QoS-aware Service Composition over Graphplan through Graph Reachability

Min Chen
Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Canada
Email: minchen2008halifax@yahoo.com

Yuhong Yan
Dept. of Computer Science and Software Engineering
Concordia University
Montreal, Canada
Email: yuhong@encs.concordia.ca

***Abstract*—QoS-aware service composition is a bi-objective task for the generation of a business process: to fulfill functional goals and to optimize the QoS criteria. Planning algorithms are frequently used for the generation of a business process to achieve functional goals. In this paper, we use a planning algorithm, GraphPlan, and a graph search algorithm, Dijkstra's algorithm, to achieve both functional goals and QoS optimization at the same time. Firstly, we analyze graph reachability in the planning graph built by Graphplan algorithm. Taking advantage of graph reachability, we propose an approach of using Graphplan technique combined with Dijkstra's algorithm to solve QoS-aware service composition problem. The experiments show our approach is able to find the optimal solution for different QoS criteria. Moreover, our approach reduces the possibilities of combinatorial explosion to a large degree when exploring the graph for the optimal path.**

***Keywords*-QoS optimization; Web service composition; Graph reachability; Dijkstra's algorithm; Graphplan;**

I. Introduction

Web services are self-described software entities which are posted across the Internet using a set of open standards such as SOAP [17], WSDL [18], and UDDI [10]. Automated Service Composition (ASC) is to automatically generate a business process to fulfill business goals that cannot be fulfilled by individual services. In addition to satisfying business goals, QoS-aware service composition requires the target business process is QoS optimized.

ASC is studied under different assumptions [14]. The most useful and practical problem is to connect SOAP services into a network by matching their parameters, so that this network of services can produce a set of required output parameters given a set of input parameters. This is the composition problem studied in this paper. AI Planning algorithms are predominant algorithms to solve ASC problems [13], [25]. The study in this paper is based on an AI planning algorithm called Graphplan [2]. Graphplan technique first builds a planning graph to represent the problem space. Then, Graphplan finds a plan through backtracking the planning graph. Without considering QoS optimization, the planning algorithms normally search for a shortest plan on the assumption that each action has the same unit (execution) time/cost.

Considering QoS optimization during service composition, QoS-aware service composition selects the best services to optimize the QoS of the target business process. In such case, a shortest plan may not be the preferred one because a longer plan may have faster response time, or less cost, or higher throughput. Morever, the way to reach each vertex in the planning graph is different from that in a normal graph. Hence, we need to modify the classic planning algorithm for the ease of using graph search algorithms, *e.g.,* Dijkstra's algorithm, to find a QoS optimized solution.

On the other hand, people also use algorithms like Dynamic Programming and Integer Programming to find a QoS optimized solution. However, these generic algorithms are not tuned for planning. Some of the studies do not consider reusing the same actions in the plan, as AI planning algorithms can.

People tend to solve the QoS-aware service composition problem in two steps. First, the control flow is chosen which becomes the template of the target business process; then the services are selected for each tasks in the control flow. This makes us facing a NP-complete problem of the so-called service selection problem. If we are not constrained by the chosen template, it is possible that other control flows with other services can have better QoS. Therefore, we want to solve the composition problem and the service selection at the same time, which means the two-step approach is not necessary.

In this paper, we combine planning algorithm called Graphplan with Dijkstra's algorithm to find a QoS-optimized plan. Our motivation is to extend Graphplan algorithm from the ASC solver to the QoS-aware service composition solver with the help of Dijkstra's algorithm. Moreover, we propose a way of changing graph reachability to ease the use of the Dijkstra's algorithm. Our approach not only keeps the advantages of AI planning algorithms, such as easy to model business logic, reuse of actions in a plan and parallel actions in a plan, but also gets a globally QoS optimized solution for different QoS criteria.

The paper is organized as follows. Section 2 gives background knowledge of QoS criteria, Graphplan technology, and Dijkstra's algorithm. Section 3 presents our solution to solve QoS-aware service composition problem over Graph-

plan through graph reachability. The experimental results are presented in Section 4. Related work is reviewed in Section 5. We end up with a conclusion in Section 6.

II. Preliminaries

A. The QoS-aware service composition problem

We take the services as stateless black boxes (no conversations) in this paper. The services expose themselves in WSDL descriptions which do not include state information [1] We can also associate semantic information to inputs and outputs using SAWSDL [16] or OWL [15].

Definition 1: Given a set D of concepts, a **service** w is a tuple $(in(w), out(w), Q(w))$, where $in(w) \subseteq D$ (resp. $out(w) \subseteq D$) denote the inputs (resp. the outputs) of w, and $Q(w)$ is a finite set of quality criteria for w.

The above definition assumes each service has one operation. For a service w with n operations $o_1, \ldots, o_n$ we may do as if we had n services $w : o_1, \ldots, w : o_n$. We use $\sigma = w_1, w_2, \ldots, w_n$ to represent a network of connected services. If they are connected in sequence, $\sigma = w_1; w_2; \ldots; w_n$, or in parallel, $\sigma = w_1||w_2|| \ldots ||w_n$.

Generally speaking, commonly used quality critera are response time, throughput, execution price, reputation and availability [4], [1]:

- **Response time** $Q_1(w)$: the time interval between the receipt of the end of transmission of an inquiry message and the beginning of the transmission of a response message to the station originating the inquiry.

$$Q_1(w_1; \ldots; w_n) = \sum Q_1(w_i) \tag{1}$$

$$Q_1(w_1|| \ldots ||w_n) = \max Q_1(w_i) \tag{2}$$

- **Throughput** $Q_2(w)$: the average rate of successful message delivery over a communication channel, *e.g.,* 10 successful invocations per second.

$$Q_2(w_1; \ldots; w_n) = \min Q_2(w_i) \tag{3}$$

$$Q_2(w_1|| \ldots ||w_n) = \min Q_2(w_i) \tag{4}$$

- **Execution price** $Q_3(w)$: the fee to invoke w.

$$Q_3(w_1, \ldots, w_n) = \sum Q_3(w_i) \tag{5}$$

- **Reputation** $Q_4(w)$: a measure of trustworthiness of w.

$$Q_4(w_1, \ldots, w_n) = \frac{1}{n} \sum Q_4(w_i) \tag{6}$$

- **Successful execution rate** $Q_5(w)$: the probability that w responds correctly to the user request.

$$Q_5(w_1, \ldots, w_n) = \prod Q_5(w_i) \tag{7}$$

[1]For stateful services, we have developed a modeling technique to convert the sequential orders of each operation into its preconditions and postconditions [13]. With that technique, we can treat the stateful and stateless services in a uniform way.

- **Availability** $Q_6(w)$: the probability that w is accessible.

$$Q_6(w_1, \ldots, w_n) = \prod Q_6(w_i) \tag{8}$$

Definition 2: **A service composition problem** is a tuple $(W, D_{\text{in}}, D_{\text{out}}, Q)$, where W is a set of services, D_{in} are provided inputs, D_{out} are expected outputs, and Q is a finite set of quality criteria.

Some of the above criteria, *e.g.,* response time and execution price, are negative, *i.e.*, the higher the value, the lower the quality. Therefore, as for the single quality criterion of response time or execution price, the minimum QoS value indicates the best quality. The other criteria are positive, *i.e.,* the higher the value, the higher the quality. Throughput, reputation, successful execution rate, and availability are in this category. As for any of such quality criteria, the maximum QoS value indicates the best quaility. In Section III, we use response time as the single QoS criterion to present our algorithm. Hence, the minimum QoS value of response time is the optimization objective for our algorithm. We will discuss other single QoS criterion in Section III-F.

B. Graphplan technique

AI planning [3] has been applied with success to static service composition [12], among others due to its support for under-specified composition requirements which are well-suited to end-user composition. The study in this paper is based on an AI planning algorithm called Graphplan [2]. Recent works have demonstrated the suitability of this model for ASC [13], [25].

The Graphplan approach contains two phases. The planning graph construction phase builds the planning graph from P_0. The graph construction algorithm stops when the goal is reached or a fixpoint is reached. The complexity of this algorithm is polynomial [2]. If the goal is reached, this means the problem has a solution. The second phase is to extract a solution using backward search from the goal layer. Normally the second algorithm is more costly. In the most general case, *i.e.,* if the problem has negative effects, the backward search phase may require backtracking and the complexity is NP-complete. If the problem has only positive effects, we see that backtracking is not needed [5]. We want to get a minimal set of services that can solve the problem.

Definition 3: **The classic planning graph** $G = (V_A \cup V_P, E)$ is a Directed Acyclic Graph (DAG) where V_A is the vertices representing actions and V_P representing propositions. Edges $E = (V_P \times V_A) \cup (V_A \times V_P)$ connect the vertices. The edges $(V_P \times V_A)$ connect the input parameters with the actions, while $(V_A \times V_P)$ connects the actions with their output parameters.

Following [25], it is possible to map a service composition problem $(W, D_{\text{in}}, D_{\text{out}}, Q)$ to a planning problem $P = ((S, W, \gamma), D_{\text{in}}, D_{\text{out}})$ with service inputs being mapped to action preconditions $(in(w) \mapsto pre(w))$ and outputs to positive effects $(out(w) \mapsto effects^{+}(w))$. Plans

can be encoded in any orchestration language with assignment, sequence, and parallel operators, *e.g.,* WS-BPEL [11]. Additionally, planning graphs enable to retrieve plans with parallel invocations, which can be encoded using parallel operations (WS-BPEL flow).

C. Dijkstra's algorithm

Dijkstra's algorithm's goal is to find single-source shortest paths in a graph [9]. In a graph, each vertex is always reachable from other vertices. Making use of the reachability among vertices, Dijkstra's algorithm does a systematic search in a graph. If the graph is finite, systematic search means that the algorithm will visit every reachable vertex, and keep track of vertices already visited to avoid infinite loop when the graph has cycles. If the graph is infinite, systematic search has a weaker definition. If a solution exists, the search algorithm still must report it in finite time; however, if a solution does not exist, it is acceptable for the algorithm to search forever. It is known that the planning graph is finite and it takes polynomial time to construct the planning graph. Thus, we are dealing with finite graph.

Suppose a graph $G = (V, E)$ has every edge $e \in E$ labeled with a distance $d(e)$. Assume the edge e is from a vertex v, we can also write in the state-space representation $d(v, e)$. For each vertex v, we define a *cost-to-come* function $C : V \rightarrow [0, \infty]$. For each vertex, the value $D^*(v)$ is called the optimal *cost-to-come* from the initial vertex v_I. This optimal value is obtained by summing edge distance, over all possible paths from v_I to v and using the path that produces the least cumulative distance. If the cost is not known to be optimal, then it is written as $D(v)$. During the search, the cost-to-come value $D(v)$ is updated once the new path has a lower value is found. Also, the path to produce this optimal value is recorded every time.

The complexity of Dijkstra's algorithm over a Graph $G = (V, E)$ is $O(|V|^2)$ without using min-priority queue. The common implementation based on a min-priority queue implemented by a Fibonacci heap and running in $O(|E| + |V|log|V|)$ is due to (Fredman & Tarjan 1984). More examples and the pseudo code can be found in [19].

III. QoS-aware Service Composition over Graphplan through Graph Reachability

A. Taking advantage of graph reachability

Dijkstra's algorithm takes advantage of graph reachability to traverse the graph.

Definition 4: **Graph reachability** is the ability to reach one vertex from other vertices within a graph.

If a vertex is able to be reached by one vertex, we call it simple graph reachability. There also exists the case that a vertex is reachable from several parents vertices together when these parent vertices all are available. In such case, we call it complex graph reachability.

In the planning graph, actions and propositions can be considered as vertices. The QoS value can be mapped to *cost-to-come* value of the vertex. It is possible to use Dijkstra's algorithm to solve QoS-aware service composition problem through GraphPlan algorithm. Dijkstra's algorithm will search a plan with the optimal QoS value.

However, we need to overcome some difficulties. First of all, a suitable formula to calculate the *cost-to-come* value for each vertex is important. It is because the QoS value is mapped into the *cost-to-come* value. The calculation of QoS value for different QoS criteria follows different formula. Secondly, two types of vertices in the planning graph, *i.e.,* action vertices and proposition vertices, lead to complex graph reachability. An action vertex and a proposition vertex correspond to a service and an input/output parameter accordingly. A service is invokable when all its inputs are available. Edges link the input parameters of a service to this service itself. Hence, an action vertex may be reachable from several propostions at the same time. Complex graph reachability makes it difficult to apply Dijkstra's algorithm. We need to find a way to change the graph reachability in planning graph for the ease of using Dijkstra's algorithm.

B. Our solution to solve QoS-aware service composition

Our idea is to combine planning algorithm called Graphplan with the Dijkstra's algorithm to find a QoS-optimized solution for QoS-aware service composition. We realize our idea in three steps.

First, a Partially Labelled Planning Graph (PLPG) is gererated to represent the problem space. The PLPG extends the classic planning graph in the way that each proposition is associated with a label.

Definition 5: A **label** T is a set of triples $\{(a, L_a, C_a)\}$. For each triple (a, L_a, C_a), a is an action enabled at action layer A_k ($k = L_a$), L_a is the layer number, and C_a is a real number representing the cost for a to be enabled.

Definition 6: The **Partially Labelled Planning Graph** (PLPG) is a Planning Graph $G = (V_A \cup V_P, E)$ where each proposition $p \in V_P$ is associated with a label T.

On the one hand, the PLPG records the QoS value in the label for the future calculation. On the other hand, the PLPG makes it ready for the graph conversion in the next step.

Next, a Layered Weighted Graph (LWG) is converted from the PLPG. The purpose of the PLPG is to simplify graph reachabilty for the ease of using Dijkstra's algorithm. The LWG takes actions in the PLPG as bridges to connect the propositions in the actions' preconditions to the propositions in the actions' effects. In such way, the LWG makes the type of vertices uniform.

Definition 7: A **vertex** v is a tuple of a set of propositions P_v and a label T_v, *i.e.,* $v = (P_v, T_v)$, where

1. $P_v \subseteq \{effects(a) | \exists a : (a, L_a, C_a) \in T_v\}$;
2. $\forall (a, L_a, C_a) \in T_v$:
   (a) $effects(a) \cap P_v \neq \emptyset$;

(b) $\nexists(a', L_{a'}, C_{a'}) \in T_v - \{(a, L_a, C_a)\}$:
$effects(a') \subseteq effects(a)$ or
$effects(a) \subseteq effects(a')$.

Definition 7 shows the label T_v keeps a record of the reachability for the propositions P_v in v. In Definition 7, condition 1 requires each proposition in P_v is able to be reached by at least one action in T_v. Condition 2(a) makes sure actions in T_v only comes from the labels of propositions in P_v. Condition 2(b) guarantees no action can be removed from T_v. Since more actions to be enabled indicate more execution price, condition 2(b) avoids unnecessary cost of extra services in T_v.

Definition 8: The **Layered Weighted Graph** (LWG) is a layered graph where the layers of vertices V_i ($0 \leq i \leq n$), edges E_i ($1 \leq i \leq n$), and weights W_i ($1 \leq i \leq n$) form an alternating sequence $(V_0, E_1, W_1, V_1, \ldots, E_n, W_n, V_n)$. For the layer E_i, an edge $(v', v) \in E_i$ ($1 \leq i \leq n$) links vertex v' in V_{i-1} with v in V_i. For the layer W_i, a weight $w(v', v) \in W_i$ is a function $w(v', v) : (v', v) \rightarrow T_{(v',v)}$ mapping an edge (v', v) in E_i to a label $T_{(v',v)}$ where $T_{(v',v)} = \{(a, L_a, C_a) | \exists (a, L_a, C_a) \in T_v : L_a = i\}$.

In the last step, we first use Dijkstra's algorithm to calulate the *cost-to-come* value for each vertex in the LWG. Then, we obtain the plan with the best *cost-to-come* value through backtracking. The QoS value of a proposition is the accumulative QoS value of a set of chained services to be invoked for obtaining this proposition. There are probably serveral ways of obtaining one proposition. The best QoS value is selected as the *cost-to-come* value for a proposition. The *cost-to-come* value for a vertex $v = (P_v, T_v)$ is the best QoS values of the propositions in P_v. In Section III, we use response time to present our algorithm. The minimum response time implies the best *cost-to-come* value. We will discuss other single QoS criterion in Section III-F.

Algorithm 1 is the main algorithm. First, we build a partially labelled planning graph G to represent the problem space (line 1). Then, we check if G can achieve all goals (line 2-3). If all goals are achieved, G is converted into a layered weighted graph GC (line 4) and a plan with the best *cost-to-come* value is generated (line 5).

Algorithm 1 $QoSWSC(A, s_0, g)$

1: $G = QoSGraphPlan(A, s_0, g)$;
2: $n = max\{i | P_i \in G\}$;
3: **if** $g \subseteq P_n$ and g generated by non no-ops **then**
4: $GC = GraphConversion(G, s_0, g)$;
5: $\pi = PlanGeneration(GC)$;
6: **print** π;
7: **end if**
8: **if** $g \not\subseteq P_k$ **then**
9: **print** $\emptyset$;
10: **end if**

C. Planning graph labelling

Algorithm 2 builds the PLPG layer by layer. Originally, we assume all the given propositions are the effects of a dummy service I and the cost of I is 0 (line 1). Line 4 to line 16 generate action layer A_i and proposition layer P_i until a fixed point is detected. $Fixedpoint(G)$ is a function to check if a fixed point layer is reached. A fixed point in LPG is a layer k such that for $\forall i (i \geq k)$, $A_i = A_k$ and $P_i = P_k$. Line 5 gets all the enabled actions for action layer A_i. The enabled actions are those whose inputs are in the previous proposition layer P_{i-1}. SA is a set of re-used or newly invoked actions contained in A_i (line 6). SP is a set of propositions that are the effects of SA (line 7). P' is a set of tuples (line 8). Each tuple consists of a proposition in SP and the proposition's label. For the propositions contained both in P_{i-1} and SP, the label of each proposition is the union of the label in P_{i-1} and that in SP (line 9). P_i consists of propositions with their corresponding labels in $P_{i-1} - P'$, $P' - P_{i-1}$, and P'' (line 10). Line 12 and 13 create arcs between actions and propositions.

Algorithm 2 $QoSGraphPlan(A, s_0, g)$

1: $P_0 = \{(p, T_p^0) | p \in s_0, T_p^0 = \{(I, 0, 0)\}\}$;
2: $SP = s_0$;
3: $i = 1$;
4: **repeat**
5: $A_i = \{a | pre(a) \subseteq P_{i-1}, a \in A\}$;
6: $SA = \{a | a \in A_i \wedge pre(a) \cap SP \neq \emptyset\}$;
7: $SP = \bigcup_{a \in SA} effects(a)$;
8: $P' = \{(p, T) | \exists p \in SP : T = \{(a, L_a, C_a) | p \in effects(a) \wedge L_a = i\}\}$;
9: $P'' = \{(p, T) | \exists (p, T_1) \in P_{i-1}, (p, T_2) \in P' : T = T_1 \cup T_2\}$;
10: $P_i = (P_{i-1} - P') \cup (P' - P_{i-1}) \cup P''$;
11: **for** each $a \in A_i$ **do**
12: link a with precondition arcs to $pre(a)$ in P_{i-1};
13: link a with to each of its $effects(a)$ in P_i;
14: **end for**
15: $i = i + 1$;
16: **until** $Fixedpoint(G)$
17: **return** $\langle P_0, A_1, ..., A_n, P_n \rangle$;

Example 1: A set of available services with their input/output parameters and response time in milliseconds are listed in Table I. The composition query is (D_{in}, D_{out}) = $(\{A, B\}, \{D, G\})$. We construct a PLPG as in Figure 1.

In Figure 1, we do not draw the no-op actions. It is because a no-op action inherits a true proposition from a previous proposition layer and has no cost. According to Algorithm 2, we calculate SA at each action layer (the shaded actions in Figure 1). To make the figure readable, we only draw the arcs connecting to or from the shaded actions in the action layers. Please notice that the graph

reaches the fixed point at layer A_3. There are four solutions: $\{w_1; w_3, w_4\}$, $\{w_1, w_2; w_3; w_4\}$, $\{w_1, w_2; w_4, w_5; w_3\}$, and $\{w_2; w_4, w_5; w_3\}$.

Table I
A SET OF AVAILABLE SERVICES

w_i	inputs	outputs	Q_1	w_i	inputs	outputs	Q_1
w_1	A	C, E	120	w_4	E	G	10
w_2	A, B	E, J	30	w_5	B, J	C	70
w_3	C	D	50				

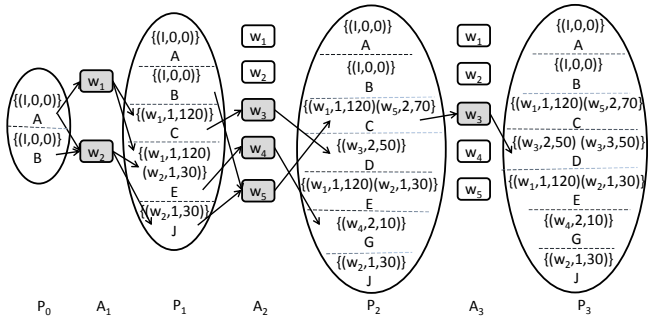

Figure 1. The partially labelled planning graph for Example 1.

D. Graph conversion

Algorithm 3 describes how the PLPG is converted into the LWG. $FindCmb(P_n, g)$ is a function that generates a set of labels for g according to layer P_n (line 1). Each vetex in V_n consists of all goals and a label generated by $FindCmb(P_n, g)$ (line 2). Starting from layer V_n, the LWG is built layer by layer until V_1 is generated (line 4-19). According to the vertex layer V_i, Algorithm 3 generates the vertex layer V_{i-1}, the edge layer E_i, and the weight layer W_i with the help of the proposition layer P_{i-1} and a function $LayerUpdate(E_i, W_i, w, v, V')$. Function $LayerUpdate(E_i, W_i, w, v, V')$ adds newly generated edges and weights into E_i and W_i accordingly (Equation 9):

$$\begin{aligned} \forall v' \in V' : & E_i = E_i \cup \{(v', v)\}, \\ & W_i = W_i \cup \{(v', v) \rightarrow w\} \end{aligned} \tag{9}$$

For a vertex v in V_i, we obtain a label w from the label T_v. w contains a set of actions which are enabled at layer i (line 7). If w is an empty set (line 8), it means no proposition in v is able to be reached at layer i. We create a copy of v called v' in the vertex layer V_{i-1} as the parent of v, since propositions in v can be reached at lower layers (line 9-10). An edge (v', v) and a weight $(v', v) \rightarrow w$ are created in E_i and W_i reprectively (line 11). $VGeneration(P_{i-1}, w, v)$ is a function to generate all possible parents vertices for v according to P_{i-1} and w. If w is not empty (line 12), function $VGeneration(P_{i-1}, w, v)$ is called to get parent vertices V' that can reach v through actions in w (line 13). Afterwards, the layer V_{i-1}, E_i and W_i are updated (line 14-15). V_0 has only one vertex v' containing the propositions given by provided inputs (line 20-21). After V_0 is generated, the layer E_1 and W_1 are updated accordingly (line 22-25).

Algorithm 3 $GraphConversion(G, s_0, g)$

Data: $G = \langle P_0, A_1, P_1, ..., A_n, P_n \rangle$ is a partially labelled planning graph, s_0 is the initial state; g is the goal;

1: $ST = FindCmb(P_n, g)$;
2: $V_n = \{(g, T) | T \in ST\}$;
3: $i = n$;
4: **repeat**
5: $V_{i-1} = E_i = W_i = \emptyset$;
6: **for** each $v \in V_i$ **do**
7: $w = \{(a, L_a, C_a) | \exists (a, L_a, C_a) \in T_v : L_a = i\}$;
8: **if** $w = \emptyset$ **then**
9: v' is a copy of v;
10: $V_{i-1} = V_{i-1} \cup \{v'\}$;
11: $LayerUpdate(E_i, W_i, w, v, \{v'\})$;
12: **else**
13: $V' = VGeneration(P_{i-1}, w, v)$;
14: $V_{i-1} = V_{i-1} \cup V'$;
15: $LayerUpdate(E_i, W_i, w, v, V')$;
16: **end if**
17: **end for**
18: $i = i - 1$;
19: **until** $i = 1$
20: $v' = (s_0, \{(I, 0, 0)\})$;
21: $V_0 = \{v'\}$;
22: **for** each $v \in V_1$ **do**
23: $w = \{(a, L_a, C_a) | \exists (a, L_a, C_a) \in T_v : L_a = 1\}$;
24: $LayerUpdate(E_1, W_1, w, v, V_0)$;
25: **end for**
26: **return** $\langle V_0, E_1, W_1, V_1, \ldots, E_n, W_n, V_n \rangle$;

E. Plan generation

Algorithm 4 describes how to find a best plan from the LWG. During forward search, we use the Dijkstra's algorithm to calculate the *cost-to-come* value for each vertex v. Since a vertex v contains a set of propositions, the *cost-to-come* value for v is the tuple of the minimum response time C_p^v for obtaining each proposition p in v. $r(v)$ is a function to get the *cost-to-come* value for each vertex v, *i.e.,* $r(v) = (C_{p_1}^v, \ldots, C_{p_j}^v)$ where $p_i \in P_v$ $(1 \leq i \leq ||P_v||)$ and $||P_v||$ is the number of propositions in P_v. $r(v)$ is calculated according to the parents of v and the weights. S is a set of actions included in the weights mapping to the edges linked to v. S is obtained from v (line 7). If S is empty, it means there are no actions in v that are able to be enabled at layer i. In such case, v only has one parent vertex v' who is a copy of v. The *cost-to-come* value of $r(v)$ is equal to that of

Algorithm 4 $PlanGeneration(\langle V_0, E_1, W_1, \ldots, W_n, V_n\rangle)$

1: **for** each $v \in V_0$ **do**
2: $r(v) = (0, \ldots, 0)$;
3: **end for**
4: $i = 1$;
5: **repeat**
6: **for** each $v \in V_i$ **do**
7: $S = \{(a, L_a, C_a) | \exists a \in T_v : L_a = i\}$;
8: **if** $||S|| = 0$ **then**
9: v' is the only parent of v;
10: $r(v) = r(v')$;
11: v' is recorded as the best parent of v;
12: **else**
13: $V' = \{v' | v' \in V_{i-1}$ is the parent of $v\}$;
14: $q = \sum_{a \in A} C_a$;
15: **for** each $v' \in V'$ **do**
16: $v'' = v$;
17: $r(v'') = CalQoS(v', v'', S)$;
18: **if** $\max\{C_p^{v''} | p \in P_{v''}\} < q$ **then**
19: $q = \max\{C_p^{v''} | p \in P_{v''}\}$;
20: $r(v) = r(v'')$;
21: v' is recorded as the best parent of v;
22: **end if**
23: **end for**
24: **end if**
25: **end for**
26: $i = i + 1$;
27: **until** $i = n$;
28: $v = \min^{-1}_{v \in V_n}\{\max_{C_p^v \in r(v)}\{C_p^v | p \in P_v\}\}$;
29: **for** $i = n, \ldots, 1$ **do**
30: v' in V_{i-1} is the best parent of v;
31: $\pi_i = \{a | \exists (v', v) \in E_i : (a, L_a, C_a) \in w(v', v)\}$;
32: $v = v'$;
33: **end for**
34: **return** π;

$r(v')$ (line 8-11). If v' has several parent vertices, we choose the vertex which can reach v with the minium response time (line 13-23). Function $CalQoS(v', v'', S)$ calculates the cost of each propositions in v'' to get the cost of reaching v'' from v' through actions in S. If p in v'' is also contained in the parent v', then $C_p^{v''} = C_p^{v'}$. Otherwise, we get a set of services S' from S for proposition p. Each service in S' can produce p. Then, $C_p^{v''}$ is calculated according to Equation 10. Starting from the selected vertex in V_n, the extraction of an optimal plan consist in retrieving the path that can reach this vertex with minimum response time (line 29-33).

$$C_p^{v''} = \min_{a \in S'}(\max_{p \in pre(a)} C_p^{v'} + C_a) \tag{10}$$

Example 2: The PLPG in Figure 1 is converted into the LWG as shown in Figure 2. For example, according to Algorithm 3, we generate the parent vertices for vertex $v = (\{G, D\}, \{(w_3, 3, 50)(w_4, 2, 10)\})$ at layer V_3. Since only w_3 in v is enabled at layer V_3, $\{(w_3, 3, 50)\}$ is the label mapping to the edges connected to v. w_3 produces D and has the input C. Hence, $\{G, C\}$ is the proposition set contained in the parent vertices of v. For the next step, we use function $VGeneration$ to find the suitable labels for $\{G, C\}$ in the parent vertices. We obtain two possible labels for $\{G, C\}$: $\{(w_1, 1, 120)(w_4, 2, 10)\}$, $\{(w_5, 2, 70)(w_4, 2, 10)\}$. However, $(w_1, 1, 120)$ can not be included in any label. w_1 produces C that is the input parameter of w_3. If w_1 is enabled at layer 1 rather than layer 2, it means w_3 in v can only be enabled at layer 2 rather layer 3, which conflicts with $(w_3, 3, 50)$ included in v. Therefore, only $\{(w_5, 2, 70)(w_4, 2, 10)\}$ is the correct label for $\{G, C\}$.

For the planning generation, the dotted arrow in Figure 2 shows the trace of backtracking. We use Algorithm 4 to calculate the value of each vertex in Figure 2. For example, the value of vertex $v = (\{G, D\}, \{(w_3, 2, 50)(w_4, 2, 10)\})$ at V_2 is $(40, 170)$. It means $C_G^v = 40$ and $C_D^v = 170$. v has two parents $v' = (\{C, E\}, \{(w_1, 1, 120)\})$ and $v'' = (\{C, E\}, \{(w_1, 1, 120)(w_2, 1, 30)\})$. According to parent v'', we have $C_G^v = \min_{w_4}(\max(C_E^{v'}) + C_{w_4}) = 120 + 10 = 130$ and $C_D^v = \min_{w_3}(\max(C_C^{v'}) + C_{w_3}) = 120 + 50 = 170$. According to v'', we get $C_G^v = 40$ and $C_D^v = 170$. Therefore, v'' is the best parent for v and $r(v) = (40, 170)$. Similarly, we calculate the values for other vertices and record the best parents accordingly. The best solution $\{w_2; w_4, w_5; w_3\}$ is obtained by retrieving the path through the dotted arrow in Figure 2. The response time of the best solution is 150 milliseconds.

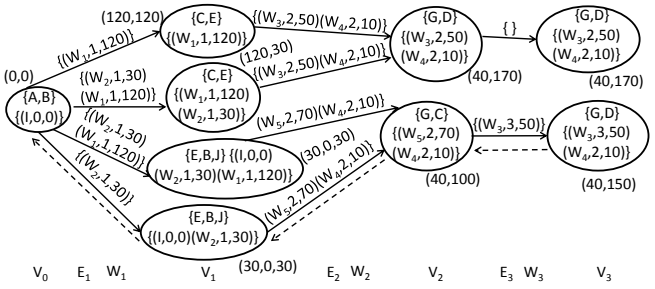

Figure 2. The Layered Weighted Graph (LWG) for Example 2.

F. Other single QoS criteria

If throughput is considered as a single QoS criteria, we use Equation 3 and Equation 4 to calculate *cost-to-come* value for vertice. We also need to change the calculation in the corresponding places in Algorithm 4. Line 2 sets the *cost-to-come* value for a proposition in V_0 as the maximum throughtput among all actions, *i.e.,* $\max_{a \in A} C_a$. To calculate the *cost-to-come* of proposition p, we replace Equation 10 by $C_p^{v''} = min_{a \in S'}(min_{p' \in pre(a)}(C_{p'}^{v''}, C_a))$.

Table II
CHANGES MADE IN ALGORITHM 4 FOR DIFFERENT SINGLE CRITERIA (r,r' ARE REAL VALUES, $k, k' \geq 0$ ARE INTEGERS)

QoS criteria	format of $r(v)$	line 2 and line 14	line 17	line 18 and line 19	line 28
exe. price	r	$r(v) = 0$ $q = \sum_{a \in A} C_a$	$r(v'') = r(v') + \sum_{a \in S} C_a$	**if** $r(v'') < q$ **then** $q = r(v'')$;	$\min^{-1}_{v \in V_n}\{r(v)\}$
reputation	(r, k)	$r(v) = (0, 0)$ $q = 0$	$r(v'') =$ $(r' + \sum_{a \in S} C_a, k' + \|\|S\|\|)$	**if** $\frac{r' + \sum_{a \in S} C_a}{k' + \|\|S\|\|} > q$ **then** $q = \frac{r' + \sum_{a \in S} C_a}{k' + \|\|S\|\|}$;	$\max^{-1}_{v \in V_n}\{\frac{r}{k}\}$
succ. exe. rate availability	r	$r(v) = 1$ $q = 0$	$r(v'') = r(v') \times \Pi_{a \in S} C_a$	**if** $r(v'') > q$ **then** $q = r(v'')$;	$\max^{-1}_{v \in V_n}\{r(v)\}$

Line 14 should be $q = 0$. If $\min\{C_p^{v''} | p \in P_{v''}\} > q$ (line 18), line 19 should change into $q = \min\{C_p^{v''} | p \in P_{v''}\}$. When the vetex v with the best QoS value is selected, $v = max^{-1}_{v \in V_n}\{min_{C_p^v \in r(v)}\{C_p^v | p \in P_v\}\}$ should be used in line 28.

For other criteria, Table II lists the changes have to be made accordingly in Algorithm 4.

IV. EXPERIMENTAL RESULT

As the WSC09 data sets are posted at [1], we use WSC09 data sets in our experiment. WSC09 has five data sets denoted as D1-D5 in Table III. Dataset 1 has 500 services and 5,000 concepts. Dataset 5 has 15,000 services and 100,000 concepts. The other datasets have 4,000-8,000 services and 40,000-60,000 concepts.

We compare our method with the method presented in our previous work [21] in terms of the quality of the solutions. In Table III, we show whether the correct QoS values can be calculated (the checkmarks). We can see that our method can find the correct QoS values for all the five datasets. However, [21]'s method can not find the solution with the best QoS value for the single criterion of execution price, reputation, successfull rate, or availability. It is because [21] uses heurisic search while searching the best solution for these criteria. Heurisic method may find the local optimal rather than the global optimal solution.

Table III
RESULTS WITH THE WSC09 DATA SETS: OUR METHOD/PAPER [21]

	D1	D2	D3	D4	D5
Resp. Time	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓
Throughput	✓/✓	✓/✓	✓/✓	✓/✓	✓/✓
Exec. price	✓/✓	✓/–	✓/–	✓/–	✓/–
Reputation	✓/–	✓/–	✓/–	✓/–	✓/–
Succ. rate	✓/✓	✓/–	✓/–	✓/–	✓/–
Availability	✓/✓	✓/–	✓/–	✓/–	✓/–

V. RELATED WORK

We study a kind of service composition problem that is to connect SOAP services into a network by matching their parameter to achieve some business goals. This is the most useful and practical service composition problem, as SOAP services are commonly used.

This kind of composition problem is also the problem presented by Web Service Challenge 2009 (WSC09) [1]. WSC09 data set has only response time and throughput. However, different QoS criteria has different calculation. The algorithms proposed to solve composition problems regarding the QoS criterion of response time or throughput may be not suitable for problems regarding other QoS criterion, *e.g.,* execution price or reputation. Our proposed methods is suitable for all types of QoS criteria.

[6], the first place winner of WSC09, uses the Dijkstra's principle to calculate the optimal value while searching all the paths. It uses a table to record all the enabled services. All the enable services and parameters have a current best quality value. However, it is very hard for [6] to calculate the quality value of other QoS criteria, *e.g.,* execution price. Since two paths leading to two parameter may share some common services on the two paths. In such case, not only the best quality values but also the services on the corresponding path for the best value need to be recorded. Enumerating all the possible paths may result in combinatorial explosion and make it become a NP-complete problem. Different from [6], [20] and [21] record the current best quality value in a tag associated to each action and each proposition in the planning graph. When thoughput or response time is a single QoS criteria, [20] and [21] use Dijkstra's algorithms to search the graph and find a best solution. When other QoS criteria rather than throughput and response time are considered, the calculation of the tag for each vertex may result in combinatorial explosion. Therefore, when the single QoS criteria is the execution price, reputation, or availability, [21] uses a heuristic search to find a best solution from the graph. However, heuristic methods will only get a local-optimal result close to the global-optimal result. Our work to convert the planning graph into the layered weighted graph is to aviod combinatorial explosion and get the global-optimal plan.

The second place paper [22] uses a simple breadth first search. This method could get only sequential solution. Therefore, they are not able to get the optimal QoS value correctly, if some services can be concurrently executed. A later paper [7] from the authors of [6] considers a subgraph of the service connection graph could be a solution. This is

an incorrect idea because firstly the service connection graph contains unreachable states from the initial state, which do not need to be constructed; secondly it removes the possibility to reuse actions in a plan. Heuristic search is also a commonly used approach for large problem space [8].

Another QoS related problem is the service selection problem [23], [24]. This kind of problem is to select a set of services to optimize the QoS for a predefinded business template which are composed of subtasks fulfilled by a set of services with varied QoS. [24] models it as a combinatorial optimization problem solved by integer programming. Since combinatorial optimization is a NP-complete problem, heuristic search can be applied to search only partially the problem space [23].

VI. Conclusion

Taking advantage of graph reachability, we use Graphplan technology combined with the Dijkstra's algorithm to solve QoS-aware service composition problem. The solution generated by our approach can satisfy both the functional requirements and the requirement of QoS optimization. The contribution is our approach can find the global optimal solution for all kinds of QoS criteria. One advantage of our approach is it reduces the possibilities of combinatorial explosion to a large degree when exploring the graph for a best plan. The other advantage is our approach can be easily extended by using multi-objective shortest path algorithms to solve QoS optimization on multiple QoS criteria for service composition problem. In the future, we will study the extension of our work for multiple QoS criteria.

References

[1] S. Bleul. Web service challenge rules. http://ws-challenge.georgetown.edu/wsc09/downloads/WSC2009Rules-1.1.pdf, 2009.

[2] A. L. Blum and M. L. Furst. Fast Planning through Planning Graph Analysis. *Artificial Intelligence Journal*, 90(1-2):225–279, 1997.

[3] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.

[4] J. E. Haddad, M. Manouvrier, and M. Rukoz. Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE T. Services Computing*, 3(1):73–85, 2010.

[5] J. Hoffman and B. Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[6] Z. Huang, W. Jiang, S. Hu, and Z. Liu. Effective pruning algorithm for qos-aware service composition. In *Proc. CEC*, pages 519–522, 2009.

[7] W. Jiang, C. Zhang, Z. Huang, M. Chen, S. Hu, and Z. Liu. Qsynth: A tool for qos-aware automatic service composition. In *Proc. ICWS*, pages 42–49, 2010.

[8] H. Kil and W. Nam. Anytime algorithm for qos web service composition. In *Proc. WWW (Companion Volume)*, pages 71–72, 2011.

[9] S. M. LaValle. *Planning Algorithms*. Cambridge, 2006.

[10] OASIS. Uddi version 2.04 api specification. http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, 2007.

[11] OASIS. Web services business process execution language (ws-bpel). http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, 2007.

[12] J. Peer. Web Service Composition as AI Planning – a Survey. Technical report, University of St.Gallen, 2005.

[13] P. Poizat and Y. Yan. Adaptive composition of conversational services through graph planning encoding. In *ISoLA (2)*, pages 35–50, 2010.

[14] J. Rao and X. Su. A survey of automated web service composition methods. In *Proc. of 1st Int. WS on Semantic Web Services and Web Process Composition, SWSWPC*, 2004.

[15] W3C. Owl web ontology language overview. http://www.w3.org/TR/owl-features/. Retrieved 2011-06-30, 2004.

[16] W3C. Semantic annotations for wsdl and xml schema (sawsdl). http://www.w3.org/TR/sawsdl/. Retrieved 2011-06-30, 2007.

[17] W3C. Soap version 1.2 part 1: Messaging framework (second edition). http://www.w3.org/TR/soap12-part1/#intro. Retrieved 2011-06-30, 2007.

[18] W3C. Web services description language (wsdl) version 2.0. http://www.w3.org/TR/wsdl20/. Retrieved 2011-06-30, 2007.

[19] Wikipedia. Dijkstra's algorithm. http://en.wikipedia.org/wiki/Dijkstra_algorithm. Retrieved 2011-06-30.

[20] Y. Yan and M. Chen. Anytime qos-aware service composition over the plangraph. In *Proc. of the 27th Annual ACM Symposium on Applied Computing*, pages 1968–1975, 2012.

[21] Y. Yan and M. Chen. Anytime qos-aware service composition over the plangraph. *Service Oriented Computing and Applications*, pages 239–290, 2013.

[22] Y. Yan, B. Xu, Z. Gu, and S. Luo. A qos-driven approach for semantic service composition. In *Proc. CEC*, pages 523–526, 2009.

[23] T. Yu, Y. Z. 0001, and K.-J. Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *TWEB*, 1(1), 2007.

[24] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Trans. Software Eng.*, 30(5):311–327, 2004.

[25] X. Zheng and Y. Yan. An Efficient Web Service Composition Algorithm Based on Planning Graph. In *Proc. of ICWS08*, pages 691–699, 2008.