

An Efficient Syntactic Web Service Composition Algorithm Based on the Planning Graph Model

Xianrong Zheng
Faculty of Computer Science
University of New Brunswick
Fredericton, NB E3B 5A3, Canada
Xianrong.Zheng@gmail.com

Yuhong Yan
Dept. Computer Science and Software Engineering
Concordia University
Montreal, QC, H3G 1M8, Canada
Yuhong@encs.concordia.ca

Abstract

In this paper, we have studied a common Web service composition problem, the syntactic matching problem, where the output parameters of a Web service can be used as the input parameters of another Web service. Many automatic Web service composition algorithms based on AI planning techniques have been proposed. However, most of them do not scale well when the number of Web services increases, or may miss finding a solution even if one exists. The planning graph, another AI planning technique, provides a unique search space. We have found that when we model the Web service composition problem as a planning graph, it actually provides a trivial solution to the problem. Instead of following the usual way to find a solution by a backward search, we put our efforts into removing the redundant Web services contained in the planning graph. Our approach can find a solution in polynomial time, but with possible redundant Web services. We have tested our algorithms on the data set used in ICEBE'05 and compared our results with existing methods.

1. Introduction

Service is defined as applying competence for the benefit of another [8]. When a single service cannot satisfy business requirements, we need to compose a group of services as a business process in order to fulfill them. In our study on the problem of Web service composition, the business requirements are described as a query with the known parameters as input and the expected parameters as output. The input and the output parameters of a Web service are defined in its WSDL file [9], and a common Web service composition problem is to match the input parameters and output parameters of Web services by name. If all input parameters of a Web service are within the output parameters

of another Web service (or a group of Web services), they can be connected. If the chained Web services accept the known parameters as input and produce the desired output parameters, it is a solution to the Web service composition problem. This process is the so-called syntactic matching problem in Web service composition [1].

The size of the search space is exponential when many available Web services exist in a Web service composition problem [6]. Therefore, efficient composition algorithms with better running time performance are preferred. Planning techniques from AI are currently the most commonly used for automatic Web service composition. When the Web service composition is modelled into STRIPS planning with restrictions on negation in pre- and post-conditions, it proves to be NP-complete [2]. Most of the automatic Web service composition algorithms based on AI planning techniques do not scale well when the number of Web services increases or may miss finding a solution, even if one exists.

However, the planning graph [3], another AI planning technique, provides a unique search space where connections between actions and propositions are expressed in a compact way. We have found that when we model the Web service composition problem into a planning graph, it provides a trivial solution to the syntactic matching problem, with possible redundant Web services. Indeed, the syntactic matching problem can be modelled into a simplified planning graph without mutual exclusion (mutex), which is partially why a planning graph is a trivial solution to the syntactic matching problem. Instead of finding a solution by a backward search in a planning graph, we put our efforts into removing redundant Web services contained in the planning graph. We have proposed several strategies to expand the planning graph with as few actions (i.e., Web services) as possible. Since a planning graph can be constructed in polynomial time, our approach provides an efficient solution to the Web service composition problem, though it may contain some redundant Web services. Though it is not an eco-

nomical solution since we may use more Web services than necessary, it is a feasible solution to the Web service composition problem. We do not attempt to provide all possible solutions, but just one that can be found in the shortest time. We have tested our algorithms on the data set used in the Web service composition contest of the IEEE International Conference on e-Business Engineering (ICEBE'05) and compared our results with the existing methods. It is an extension of our previous work on semantic Web service composition [10].

The rest of this article is organized as follows: Section 2 provides the necessary background on planning graph techniques, Section 3 demonstrates how to model the Web service composition problem based on the planning graph techniques, Section 4 shows the performance study of our proposed algorithm, Section 5 reviews some related work, and Section 6 concludes the article.

2. A brief introduction to the planning graph

Planning graph techniques are studied in the AI planning domain. A *planning graph* is a very powerful search space. Mathematically, a planning graph is a directed layered graph in which arcs are permitted only from one layer to the next. Nodes in level 0 correspond to set P_0 of propositions denoting the initial state of a planning problem. Level 1 consists of an action level, A_1 , and a proposition level, P_1 . A_1 is the set of actions (ground instances of operators) in which preconditions are nodes in P_0 , and P_1 is the union of P_0 and the sets of positive effects of actions in A_1 . An action node in A_1 is connected by incoming *precondition arcs* to preconditions in P_0 , and it is connected by *outgoing arcs* to positive or negative effects in P_1 . Outgoing arcs are labelled as *positive* or *negative*. Note that negative effects are not deleted from P_1 , thus $P_0 \subseteq P_1$. The process continues until it reaches the *fixed point* level of the graph.

A *solution plan* of a planning graph is a sequence of sets of actions $\langle \pi_1, \pi_2, \dots, \pi_k \rangle$. It is a layered plan since it is organized into levels corresponding to those of the planning graph with $\pi_i \subseteq A_i$. The first level π_1 is a subset of independent actions in A_1 that can be applied in any order to the initial state and will lead to a state that is a subset of P_1 . Actions in $\pi_2 \subseteq A_2$ proceed until the level π_k , actions lead to a state in which the goal is met [3].

In order to illustrate some important concepts of the classical planning and the planning graph, some definitions and theorems are given below.

Definition 1 In classical planning, a *state* s is a subset of a finite set of proposition symbols L , where $L = \{p_1, \dots, p_n\}$. s tells us which propositions currently hold. If $p \in s$, then p holds in the state of the world represented by s , and if $p \notin s$, then p does not hold in the state of the world represented by s .

Definition 2 In classical planning, an *action* is a triplet of subsets of L , which can be written as $a = (\text{precond}(a), \text{effects}^-(a), \text{effects}^+(a))$. The set $\text{precond}(a)$ is called the *preconditions* of a . $\text{effects}^-(a)$ and $\text{effects}^+(a)$ are called *negative effects* and *positive effects* of action a respectively, and they are disjointed, i.e., $\text{effects}^-(a) \cap \text{effects}^+(a) = \emptyset$. The action, a , is applicable to a state, s , if $\text{precond}(a) \subseteq s$.

Definition 3 In classical planning, a *state transition function* is $\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$ if action a is applicable to state s , and $\gamma(s, a)$ is undefined otherwise.

Definition 4 In classical planning, a *planning problem* is a triplet $P = ((S, A, \gamma), s_0, g)$. S is a set of states and has a property that if $s \in S$, then, for every action a that is applicable to s , the set $(s - \text{effects}^-(a)) \cup \text{effects}^+(a) \in S$. In other words, whenever an action is applicable to a state, it produces another state. A is a set of actions. γ is a state transition function. $s_0 \in S$ is the initial state. $g \subseteq L$ is a set of propositions called *goal propositions* that give the requirements that a state must satisfy in order to be a goal state. A *plan* is any sequence of actions $\pi = \langle a_1, \dots, a_k \rangle$, where $k \geq 0$.

Definition 5 In classical planning, a *planning operator* is a triplet $o = (\text{name}(o), \text{preconds}(o), \text{effects}(o))$. $\text{name}(o)$, the name of the operator, is a syntactic expression of the form $n(x_1, \dots, x_k)$, in which n is a symbol called an operator symbol, x_1, \dots, x_k are the variable symbols that appear anywhere in o , and n is unique. $\text{preconds}(o)$, the *preconditions* of o , is a set of literals which contains atoms and negations of atoms. $\text{effects}(o)$, the *effects* of o , is also a set of literals.

Definition 6 In a planning graph, if and only if action a and action b satisfy $\text{effects}^-(a) \cap [\text{precond}(b) \cup \text{effects}^+(b)] = \emptyset$, and $\text{effects}^-(b) \cap [\text{precond}(a) \cup \text{effects}^+(a)] = \emptyset$, then a and b are **independent**. A set of actions is independent when every pair of the set is independent.

Definition 7 In a planning graph, a set π of independent actions is applicable to a state s , if and only if $\text{precond}(\pi) \subseteq s$. The result of applying the set π to s is defined as: $\gamma(s, \pi) = (s - \text{effects}^-(\pi)) \cup \text{effects}^+(\pi)$, in which $\text{precond}(\pi) = \cup\{\text{precond}(a) | a \in \pi\}$, $\text{effect}^+(\pi) = \cup\{\text{effect}^+(a) | a \in \pi\}$, and $\text{effects}^-(\pi) = \cup\{\text{effects}^-(a) | a \in \pi\}$.

Definition 8 In a planning graph, a *layered plan* is a sequence of sets of actions $\langle \pi_1, \pi_2, \dots, \pi_n \rangle$, in which each π_i ($i = 1, \dots, n$) is independent. π_1 is applicable to s_0 . π_i is applicable to $\gamma(s_{i-2}, \pi_{i-1})$ when $i = 2, \dots, n$. $g \subseteq \gamma(\dots(\gamma(\gamma(s_0, \pi_1), \pi_2) \dots \pi_n))$.

Definition 9 In a planning graph, two action a and b , in level A_i are **mutual exclusion** (mutex) if either a and b are not independent or if a precondition of a is mutex with a precondition of b . Two proposition p and q in P_i are mutex if every action in A_i that has p as a positive effect is mutex with every action that produces q , and there is no action in A_i that both produces both p and q . The set of mutex pairs in A_i is denoted as μA_i , and the set of mutex pairs in P_i is denoted as μP_i .

Definition 10 In a planning graph, a **fixed point level** is a level k such that for $\forall i, i > k$, level i is identical to level k , i.e., $P_i = P_k$, $\mu P_i = \mu P_k$, $A_i = A_k$, and $\mu A_i = \mu A_k$.

Theorem 1 If a set π of independent actions is applicable to state s then, for any permutation $\langle a_1, a_2, \dots, a_k \rangle$ of the elements of π , the sequence $\langle a_1, a_2, \dots, a_k \rangle$ is applicable to s , and the state of applying π to s is $\gamma(\dots \gamma(\gamma(s, a_1), a_2) \dots a_k)$.

Theorem 2 If $\langle \pi_1, \pi_2, \dots, \pi_k \rangle$ is a solution plan, then a sequence of actions corresponding to any permutation of the elements of π_1 , followed by any permutation of π_2, \dots , followed by any permutation of π_k is a path from initial state s_0 to a goal state.

Theorem 3 If two propositions p and q are in P_{i-1} , and $(p, q) \notin \mu P_{i-1}$, then $(p, q) \notin \mu P_i$. If two actions a and b are in A_{i-1} , and $(a, b) \notin \mu A_{i-1}$, then $(a, b) \notin \mu A_i$.

Theorem 4 Every planning graph has a fixed point level k , which is the smallest k such that $|P_{k-1}| = |P_k|$, and $|\mu P_{k-1}| = |\mu P_k|$.

The planning graph techniques depart significantly from a state space approach and a plan space approach [3]. The state space approach models a plan as a sequence of actions. The plan space approach models a plan as a partially ordered set of actions. A valid plan is a sequence of actions that meets constraints of the partial order. However, the planning graph approach models a plan as a sequence of sets of actions which is more generalized than a sequence of actions while less generalized than a partial order.

Moreover, the planning graph structure provides an efficient way to estimate which set of propositions is reachable from s_0 with which actions. A goal is reachable from s_0 only if it appears in some proposition sets of the planning graph. However, this is not a sufficient condition. This weak reachability condition is compensated for by low complexity: the planning graph is of polynomial size and can be built in polynomial time.

3. Using planning graph techniques to model a syntactic Web service composition problem

3.1. Syntactic Web service composition problem

A WSDL file describes the interface of a Web service. Input and output parameters of the Web service are defined as messages in the WSDL file. Assume a Web service, w , has a set of input parameters $w_{in} = \{i_1, \dots, i_n\}$ and a set of output parameters $w_{out} = \{o_1, \dots, o_n\}$. When w is invoked with all input parameters, w_{in} , the output parameters, w_{out} , are returned.

When a single Web service cannot satisfy business requirements, we need to compose multiple Web services for fulfilling them. In this study, business requirements are expressed as a composition request.

Definition 11 A Web service composition request, r , is a pair of $\langle r_{in}, r_{out} \rangle$, where r_{in} is a set of input parameters and r_{out} is a set of output parameters.

The task of a Web service composition is to find out a chain of connected Web services in which the output parameters of a Web service can be the input parameters of another Web service (or a group of Web services), and the input and output parameters of the chain satisfy the composition request. Here, parameters are matched by name, i.e., Web service w_1 and w_2 can be connected if they satisfy $w_{in}^2 \subseteq w_{out}^1$. This is the so-called full match situation [6]. When $w_{in}^2 \cap w_{out}^1 \neq \emptyset$, $w_{in}^2 \not\subseteq w_{out}^1$, and $w_{in}^2 \not\supseteq w_{out}^1$, this is the so-called partial match situation.

Definition 12 A Web service composition request $r = \langle r_{in}, r_{out} \rangle$ is satisfied if a set of Web services $\{w_1, \dots, w_n\}$ exists such that $w_{in}^i \subseteq r_{in} \cup w_{out}^1 \cup \dots \cup w_{out}^{i-1}$ and $r_{out} \subseteq r_{in} \cup w_{out}^1 \cup \dots \cup w_{out}^n$, where $i = 1, \dots, n$.

3.2. Representing a Web service composition problem as a planning graph

A planning graph represents a procedure close to iterative deepening, which discovers a new part of the search space at each iteration. The planning graph iteratively expands itself one level at a time. The process of graph expansion continues until either it reaches a level where the proposition set contains all goal propositions or a fixed point level. The goal cannot be attained if the latter happens first. Otherwise, the planning graph searches backward from the last level of the graph for a solution.

A planning graph returns nothing if the planning problem has no solution; otherwise, it returns a sequence of sets of actions that is a solution to the problem. Additionally, every planning graph ends at a fixed point level, k , according

to Theorem 4, and only takes polynomial time to expand itself to level k . Therefore, the planning graph approach contributes greatly to the scalability of the planning problem. It also contributes greatly to the incrementality of the planning problem because of its iterative deepening feature. Finally, the planning graph approach has features of soundness, completeness, and termination.

The Web service composition problem can be mapped to a planning graph as follows:

–Each service, w , in a service composition problem is mapped to an action, a , of a planning graph. The input parameters of the service, w_{in} , are mapped to the action’s preconditions, $preconds(a)$, and its output parameters, w_{out} , are mapped to the action’s effects, $effects(a)$. All input and output parameters are positive atoms, so each service only contains positive effects, since parameters can be used many times. Thus, to be more precise, the output parameters of each service will be mapped to the action’s positive effects, but in the following section, we will still use the term effects instead of positive effects, because of its conciseness.

–The input parameters of the composition request, r_{in} , are mapped to the initial state, s_0 , of a planning problem, i.e., the proposition level P_0 in level 0 of a planning graph. Level 1 of the planning graph consists of an action level, A_1 , and a proposition level, P_1 . A_1 is a set of services where input parameters are nodes in P_0 , i.e., $A_1 = \{a | preconds(a) \subseteq P_0\}$. P_1 is a union of P_0 and the sets of effects of the services in A_1 , i.e., $P_1 = P_0 \cup \{effects(a) | a \in A_1\}$. The process continues until it reaches the fixed point level of the graph.

–The output parameters of the composition request, r_{out} , are mapped to goal propositions, g . If a planning graph reaches a proposition level which contains all required parameters, then it searches backward from the last level of the graph for a solution.

As we know, in a classical planning graph, an action may have both positive and negative effects. However, in the Web service composition problem, each Web service only has positive effects. Consequently, any parts that involve the negative effects of an action in a classical planning graph in definitions 2, 3, and 7 discussed in Section 2 should be omitted since we model each Web service as an action. In fact, the planning graph we used here is a *simplified planning graph*, but most of the properties and theorems of a classical planning graph still hold in a simplified planning graph, which facilitates analysis of its complexity. The revised definitions and theorems about the simplified planning graph are shown below.

Definition 13 A *simplified planning graph* is a planning graph where each pair of actions are independent and no mutex relations exist between actions or propositions.

Definition 14 In a simplified planning graph, a fixed point level is a level, k , such that for $\forall i, i > k$, level i is identical to level k , i.e., $P_i = P_k$ and $A_i = A_k$.

Theorem 5 Every simplified planning graph has a fixed point level, k , which is the smallest k such that $P_{k-1} = P_k$.

Example 1 A Web service composition example is shown in Figure 1. w_1 to w_4 are four Web services. The input and the output parameters for each Web service are labelled beside its box. $\{A, B, C, D\}$ and $\{K, L\}$ are the input and the output parameters of the Web service composition request. Figure 2 shows the simplified planning graph of this example. This composition problem is solvable, because the planning graph reaches P_3 , which contains the output $\{K, L\}$.

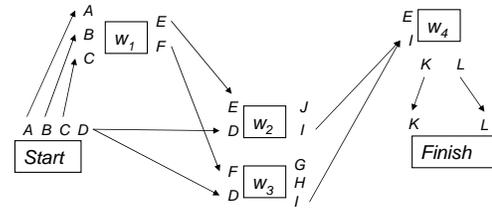


Figure 1. An example of Web service composition

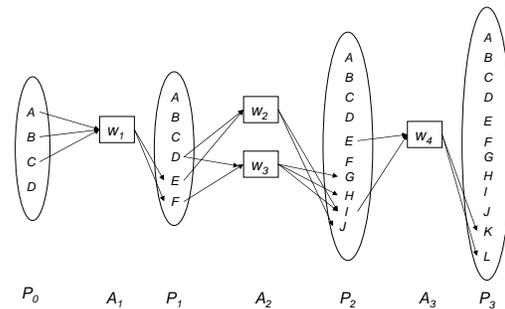


Figure 2. The simplified planning graph for the example in Figure 1

Since there is no mutex in the simplified planning graph, all the actions (i.e., Web services) in the same layer can be executed at the same time. For example, in Figure 2, w_2 and w_3 can be executed at the same time. For a solvable Web service composition problem, the simplified planning graph itself is a trivial solution, because it has a proposition set that contains g . That means for a solvable problem, we can always find a solution by constructing the planning graph.

Proposition 1 For a solvable Web service composition problem, the simplified planning graph is a trivial solution.

Example 2 The trivial solution for Example 1 is $w_1\{w_2, w_3\}w_4$, where $\{w_2, w_3\}$ means w_2 and w_3 are paralleled actions.

We can see that although the trivial solution is a feasible solution, it may contain some redundant Web services. For example, w_2 and w_3 are redundant since only one of them needs to be kept in a solution.

The classical planning graph algorithm uses a backward search for generating final solutions, which is the most time-consuming part of the planning graph techniques. Researches have been working on improving it. Instead of improving the backward search, we put our efforts into removing the redundant Web services during the construction of the planning graph. We still guarantee at least one solution will be kept after we remove these redundant Web services.

3.3. Four strategies to prune redundant Web services in a composition path

Four strategies to prune redundant Web services are shown below.

Strategy 1: void adding a Web service w' to an action set A_{i+1} if its outputs are already produced by other Web services in the same action set. In other words, we add an action only if it generates new output, because only new output makes expanding the planning graph and reaching the goals possible. Formally, for each Web service w' of A_{i+1} , if $w'_{out} \subseteq \{e|e \in w_{out}, w \in A_{i+1} \setminus w'\}$, then $A_{i+1} \leftarrow A_{i+1} \setminus w'$, where $i \geq 1$. This strategy does not remove existing solution, because it does not affect the expansion of the planning graph in the next stage.

Strategy 2: avoid adding a Web service w' to A_{i+1} if its outputs are existing in the previous proposition level P_i . Formally, $\forall w' \in A_{i+1}$, if $w'_{out} \subseteq P_i$, then $A_{i+1} \leftarrow A_{i+1} \setminus w'$, where $i \geq 1$. This strategy does not remove existing solution, because it does not affect the expansion of the planning graph in the next stage. This strategy proves to be the most effective one for pruning redundant Web services.

Strategy 3: delay adding a Web service where output is not used in the next action layer. Formally, $\forall w' \in A_i$, if $w'_{out} \cap \{e|e \in w_{in}, w \in A_{i+1}\} = \emptyset$, then $A_i \leftarrow A_i \setminus w'$. Since w' can always be added to any downstream layers, this removal will not eliminate possible solutions.

Strategy 4: stop expanding the planning graph when the goal propositions are in a proposition layer. Formally, a composition path $\langle w_1, \dots, w_k \rangle$ for A_i , let $A_i^j = w_j \cup A_i^{j-1}$ ($j = 1, \dots, k$), where $A_i^0 = \emptyset$. If j is the minimum integer such that $g \subseteq \{e|e \in w_{in}, w \in A_i^j\}$, then $A_i \leftarrow A_i^j$.

The classical planning graph adds the possibly triggered actions into the graph before it checks the termination condition. This strategy can stop the construction of the planning graph a little bit earlier.

Although our experiments prove that all the four strategies can remove most redundant Web services from a solution, it is not guaranteed that all the redundancy will be removed completely. The advantage of our approach is that the expensive search in finding a solution is avoided. In fact, we only need to construct a simplified planning graph to make our algorithm polynomial.

3.4. Web service composition algorithm based on planning graph techniques

In this subsection, we describe our algorithms in detail. Algorithm 1 is the main Web service composition algorithm based on a simplified planning graph. Its input parameters are A , s_0 , and g , where A is a set of actions, s_0 is the initial state, and g is a set of goal propositions. If a Web service composition problem has a solution, it will output it. Otherwise, no solution will be produced. Initially, s_0 will be assigned to P_0 which is a proposition set in level 0 of a simplified planning graph, G . Then, the algorithm expands G iteratively until it reaches a level where a proposition set contains g or the fixed point level of G . If the former happens first, then the algorithm will output a solution $\langle A_1, A_2, \dots, A_i \rangle$. Otherwise, the goal is not attained.

Algorithm 1 Compose(A, s_0, g)

Notes about the algorithm: $G = \langle P_0, A_1, P_2, \dots, A_i, P_i \rangle$ is a simplified planning graph.

$i \leftarrow 0, P_0 \leftarrow s_0, G \leftarrow P_0$

repeat

$G \leftarrow Expand(G)$

$i = i + 1$

until ($g \subseteq P_i \vee Fixedpoint(G)$)

if $g \subseteq P_i$ **then**

$Output(\langle A_1, A_2, \dots, A_i \rangle)$

else

$Output(\emptyset)$

if $Fixedpoint(G)$ **then**

$print(\text{"Reach fixed point"})$

Algorithm 2 shows the expansion section of Algorithm 1. Its input parameters are $\langle P_0, A_1, \dots, A_i, P_i \rangle$. It expands the planning graph to one more level $\langle P_0, A_1, \dots, A_{i+1}, P_{i+1} \rangle$. The classical way to construct a simplified planning graph is that for an action a , if $preconds(a) \in P_i$ and it is valid (not used before), a will be added to A_{i+1} and set invalid (not usable) thereafter. Then, $P_{i+1} = P_i \cup \{effects(a)|a \in A_i\}$. In order to reduce the redundancy of Web services added into the planning graph, we apply the four strategies described in Subsection 3.3.

Algorithm 2 *Expand*($\langle P_0, A_1, \dots, A_i, P_i \rangle$)

Notes about the algorithm: *valid*(a): action a can be used in the algorithm; *invalid*(a): action a can not be used in the algorithm again.

```
for  $a \in A$  do
  if  $precond(a) \subseteq P_i \wedge valid(a)$  then
     $A_{i+1} \leftarrow A_{i+1} \cup a$ 
     $invalid(a)$ 
 $P_{i+1} \leftarrow P_i$ 
for  $a \in A_{i+1}$  do
  if  $effects(a) \not\subseteq P_{i+1}$  then
     $P_{i+1} \leftarrow P_{i+1} \cup effects(a)$ 
  else
     $A_{i+1} \leftarrow A_{i+1} \setminus a$ 
     $invalid(a)$ 
  if  $g \subseteq P_{i+1}$  then
    discard all the other actions in  $A_{i+1}$ 
    break
for  $a \in A_i$  do
  if  $effects(a) \cap \{precond(a) | a \in A_{i+1}\} = \emptyset$  then
     $A_i \leftarrow A_i \setminus a$ 
     $valid(a)$ 
return  $\langle P_0, A_1, \dots, A_{i+1}, P_{i+1} \rangle$ 
```

In Algorithm 2, we first generate a new action layer A_{i+1} and initialize $P_{i+1} \leftarrow P_i$ (the first *for* loop). In the second *for* loop, we try to apply Strategy 1 and 2. For an action $a \in A_{i+1}$, if $effects(a) \not\subseteq P_{i+1}$ which means a generates new outputs compared with the parameters in P_{i+1} , we keep a inside A_{i+1} and $P_{i+1} \leftarrow P_{i+1} \cup effects(a)$. Otherwise, if $effects(a) \subseteq P_{i+1}$, we remove a from A_{i+1} , i.e., $A_{i+1} \leftarrow A_{i+1} \setminus a$. a is set to be invalid thereafter since we will not use it again. Strategy 4 is used in the third *if* statement. If after adding an action, a , we find that $g \subseteq P_{i+1}$, we discard all the unchecked actions in A_{i+1} and quit the second *for* loop. In the third *for* loop, we try to apply Strategy 3. For each action $a \in A_i$, if $effects(a) \cap \{precond(a) | a \in A_{i+1}\} = \emptyset$, a will be removed from A_i since it has no effect in generating A_{i+1} . a is set to be valid in order to be used again.

Algorithm 3 *Fixedpoint*($\langle P_0, A_1, \dots, P_i, A_i \rangle$)

```
if  $P_i = P_{i-1}$  then
  return true
else
  return false
```

Algorithm 3 demonstrates the fixed point section of algorithm 1. Its input parameters are $\langle P_0, A_1, \dots, A_i, P_i \rangle$, and its output parameter is a boolean value. If $P_i = P_{i-1}$, the algorithm will return true. Otherwise it will return false.

3.5. Algorithm analysis

Let (O, s_0, g) be a planning problem, where O is a set of operators (each action is a ground operator), s_0 is the initial state, and g is a set of goal propositions. Suppose that the planning problem has n propositions and m actions. It follows that the number of elements that are contained in every proposition set and action set are no more than n and m respectively. We also suppose that c is the number of constant symbols of the problem, $e = \max_{o \in O} \{|effects^+(o)|\}$, and α is an upper bound on number of parameters of any operator. Then, the number of actions produced by each operator is no more than c^α , therefore the maximum number of actions produced by all operators is $|O| \times c^\alpha$, which means $m \leq |O| \times c^\alpha$. Since the number of propositions produced by each operator is no more than e , the number of propositions produced by each action is also no more than e . Furthermore, the total number of propositions produced by all actions is no more than $e \times |O| \times c^\alpha$. As a result, it follows that $n \leq e \times |O| \times c^\alpha$. So, n and m are polynomial.

Every simplified planning graph has a fixed point level, k , according to Theorem 5. The reason for this is that its proposition sets and action sets monotonically increase from one level to the next, i.e., $P_{i-1} \subseteq P_i$ ($i = 1, \dots, k$), $A_{i-1} \subseteq A_i$ ($i = 2, \dots, k$), while the number of elements that are contained in every proposition set and action set is bounded; thus, every simplified planning graph must reach a fixed point level. Additionally, the main work of Strategy 1, 2, 4 is to check whether a subset relation exists between two sets or not, and that of Strategy 3 is to check whether two sets have common elements or not. Both can be done in polynomial time. Therefore, the complexity of the simplified planning graph is polynomial, because it has finite distinct levels and the number of elements that are contained in every proposition set and action set is no more than n and m , which are both polynomial.

4. Performance study

We used the sample repository and challenge of ICEBE'05 to study the performance of the Planning Graph based Service Composition Algorithm (PGSCA). The sample repository and challenge can be downloaded from [1]. All requests in the data set can be solved by using full-matching composition. Consequently, some simple algorithms may work. However, it should be noted that PGSCA can address both full- and partial matching Web service composition problems. Running time is used as the metric to compare the performance between PGSCA and the Service Composition Algorithm (SCA) used in Georgetown Java Software, which can also be downloaded from [1].

The main idea of SCA is to build a new chain to record a possible solution path for a Web service which can be

served as the first element of the solution path. The output parameters of the Web service will be added to a set of available parameters for the chain. Then the algorithm scans the whole Web service repository to find whether there is a new Web service in which input parameters are in the set of available parameters for a chain. If so, the Web service will be added to the chain and its output parameters will be added to the set of available parameters for the chain. The algorithm continues to scan the repository and may add a new Web service to a chain again. If it cannot find such a Web service in one scan, the algorithm will terminate immediately.

The idea of SCA is clear, but it is not an efficient algorithm because it scans the Web service repository again and again until it cannot find a new Web service that can be added to a chain. In other words, it does not tell us under what conditions it should be terminated. Moreover, SCA may leave out some solutions since it does not consider all possible composition paths. However, PGSCA will terminate when it reaches the fixed point level of a simplified planning graph. In addition, it can find the composition path if the Web service composition problem has a solution, though it may not be a concise one.

4.1. Test cases

The sample repository that we used have 143 Web services. Each Web service is stored as a WSDL file. An example of such a file is shown below:

```
< messageName = "ServiceP01a1182959_Request" >
< partName = "P19a0620411" type = "xsd:string" / >
< /message >
< messageName = "ServiceP01a1182959_Response" >
< partName = "P82a3034588" type = "xsd:string" / >
< /message >
< portTypeName = "ServiceP01a1182959_Port" >
< operationName = "ServiceP01a1182959" >
< inputMessage = "tns:ServiceP01a1182959_Request" / >
< outputMessage = "tns:ServiceP01a1182959_Response" / >
< /operation >
< /portType >
```

The composition requests are stored as XML files which are shown below:

```
< WSCheck >
< CompositionRoutineName = "C1" >
< Provided >
P88a1468148, P11a4043551, P68a6331952, P96a3868936,
P44a8641788, P61a7652575, P05a8359303, P81a4931581,
P71a7343790, P96a7526345
< /Provided >
< Resultant > P66a0567777 < /Resultant >
< /CompositionRoutine >
< CompositionRoutineName = "C2" >
< Provided > P97a7397147, P24a2859947 < /Provided >
< Resultant > P81a6362805 < /Resultant >
< /CompositionRoutine >
< CompositionRoutineName = "C3" >
< Provided > P19a0620411 < /Provided >
< Resultant > P24a8052504 < /Resultant >
< /CompositionRoutine >
< CompositionRoutineName = "C4" >
< Provided > P38a6536687, P22a9481780 < /Provided >
< Resultant > P68a9119134 < /Resultant >
```

```
< /CompositionRoutine >
< /WSCheck >
```

The above composition requests include four sub requests, i.e., $C1$, $C2$, $C3$, and $C4$. It should be noted that each of them has a solution. All experiments were performed on a PC platform with a Pentium 4 CPU (3.00 GHz), Windows XP, and 512 MB RAM. All algorithms were implemented in Java. For each test case, PGSCA and SCA will be executed 20 times and the average running time will be reported.

4.2. Result analysis

Figure 3 shows the comparison of running times between PGSCA and SCA under test case $C1$. In it, there are ten provided parameters and one required parameter. PGSCA produces a concise solution with 4 services which are exactly the same ones that are contained in the correct solution, while SCA produces a solution which contains 22 services of which 18 are redundant. This case demonstrates that SCA may have some redundant solutions. As shown in Figure 3, the running time of PGSCA is less than that of SCA in most cases. It should be noted that it takes more time to find a solution when we first run PGSCA or SCA. The reason for this is that it takes some extra time to perform some initialized work the first time. Thus, we omit the extra time in Figures 3, 4, and 5. The average running time of PGSCA is 2405 milliseconds while the average running time of SCA is 2471 milliseconds. The average running time of PGSCA is 66 milliseconds less than that of SCA in test case $C1$ which demonstrates that our proposed algorithm is more efficient than SCA.

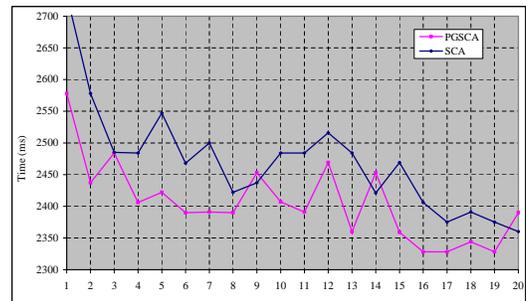


Figure 3. Test case $C1$

Figure 4 shows a comparison of running time between PGSCA and SCA under test case $C2$. In it, there are two provided parameters and one required parameter. Each algorithm produces a concise solution with 4 services which are exactly the same ones that are contained in the correct solution. Web services in two solutions and their arrange order are the same. In most cases, the running time of

PGSCA is less than that of SCA. The average running time of PGSCA is 2413 milliseconds while the average running time of SCA is 2446 milliseconds. The average running time of PGSCA is 33 milliseconds less than that of SCA in this test case, which demonstrates that our proposed algorithm is better than SCA.

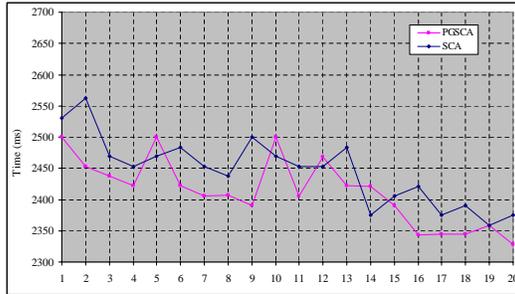


Figure 4. Test case C2

There is one provided parameter and one required parameter in test case C3. PGSCA produces a solution with 8 services while the correct solution only has 3 services, which means that PGSCA produces 5 redundant services. Unfortunately, SCA cannot find a solution. The average running time of PGSCA is 2411 milliseconds. This case demonstrates that SCA may leave out some solutions, but PGSCA does not.

Figure 5 shows a comparison of running times between PGSCA and SCA under test case C4. In it, there are two provided parameters and one required parameter. Each algorithm produces a concise solution with 2 services which are exactly the same ones that are contained in the correct solution. Web services in two solutions and their arrange order are the same. In most cases, the running time of PGSCA is less than that of SCA. The average running time of PGSCA is 2396 milliseconds, yet the average running time of SCA is 2423 milliseconds. Our proposed algorithm is still better than SCA in this case, though its advantage is not obvious. A possible reason for this is that there are only two services in the solutions, which are not enough to show the scalability of PGSCA.

5. Related work

An automated Web service composition algorithm is critical to the success of Web service applications. Several methods have been proposed to attain this goal. Most of them can be classified as workflow based techniques and AI planning techniques [7]. Other methods include Algebraic Process, π -Calculus, Petri Nets, Model Checking, and Finite-State Machines techniques [4].

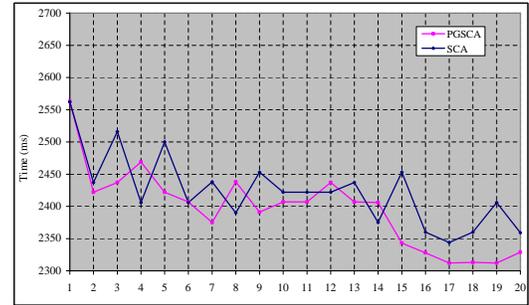


Figure 5. Test case C4

In workflow-based techniques, the composite Web services are defined as similar to workflows which consist of several Web services with control and data flow. Several industry standards use this method such as BPEL (Web service Business Process Execution Language), BPML (Business Process Execution Language), and HP's eFlow project [11].

In AI planning techniques, the provided parameters and required parameters of a Web service requester are modelled as the initial states and goal states of a planning problem respectively. An available Web service set in each step is modelled as an action set of the planning problem. The input parameters and output parameters of each Web service are modelled as the preconditions and effects of a corresponding action respectively. Several AI planning techniques are used to solve the service composition problem such as situation calculus, Planning Domain Description Language (PDDL), rule-based planning, theorem proving [7].

Web service composition techniques must satisfy several requirements: connectivity, non-functional quality-of-service properties, correctness, and scalability [4]. In complicated business transactions, multiple services are likely to be used in a complex invocation chain. Therefore, scalability is most important to Web service composition algorithms. π -Calculus offers concise notation and thus facilitates specification of complex services. However, most Web service composition techniques cannot scale with the number of composed Web services. An AI-planning based Web service composition algorithm named WSPR is presented in [5]. It computes the cost of achieving individual parameters starting from w_{in} by using a forward search. Then it approximates the optimal sequence of Web services that connects w_{in} to w_{out} by using regression search leveraging on the results that are obtained from the first step, as guidance. WSPR may leave out a solution, since it uses a heuristic search algorithm in the regression search.

6. Conclusions

In this paper, we present an efficient syntactic Web service composition algorithm based on a simplified planning graph. For a solvable problem, it can find a solution in polynomial time, but with possible redundant Web services. We apply four strategies to make the solution concise. This approach opens a new path to improve the planning graph when we solve the Web service composition problem. We will further study how to use effective strategies to solve the redundancy problem.

References

- [1] <http://ws-challenge.org>.
- [2] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2), 1994.
- [3] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning Theory and Practice*. Morgan Kaufmann Publishers, San Francisco, 2004.
- [4] N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing*, 8(6), 2004.
- [5] S. Oh, D. Lee, and S. Kumara. Web Service Planner (WSPR): An Effective and Scalable Web Service Composition Algorithm. *International Journal of Web Services Research*, 4(1), 2007.
- [6] S. Oh, B. On, E. Larson, and D. Lee. BF*: Web Services Discovery and Composition as Graph Search Problem. In *IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE05)*, pages 784–786, 2005.
- [7] J. Rao. *Semantic Web Service Composition via Logic-based Program Synthesis*. PhD thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, 2004.
- [8] B. Stauss, K. Engelmann, A. Kremer, and A. Luhn. *Services Science: Fundamentals, Challenges and Future Developments*. Springer, Berlin, 2007.
- [9] W3C. WSDL Specification, 2003, retrieved in 2004. <http://www.w3.org/TR/wsdl/>.
- [10] Y. Yan and X.R. Zheng. A Planning Graph Based Algorithm for Semantic Web Service Composition. In *IEEE Joint Conference on E-Commerce Technology (CEC'08) and Enterprise Computing, E-Commerce and E-Services (EEE'08)*, accepted, 2008.
- [11] T. Yu, Y. Zhang, and K.J. Lin. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. *ACM Transactions on the Web*, 1(1), 2007.