirements is the goal of this paper. A survey of the problem of web service composition shows that several researches have been done in this perspective, and each has its own merits.

A heuristic A* search algorithm is proposed in [7] for the problem of automatic web service composition. For a given request, a digraph called service dependency graph is constructed firstly with a part of the original services chosen from an external repository. Meanwhile, some techniques are applied to reduce the useless nodes in the graph. Then the heuristic-based search algorithm named A* is executed over the optimized graph to seek the minimal composition which fulfills the user request. Though it can obtain compositions with minimal number of services on WSC-2008’s datasets, it may show a poor performance in large-scale and real-time environments. On one hand, different kinds of optimizations on the service dependency graph may spend large quantities of time. On the other hand, quite a few iterations of A* search algorithm aren’t applicable to real-time scenarios.

A scalable and approximate mechanism is presented in [10] to get the near-minimal compositions against time. The authors proposed an on-the-fly strategy to construct only a path of the auxiliary graph instead of the complete graph. Additionally, a deterministic approach and a probabilistic approach are discussed to find the path with the minimal number of services, which is the final result of composition. Though the algorithm has a superior service composition time compared to other algorithms, the greedy strategy adopted always gets stuck in local optima. As a result, it always generates compositions with more services than the others. In a word, it performs well in efficiency but it remains to be improved in terms of the quality of solutions.

Pablo et al. [11] present a composition framework integrating fine-grained I/O service discovery strategy and an optimal composition search algorithm. To improve the efficiency of the generation of a layered service composition graph, the discovery and matchmaking phases are optimized using indexes and cache. Once the graph is generated, many optimizations are applied to reduce the graph size. Then a search which is modelled as a state-transition system is performed over the graph to find the minimal composition among all the possible compositions satisfying the request provided by user. Experimental results show the scalability and flexibility of the composition framework. However, similar with the mechanism in [7], though lots of optimizations are used to improve the optimal composition search performance, much extra time is spent in the step. On the other side, the search algorithm isn’t efficient enough to be applied to large-scale and real-time environments.

In summary, despite above algorithms to optimize number of services, there is a lack of approaches that have the ability to minimize the number of services of the composition effectively and efficiently. This paper proposes an effective and efficient mechanism in an effort to find compositions with minimal number of services in large-scale and real-time scenarios.
3 Preliminaries and Motivation

3.1 Preliminary Knowledge

Web service composition is a well studied problem, and semantic web services are
the foundation of the problem. In this paper, a semantic web service is formally
defined as follows [12], [13], [14].

**Definition 1.** Given a set of concepts named Con, a Semantic Web Service
("service" for short) is defined as a tuple $s = \{\text{In}_s, \text{Out}_s\}$, where $\text{In}_s = \{\text{in}_1^s, \ldots, \text{in}_n^s\}$ is the set of inputs required to invoke the semantic web service $s$, and $\text{Out}_s = \{\text{out}_1^s, \ldots, \text{out}_n^s\}$ is the set of outputs generated by executing the service $s$. Each element of $\text{In}_s$ and $\text{Out}_s$ is actually a semantic concept belonging to the set Con, namely, $\text{In}_s \subseteq \text{Con}$ and $\text{Out}_s \subseteq \text{Con}$.

Individual services can be combined by connecting their matched inputs and
outputs to construct compositions [15], [16].

**Lemma 1.** Given an output $\text{out}_s \in \text{Out}_s$ of a service $s$, as well as an input $\text{in}_{s'} \in \text{In}_{s'}$ of another service $s'$, if $\text{out}_s$ and $\text{in}_{s'}$ are equivalent concepts or $\text{out}_s$ is a sub-concept of $\text{in}_{s'}$, $\text{out}_s$ matches $\text{in}_{s'}$ (i.e., $\text{in}_{s'}$ is matched by $\text{out}_s$).

There are mainly two kinds of structures of these compositions, namely sequential structure and parallel structure [17]. The services organized as a sequential structure are invoked in order, while the services organized as a parallel structure are invoked synchronously. A composition can be described as follows.

**Definition 2.** A Composition containing the set of services $S = \{s_1, \ldots, s_n\}$ is defined as $\Omega_S = s_1, \ldots, s_n$. If the services are chained in sequence, the composition is expressed as $\Omega_S = s_1 \rightarrow \ldots \rightarrow s_n$; if in parallel, then $\Omega_S \parallel = s_1 \parallel \ldots \parallel s_n$. The set of services involved in $\Omega_S$ is defined as $\text{Servs}(\Omega_S) = S$. Moreover, the length of a composition $\Omega_S$ is defined as $\text{Len}(\Omega_S) = |S|$, namely the number of services in $\Omega_S$.

The aim of the service composition problem is to automatically select the
minimal composition of available services to fulfil a user request that is defined as follows.

**Definition 3.** A Request from users is defined as a tuple $R = \{\text{In}_R, \text{Out}_R\}$, where $\text{In}_R = \{\text{in}_1^R, \ldots, \text{in}_n^R\}$ is the set of inputs provided by users ($\text{In}_R \subseteq \text{Con}$), and another element $\text{Out}_R = \{\text{out}_1^R, \ldots, \text{out}_n^R\}$ represents the set of expected outputs ($\text{Out}_R \subseteq \text{Con}$).

On the basis of the above concepts, the precise definition of the web service
composition problem is given as follows.

**Definition 4.** A Web Service Composition Problem is defined as, for a given
composition request $R$, to seek for a composition $\Omega_S$ fulfilling $R$ with the optimization objective of $\min|S|$, namely, the $\Omega_S$ contains the minimal number of services.
3.2 A Motivating Example

Graph is a natural and intuitive way to express the complex interaction relations between entities. *Service Dependency Graph* is a digraph used to describe services and the matching relations among them [8], [10], [12]. For a given request \( R = \{\{in_1, in_2\}, \{out_1, out_2\}\} \), an example of service dependency graph is shown in Fig.1. Each service is represented as a rectangle, while the inputs and outputs of a service are represented as circles. Furthermore, the matching relations among services are represented as edges connecting two circles.

Fig. 1. A service dependency graph example with the optimal composition highlighted.

As can be seen from Fig.1, the highlighted composition which can be represented as \( \Omega = s_o \rightarrow ((A \rightarrow D) \parallel (B \rightarrow E)) \rightarrow G \rightarrow J \rightarrow s_k \) contains eight services in total (including the \( s_o \) and the \( s_k \)). There are also several other compositions satisfying the same user request \( R \) such as \( \Omega' = s_o \rightarrow ((A \rightarrow C \rightarrow I) \parallel (B \rightarrow F \rightarrow H \rightarrow K)) \rightarrow s_k \), whereas the number of services of them are unexceptionally more than eight. Therefore, the composition \( \Omega \) highlighted in the graph is the optimal one with the minimal number of services.

In large-scale scenarios, the service dependency graph may be exceedingly complex, which leads to a huge search space [18]. As a consequence, it is formidable to extract the optimal composition from the graph. There is no doubt that the exhaustive combinatorial search can guarantee the optima, but it will take an unacceptable long time to generate the compositions and isn’t applicable to real-time environments. To sum up, we should pay attention not only to the quality of the resulting composition but also to the efficiency of the composition algorithm.

4 Detailed Methodology

In this section, an efficient mechanism is proposed for the problem of web service composition. Given a composition request \( R = \{In_R, Out_R\} \) and a service repository \( S_r \), a service dependency graph is firstly constructed with the relevant services for the request. Then, search steps on the graph are transformed into dynamic knapsack problems, after which a knapsack-variant algorithm is proposed.
to solve each dynamic knapsack problem. Finally, an optimization strategy is adopted to reduce the spatial complexity of the knapsack-variant algorithm.

4.1 Service Dependency Graph

As shown in Fig.1, a service dependency graph is a layered digraph. The first layer contains only one dummy service \( s_0 = \{ \emptyset, In_R \} \), similarly, there is only a dummy service \( s_k = \{ Out_R, \emptyset \} \) in the last layer, while the concrete services in the other layers are selected from \( S_r \). Moreover, each layer contains the services whose inputs are all matched by the outputs generated by previous layers.

The generation process of a service dependency graph is shown in Algorithm 1. Given the request \( R \) and the repository \( S_r \), \( s_0 \) is firstly added to the first layer \( L_0 \), after which each following layer \( L_i \) is constructed with the services whose inputs are all matched by the outputs generated by previous layers. \( s_k \) will be added to the last layer if the set of expected outputs \( Out_R \) is included in \( Out_{all} \). Finally, unused services making no contribution to \( Out_R \) are removed from the graph by traversing from the last layer to the first layer.

Algorithm 1: Construction of Service Dependency Graph

| Input: \( R, S_r \) |
|------------------|
| Output: \( L \) |
| 1. \( i \leftarrow 0, L_0 \leftarrow \{s_o\}, i \leftarrow i + 1, Out_{all} \leftarrow In_R \) |
| 2. repeat |
| 3. for service \( s \in S_r \) do |
| 4. if \( s \notin L_j (\forall j < i) \) and \( In_s \subseteq Out_{all} \) then |
| 5. \( L_i \leftarrow L_i \cup \{s\} \) |
| 6. \( Out_{all} \leftarrow Out_{all} \cup Out_s \) |
| 7. \( i \leftarrow i + 1 \) |
| 8. until \( Out_R \subseteq Out_{all} \); |
| 9. \( tot \leftarrow i, L_{tot} \leftarrow \{s_k\}, j \leftarrow tot, In_{all} \leftarrow Out_R \) |
| 10. while \( j \geq 0 \) do |
| 11. for service \( s \in L_j \) do |
| 12. if \( Out_s \cap In_{all} = \emptyset \) then |
| 13. \( L_j \leftarrow L_j \setminus \{s\} \) |
| 14. for service \( s \in L_j \) do |
| 15. \( In_{all} \leftarrow In_{all} \cup In_s \) |
| 16. \( j \leftarrow j - 1 \) |
| 17. return \( L \) |

4.2 The Dynamic Knapsack Problem

Once the service dependency graph is completed, the composition problem is regarded as searching for a reachable path from \( s_o \) to \( s_k \). Each search step on the graph is defined as determining the optimal precursors of each service.
Definition 5. The set of precursors of a service $s \in L_i$ is defined as $\text{Pre}(s) = \{s' \mid s' \in L_j (\forall j < i) \land \text{In}_s \cap \text{Out}_{s'} \neq \emptyset\}$. Specially, $\text{Pre}(s_o) = \emptyset$.

The search step of service $G$ is shown in Fig.2. Note that, the minimal composition ending with a service $s$ is expressed as $\Omega^s$, and $c_i$ in the figure represents an input or output concept of services. Assuming that the minimal compositions ending with the precursors of $G$, i.e., $\Omega^C$, $\Omega^D$, $\Omega^E$, and $\Omega^F$, have been determined in advance, the search step of $G$ is defined as selecting the optimal subset of $\{\Omega^C, \Omega^D, \Omega^E, \Omega^F\}$ to compose the $\Omega^G$, which is actually a greedy strategy. On the basis of the greedy strategy, an equivalent transformation approach is proposed to transform each search step similar to the one shown in Fig.2 into a Dynamic Knapsack Problem.

As shown in Fig.3, assuming that the optimal precursors of a service $s$ require to be determined, $s$ is abstracted into a knapsack whose capacity is $\text{In}_s$ (the set of inputs of the service $s$). Each precursors of $s$ is spontaneously regarded as an item with dynamic volume and cost. The problem is to minimize the sum of the cost of the items in the knapsack so that the sum of the volume is equal to the knapsack’s capacity. The volume of an item $s' \in \text{Pre}(s)$ is relevant to $\text{Out}_{s'}$ (the set of outputs of the service $s'$), and the cost of $s'$ is measured with $\text{Serve}(\Omega^{s'})$ (the set of services involved in the composition $\Omega^{s'}$), which is too inconvenient to be applied to the following composition algorithm. Therefore, an approach is presented to quantify the volume and the cost of each item.

**Algorithm 2:** Generation of Subsets

| Input: $V$ |
|-------------|
| Output: $\text{Subs}$ |
| 1 $\text{Subs} \leftarrow \{\emptyset\}$, $\text{upper bound} \leftarrow 2^{|V|}$ |
| 2 for $\text{index} = 0$; $\text{index} < \text{upper bound}$; $\text{index}++$ do |
| 3 $i \leftarrow 0$, $\text{tmp} \leftarrow \text{index}$, $\text{subset} \leftarrow \{\}$ |
| 4 while $\text{tmp} > 0$ do |
| 5 if $(\text{tmp mod 2}) > 0$ then |
| 6 $\text{subset} \leftarrow \text{subset} \cup \{V[i]\}$ |
| 7 $\text{tmp} \leftarrow \text{tmp div 2}$, $i \leftarrow i + 1$ |
| 8 $\text{Subs[\text{index}]} \leftarrow \text{subset}$ |
| 9 return $\text{Subs}$ |
The quantization of volume. Firstly, all the subsets of $In_x$ is obtained by Algorithm 2 in a certain order. Then, on the ground of the returned subsets $Subs$, a mapping table is constructed to quantify the volume of the knapsack and each item. Taking the problem in Fig.3 as an example, the mapping table used to support the volume quantization is shown as Table 1.

Table 1. The mapping table.

| index | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
|-------|----|----|----|----|----|----|----|----|
| $Subs[index]$ | $\emptyset$ | $\{c_1\}$ | $\{c_2\}$ | $\{c_1, c_2\}$ | $\{c_3\}$ | $\{c_1, c_3\}$ | $\{c_2, c_3\}$ | $\{c_1, c_2, c_3\}$ |
| item/knapsack | C  | D  |    |    |    |    |    |    |
| volume | 0  | 1  | 2  | 3  | 4  | 5  | 6  | 7  |
| index(binary) | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| index | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| $Subs[index]$ | $\{c_4\}$ | $\{c_1, c_4\}$ | $\{c_2, c_4\}$ | $\{c_1, c_2, c_4\}$ | $\{c_3, c_4\}$ | $\{c_1, c_3, c_4\}$ | $\{c_2, c_3, c_4\}$ | $\{c_1, c_2, c_3, c_4\}$ |
| item/knapsack | F  | E  | G  |    |    |    |    |    |
| volume | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 |
| index(binary) | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

By means of Table 1, the volume of the knapsack and each item can be quantified as follows.

- The capacity of the knapsack $s$ is quantified as the upper bound of $index$, namely $|Subs| - 1$.
- Assuming that the service $s'$ provides the set of outputs $Out \subseteq Out_s'$ for the service $s$, the volume of $s'$ is quantified as the value of the $index$ which satisfies the condition that $Subs[index] = Out$.

Taking the problem in Fig.3 as an instance, the capacity of the knapsack $G$ is $V_{cap} = 15$ after quantization. Two different feasible solutions of the dynamic knapsack problem are shown in Fig.4 respectively. As can be seen from the solution I, the Service $D$ provides the set of outputs $\{c_1, c_2\}$ for the Service $G$, thus the volume of the item $D$ is quantified as $volume_D = 3$ according to Table 1. In addition, $volume_E = 4$ owing to the fact that the Service $E$ provides the set of outputs $\{c_3\}$ for $G$. Similarly, $volume_F = 8$. It is not difficult to observe that $volume_D + volume_E + volume_F = 3 + 4 + 8 = V_{cap}$, hence the knapsack $G$ can be filled with the set of items $\{D, E, F\}$, which indicates the effectiveness of the quantization.

The volume of an item is changeless in the 0-1 knapsack problem, while the volume of an item is changeable in the dynamic knapsack problem. For example, let’s discuss the solution II shown in Fig.4(b). Despite the fact that the set of outputs of the Service $D$ is $\{c_1, c_2\}$, $D$ only provides $\{c_2\}$ for $G$ seeing that $\{c_1\}$ is provided by the Service $C$, which leads to the change of $volume_D$ from 3 to 2. Meanwhile, $volume_C + volume_D + volume_E + volume_F = 1 + 2 + 4 + 8 = V_{cap}$. The outputs provided by the service $s'$ for the service $s$ are uncertain before decision-making, hence the volume of the item $s'$ can’t be determined in advance.
The quantization of cost. Considering that the goal of this paper is minimizing the number of services involved in the resulting composition, the cost of an item $s'$ is designed as the minimal number of services that require to be invoked to generate the outputs of the service $s'$. Therefore, the cost of each item is quantified as follows.

- Assuming that $Ser$ represents the set of services which belong to $Servs(Ω')$ and haven’t been invoked yet, the volume of the item $s'$ is quantified as the size of the $Ser ∪ \{s'\}$.

Taking the solution I shown in Fig.4(a) as an example, suppose that the set of items $\{D, E, F\}$ are put into the knapsack $G$ in order of the name $(D→E→F)$. We have already assumed that the minimal compositions ending with the precursors of $G$ have been determined in advance. Figure 2 reveals that $Ω^D = s_o→A→D$, $Ω^E = s_o→B→E$ and $Ω^F = s_o→B→F$. On account of the fact that $D$ is included in the knapsack $G$ before $E$, the set of services $\{s_o\}$ has been invoked in advance. As a result, only $\{B\}$ requires to be invoked before $E$, which leads to $cost_E = 2$.

Similarly, $cost_D = 3$, $cost_F = 1$ and the total cost of the solution I is calculated as $cost_D + cost_E + cost_F = 3 + 2 + 1 = 6$.

The cost of an item is changeable as well in the dynamic knapsack problem. As shown in the solution II (assume that the items are put into the knapsack $G$ in order of $C→D→E→F$), the set of services $\{s_o, A\}$ has been invoked when putting the item $C$ into the knapsack. Consequently $D$ can be invoked directly, which leads to the change of $cost_D$ from 3 to 1.

In summary, only by ingeniously determining the dynamic volume and cost of an item can we solve the dynamic knapsack problem.

4.3 The Knapsack-Variant Algorithm

The dynamic knapsack problem aiming at determining the optimal precursors of a service $s$ can be described as follows. Given a knapsack $s$ whose volume capacity is $V_{cap}$, and a set of items $Pre(s) = \{s_1, s_2, \ldots, s_N\}$ where $N = |Pre(s)|$ represents the number of items, each with a dynamic volume $volume_i$ and a
dynamic cost $cost_i$, some items are selected from $Pre(s)$ to fill the knapsack $s$ with the objective of:

$$\text{minimize} \quad \sum_{i=1}^{N} cost_i \cdot x_i$$

$$\text{subject to} \quad \sum_{i=1}^{N} volume_i \cdot x_i = V_{cap},$$

$$x_i \in \{0, 1\}.$$  \hspace{1cm} (1)

where $x_i$ represents the number of the item $i$ to include in the knapsack. Unlike the 0-1 knapsack problem, all the $volume_i$ and $cost_i$ are uncertain here, which leads to the inapplicability of the 0-1 Knapsack Algorithm. In this section, a Knapsack-Variant Algorithm is proposed to solve the problem by determining the volume and cost of each service dynamically.

Let $C[i][v]$ represent the minimal cost of selecting items from $\{s_1, s_2, \ldots, s_i\}$ ($1 \leq i \leq N$) to fill a temporary knapsack whose capacity is $v$ ($1 \leq v \leq V_{cap}$), and $I[i][v]$ the set of items selected to minimize $C[i][v]$. Then,

$$C[i][v] = \min \{C[i-1][v], C[i-1][v-volume_i] + cost_i\}$$

where $volume_i = DV(s_i, In_s, Subs, v)$, $cost_i = DC(s_i, I, i, v, volume_i)$.

$volume_i$ and $cost_i$ are determined dynamically in the process of the dynamic programming, which is the quintessence of the knapsack-variant algorithm.

The function $DV$ in Algorithm 3 is used to dynamically calculate the volume of an item. For the given temporary knapsack with the capacity of $v$, the outputs provided by service $s_i$ for the knapsack are determined as $Out_{s_i} \cap Subs[v]$. Thus, the volume of the item $s_i$ can be quantified by the quantization approach proposed in Sect.4.2. Inspired by the one-to-one match between the $index(binary)$ and the $Subs[index]$ in Table 1, a binary method is applied to determine the $index$ which satisfies $Subs[index] = Out_{s_i} \cap Subs[v]$, namely the $volume_i$.

\textbf{Algorithm 3:} Determination of Volume of Items

\begin{verbatim}
Input: $s_i, In_s, Subs, v$
Output: $DV(s_i, In_s, Subs, v)$
1 map ← {}, volume ← 0, Out ← $Out_{s_i} \cap Subs[v]$
2 for index = 0; index < $|In_s|$; index++ do
3 \hspace{1cm} c ← $In_s[index]$
4 \hspace{1cm} map[c] ← index
5 for concept $c \in Out$ do
6 \hspace{1cm} index ← map[c]
7 \hspace{1cm} volume ← volume + $2^{index}$
8 $DV(s_i, In_s, Subs, v)$ ← volume
9 return $DV(s_i, In_s, Subs, v)$
\end{verbatim}
For the given temporary knapsack with the capacity of v, the reason why the outputs provided by si for the knapsack are determined as Outsi ∩ Subsv can be explained as follows. The knapsack with the capacity of v corresponds to a temporary service sv with the inputs of Subsv. Therefore, the largest set of outputs provided by the service si for sv is obviously Outlar = Outsi ∩ Subsv. However, if a smaller one Outsma ⊂ Outlar is provided for sv, Outlar − Outsma may require to be provided by extra services selected from {s1, s2, ..., si−1}, which causes the loss of the local optimum, let alone the global optimum.

Moreover, the function DC shown in Algorithm 4 is applied to determine the cost of an item si. Since the items that have been included in the knapsack are cached in the data structure I, the set of services Ser ⊆ Servs(Ωsi) which haven’t been invoked can be obtained drawing support from I, after which the cost of the item si is quantified as |
Ser| + 1.

Algorithm 4: Determination of Cost of Items

Input: si, I, i, v, volumei
Output: DC(si, I, i, v, volumei)
1 Union ← {}  
2 for service s ∈ I[i−1][v − volumei] do  
3 Union ← Union ∪ Servs(Ωs)  
4 Inter ← Servs(Ωsi) ∩ Union  
5 Ser ← Servs(Ωsi) − Inter  
6 DC(si, I, i, v, volumei) ← |Ser| + 1  
7 return DC(si, I, i, v, volumei)

According to the optimization model in (2), by systematically increasing the values of i (from 1 to N) and v (from 1 to Vcap), the composition Ωsi with the minimal number of services will be finally obtained when i = N and v = Vcap.

\[ \text{Len}(Ωsi) = C[N][Vcap] + 1. \]  
(3)

Therefore, the time complexity of the search step is \(O(NV_{cap})\), so is the spatial complexity. However, the spatial complexity of (2) can be further optimized.

Considering that \(C[i][v]\) is only relevant to \(C[i−1][v^′] (1 \leq v^′ \leq v)\), \(C[i][v]\) can be replaced by an one-dimensional array \(C[v]\). Then,

\[ C[v] = \min \{C[v], C[v − volumei] + cost_i\} \]

where \(volume_i = DV(s_i, In_s, Subs, v), \)

\(cost_i = DC(s_i, I, i, v, volume_i). \)  
(4)

The problem can be solved by systematically increasing the values of i (from 1 to N) and decreasing v (from Vcap to 1), hence the spatial complexity is reduced from \(O(NV_{cap})\) to \(O(V_{cap})\).

The knapsack-variant algorithm is shown in Algorithm 5. Search steps on the graph are carried out layer by layer. Each search step depends on the optimization results of search steps in previous layers and is transformed into a knapsack problem that can be solved by (4). After completing the last search step of sk, the expected composition with the length of \(\text{Len}(Ωsk)\) can be obtained.
Algorithm 5: Knapsack-Variant Algorithm

Input: \( L \)
Output: \( \text{Len}(\Omega^k) \)

1. \( \text{Servs}(\Omega^0) \leftarrow \{ s_0 \}, \text{Len}(\Omega^0) \leftarrow 1 \)
2. for \( \text{index} = 1; \text{index} < |L|; \text{index}++ \) do
   3. for service \( s \) in \( L_{\text{index}} \) do
      4. \( \text{pres} \leftarrow \text{Pre}(s) \), \( N \leftarrow |\text{pres}| \)
      5. \( \text{Subs} \leftarrow \) all subsets of \( I_{\text{pres}} \), \( V_{\text{cap}} \leftarrow |\text{Subs}| - 1 \)
      6. \( C[0..V_{\text{cap}}] \leftarrow +\infty \), \( C[0] \leftarrow 0 \), \( I[0..N][0..V_{\text{cap}}] \leftarrow \{ \} \)
      7. for \( i = 1; i <= N; i++ \) do
         8. \( s_i \leftarrow \text{pres}[i] \)
         9. for \( v = V_{\text{cap}}; v > 0; v-- \) do
            10. \( \text{volume}_i \leftarrow \text{DV}(s_i, I_{\text{pres}}, \text{Subs}, v) \)
            11. \( \text{cost}_i \leftarrow \text{DC}(s_i, I, i, v, \text{volume}_i) \)
            12. if \( C[v - \text{volume}_i] + \text{cost}_i < C[v] \) then
               13. \( C[v] \leftarrow C[v - \text{volume}_i] + \text{cost}_i \)
               14. \( I[i][v] \leftarrow I[i - 1][v - \text{volume}_i] \cup s_i \)
            else
               15. \( I[i][v] \leftarrow I[i - 1][v] \)
         16. \( \text{Servs}(\Omega^i) \leftarrow \{ \} \)
         17. for service \( \text{item} \) in \( I[N][V_{\text{cap}}] \) do
            18. \( \text{Servs}(\Omega^i) \leftarrow \text{Servs}(\Omega^i) \cup \text{Servs}(\Omega^{\text{item}}) \)
      19. \( \text{Len}(\Omega^i) \leftarrow C[V_{\text{cap}}] + 1 \)
3. return \( \text{Len}(\Omega^k) \)

5 Experimental Results

In order to evaluate the performance of the proposed composition algorithm, we completed a group of experiments on eight public repositories from the 2008 Web Service Challenge. Services in each repository are defined using a WSDL file, and inputs and outputs are semantically described in an XML file called ontology.

Table 1 shows the detailed characteristics of each dataset. The number of services and concepts in the ontology of each dataset are shown in rows \#Services and \#Concepts respectively. Row \#Sol.Services indicates the number of services for the optimal solution provided by the WSC’08.

Table 2. Characteristics of the Datasets.

| WSC-2008's Datasets | D-01 | D-02 | D-03 | D-04 | D-05 | D-06 | D-07 | D-08 |
|----------------------|------|------|------|------|------|------|------|------|
| \#Services           | 158  | 558  | 604  | 1041 | 1090 | 2198 | 4113 | 8119 |
| \#Concepts           | 1540 | 1565 | 3089 | 3135 | 3067 | 12468| 3075 | 12337|
| \#Sol.Services       | 10   | 5    | 40   | 10   | 20   | 40   | 20   | 30   |
To validate our composition algorithm, we compared our approach with three different approaches in the same experimental environment. For each dataset, we mainly focused on the number of services in the composition result ($\#C.\text{Services}$) and the execution time to extract the solution from the service dependency graph ($C.\text{Time}$). The results are shown in Table 3.

Table 3. Comparison with other approaches.

| Datasets | D-01 | D-02 | D-03 | D-04 | D-05 | D-06 | D-07 | D-08 |
|----------|------|------|------|------|------|------|------|------|
| **Method in [7]** | | | | | | | | |
| #C.\text{Services} | 10 | 5 | 40 | 10 | 20 | 35 | 20 | 30 |
| C.\text{Time (ms)} | 47 | 78 | 1028 | 54 | 1295 | 137 | 243 | 191 |
| **Method in [10]** | | | | | | | | |
| #C.\text{Services} | 14 | 5 | 48 | 12 | 34 | 47 | 20 | 36 |
| C.\text{Time (ms)} | 1 | 1 | 2 | 1 | 2 | 2 | 3 | 2 |
| **Method in [11]** | | | | | | | | |
| #C.\text{Services} | 10 | 5 | 40 | 10 | 20 | 35 | 20 | 30 |
| C.\text{Time (ms)} | 61 | 52 | 176 | 122 | 156 | 855 | 193 | 304 |
| **Our Method** | | | | | | | | |
| #C.\text{Services} | 10 | 5 | 40 | 10 | 20 | 35 | 20 | 30 |
| C.\text{Time (ms)} | 6 | 10 | 21 | 13 | 22 | 61 | 33 | 20 |

As can be seen from Table 3, compared with other approaches, our approach can generate compositions with the same or smaller number of services. On the dataset $D-06$, our approach succeeds to find a better composition than the solution provided by the WSC’08 (35 versus 40). The execution time of [10] is no more than 3 ms on all datasets, which proves that the method is efficient enough to solve the service composition problem. However, it always generates the compositions with more services than the others. Considering that all the methods except [10] can find the minimal compositions, we compare our method with those methods in terms of the execution time.

Figure 10 shows that, our algorithm takes far less time to generate solutions than the other two. The dotted lines in blue and orange represents the average

Fig. 5. Efficiency comparison.
execution time of [7] and [11] respectively, while the green one shows the average time of our algorithm. Apparently our algorithm is over 10 times faster than [7] and nearly 17 times faster than [11] on average, which is a significant improvement.

Table 4. Further comparison considering the service dependency graph.

| Datasets | D-01 | D-02 | D-03 | D-04 | D-05 | D-06 | D-07 | D-08 |
|----------|------|------|------|------|------|------|------|------|
| **Method in [7]** | | | | | | | | |
| G.Size | 17 | 19 | 60 | 31 | 62 | 95 | 89 | 78 |
| G.Time (ms) | 37 | 43 | 872 | 219 | 4861 | 536 | 7533 | 4761 |
| Tot.Time (ms) | 84 | 121 | 1900 | 273 | 6156 | 673 | 7776 | 4952 |
| **Method in [11]** | | | | | | | | |
| G.Size | 13 | 13 | 40 | 25 | 52 | 75 | 70 | 58 |
| G.Time (ms) | 138 | 297 | 553 | 472 | 618 | 891 | 1253 | 1374 |
| Tot.Time (ms) | 199 | 349 | 729 | 594 | 774 | 1746 | 1446 | 1678 |
| **Our Method** | | | | | | | | |
| G.Size | 60 | 61 | 104 | 43 | 101 | 170 | 140 | 124 |
| G.Time (ms) | 5 | 12 | 62 | 15 | 62 | 181 | 403 | 576 |
| Tot.Time (ms) | 11 | 22 | 83 | 28 | 84 | 242 | 436 | 596 |

We further compare our method with [7] and [11] taking the generation of the service dependency graph into account. As shown in Table 4, the size of the graph (G.Size) generated by [7] and [11] is smaller than ours because many optimizations are applied to reduce the graph size in these two methods, which leads to the fact that the time taken to generate the graph (G.Time) is longer than ours. Therefore, our knapsack-variant algorithm is executed over the the graph with larger size but is still over 10× faster than the other two, which sufficiently indicates the efficiency of our composition algorithm. Even though taking the G.Time into consideration, the total time (Tot.Time = G.Time + C.Time) of our mechanism is still much less than the others. As a result, our mechanism is more applicable to the large-scale or real-time scenarios.

6 Conclusions

In this paper we proposed an effective and efficient mechanism to automatically generate the minimal composition over a service dependency graph. Each search step on the graph is ingeniously transformed into a dynamic knapsack problem, after which the proposed knapsack-variant algorithm is executed to minimize the number of services by solving each dynamic knapsack problem. Moreover, a full validation performed on eight different datasets from Web Service Challenge 2008 shows that our algorithm outperforms the state-of-the-art methods, as it executes much faster than the state-of-the-arts while keeping the minimal composition results and is applicable to the large-scale or real-time scenarios.

7 Acknowledgment

This work is funded by the Natural Science Foundation of China (Nos. 61673204, 61321491), State Grid Corporation of Science and Technology Projects (Funded
No. SGLNXT00DKJS1700166), the Program for Distinguished Talents of Jiangsu Province, China (No. 2013-XXRJ-018), and the Fundamental Research Funds for the Central Universities (No. 020214380026).

References

1. Bellwood, T., Bryan, D., Drahuk, V., Ehnebuske, D., Glover, T., Hately, A.: Uddi version 2.04 api specification. UDDI Committee Specification, OASIS (2002)
2. Hadley, M., Mendelsohn, N., Moreau, J., Nielsen, H., Gudgin, M.: Soap version 1.2 part 1: Messaging framework. W3C REC REC-soap12-part1-20030624, June (2003) 240–8491
3. Chinnici, R., Gudgin, M., Moreau, J.J., Schlimmer, J., Weerawarana, S.: Web services description language (wsdl) version 2.0 part 1: Core language. W3C working draft 26 (2004)
4. Sirin, E., Parsia, B.: Planning for semantic web services. In: Semantic Web Services Workshop at 3rd International Semantic Web Conference, Springer Hiroshima, Japan (2004) 33–40
5. Klusch, M., Gerber, A., Schmidt, M.: Semantic web service composition planning with owls-xplan. In: Proceedings of the 1st Int. AAAI Fall Symposium on Agents and the Semantic Web. (2005) 55–62
6. Akkiraju, R., Srivastava, B., Ivan, A.A., Goodwin, R., Syeda-Mahmood, T.: Semaplan: Combining planning with semantic matching to achieve web service composition. In: Web Services, 2006. ICWS’06. International Conference on, IEEE (2006) 37–44
7. Rodriguez-Mier, P., Mucientes, M., Lama, M.: Automatic web service composition with a heuristic-based search algorithm. In: Web Services (ICWS), 2011 IEEE International Conference on, IEEE (2011) 81–88
8. Rodriguez-Mier, P., Mucientes, M., Vidal, J.C., Lama, M.: An optimal and complete algorithm for automatic web service composition. International Journal of Web Services Research (IJWSR) 9(2) (2012) 1–20
9. Rodriguez-Mier, P., Mucientes, M., Lama, M.: Hybrid optimization algorithm for large-scale qos-aware service composition. IEEE transactions on services computing 10(4) (2017) 547–559
10. Chattopadhyay, S., Banerjee, A., Banerjee, N.: A scalable and approximate mechanism for web service composition. In: Web Services (ICWS), 2015 IEEE International Conference on, IEEE (2015) 9–16
11. Rodriguez-Mier, P., Pedrinaci, C., Lama, M., Mucientes, M.: An integrated semantic web service discovery and composition framework. IEEE transactions on services computing 9(4) (2016) 537–550
12. Xia, Y.M., Yang, Y.B.: Web service composition integrating qos optimization and redundancy removal. In: Web Services (ICWS), 2013 IEEE 20th International Conference on, IEEE (2013) 203–210
13. 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, ACM (2012) 1968–1975
14. Chen, M., Yan, Y.: Redundant service removal in qos-aware service composition. In: Web Services (ICWS), 2012 IEEE 19th International Conference on, IEEE (2012) 431–439
15. Paolucci, M., Kawamura, T., Payne, T.R., Sycara, K.: Semantic matching of web services capabilities. In: International Semantic Web Conference, Springer (2002) 333–347

16. Rodriguez-Mier, P., Mucientes, M., Lama, M.: A hybrid local-global optimization strategy for qos-aware service composition. In: Web Services (ICWS), 2015 IEEE International Conference on, IEEE (2015) 735–738

17. Rodriguez-Mier, P., Mucientes, M., Lama, M., Couto, M.I.: Composition of web services through genetic programming. Evolutionary Intelligence 3(3-4) (2010) 171–186

18. Rao, J., Su, X.: A survey of automated web service composition methods. In: International Workshop on Semantic Web Services and Web Process Composition, Springer (2004) 43–54