Noname manuscript No.
(will be inserted by the editor)

Anytime QoS-Aware Service Composition over the PlanGraph

Yuhong Yan · Min Chen

Received: date / Accepted: date

Abstract Automatic service composition is the generation of a business process to fulfill business goals that cannot be fulfilled by individual services. Planning algorithms are frequently used to solve this problem. In addition to satisfying functional goals, recent research is geared towards selecting the best services to optimize the QoS of the result business process. Without considering QoS, the planning algorithm normally searches for the shortest plan, which actually implies the unit execution time for each service. With QoS, a longer plan may have better QoS values, and thus is preferred over a shorter one. In this paper, we are motivated to combine a systematic search algorithm like Dijkstra's algorithm with a planning algorithm, GraphPlan, to achieve both functional goals and QoS optimization at the same time. The planning graph generated by GraphPlan is a compact representation of all execution paths, which makes it feasible to apply Dijkstra's principle. In our new QoSGraphPlan algorithm, we extend Dijkstra's algorithm from working on a single source graph to working on the planning graph whose nodes have multiple sources. Using our method, we can get the best QoS value for throughput and response time in polynomial time when they are the single criteria. For the other QoS criteria, such as execution time, reputation, successful execution rate, and availability, our algorithm is exponential for both single criterion problem and multiple criteria problem. In this case, we extend QoSGraphPlan with beam search to solve the combination explosion problem. As our algorithms search for an optimal solution during the process of constructing the planning graph, they belong to the category of anytime algorithms that return better solutions if they keep running for a longer time.

Keywords QoS optimization · Graph Plan · Web service composition

1 Introduction

Web services are self-described software entities which are posted across the Internet using a set of open standards such as SOAP [29], WSDL [30], and UDDI [19]. With these open standards, Web services are automatically invokable and interoperable. This leads to the important feature of composability, meaning that it is possible to automatically generate a business process to fulfill business goals that cannot be fulfilled by individual services.

Automated Service Composition (ASC) is studied under different assumptions [21,25]. 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 frequently used to solve this problem [23,24,40]. AI planning algorithms search the problem space to find a path from the initial state to a goal state. Normally the planning algorithms stop at the first found feasible solution, which corresponds to the shortest plan. This actually implies the execution

This work is supported by project "Building Self-Manageable Web Service Process" of Canada NSERC Discovery Grant, RGPIN/298362-2007

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

Min Chen
Dept. of Computer Science and Software Engineering, Concordia University, Montreal, Canada

time for each service is a unit. Thus, the shortest path has the shortest execution time, hence the shortest response time.

In addition to satisfying business goals, recent research moves to selecting the best services to optimize the QoS measures, such as throughput and response time, of the target business process. With QoS consideration, a shortest plan may not be preferred, because a longer plan may have better QoS values. For example, a plan with three consecutive services may be faster than a plan with two consecutive services, when the response time of the services varies. Therefore, we need to modify the classic planning algorithm to become QoS aware so as to find a QoS optimized solution. In this paper, we tackle the problem to satisfy the functional requirements, *i.e.,*produce functional goals, and optimize the QoS at the same time.

People have used optimization algorithms such as Dynamic Programming (DP) [14,15] and Integer Programming (IP) [9] to find a QoS optimized solution. We use planning algorithm topped by optimization algorithm in this paper. We consider combining planning algorithm and optimization algorithm can use knowledge in both topic and reveal more facts about this problem. A common problem in using DP is that the optimal paths to reach individual functional goals are combined to be the optimal solution. This kind of optimal solutions can contain multiple redundant services that removing them does not worsen the QoS value. This problem is ignored in the literatures until our paper [8]. In this paper, we discuss the cause of the redundant services and how to remove them. We also find that some of the models, *e.g.,* [15] does not consider the possibility of reusing the same service in a plan, which can be avoided if using a planning algorithm. More comparison of our method with related work can be found in Section 5.

The main contribution of our work is the combination of Dijkstra graph traversing with planning graph algorithm to achieve both functional service composition and QoS optimization simultaneously. We apply the principle of Dijkstra's algorithm to systematically search the optimal QoS path on a planning graph. A planning graph generated by the GraphPlan algorithm is a compact representation of all execution paths, which makes it feasible to apply Dijkstra's principle. We extend Dijkstra's algorithm from working on a single source graph to working on the planning graph whose nodes have multiple sources. As our algorithms work during the construction of the planning graph, we make them return only the best solutions during the search. This fact puts our algorithms into the category of anytime algorithms [32]. We develop a group of algorithms in the same spirit to deal with either single criterion or multiple QoS criteria. Through our study, we find that response time and throughput can be optimized in polynomial time, while the other criteria optimization problems are NP complete problem. Our algorithms are able to perform the following tasks:

1. For throughput and response time, QoSGraphPlan can find a solution with the globally optimized QoS value and possibly redundant services in polynomial time. It takes polynomial time to remove the redundant services.
2. For the single criterion problem with execution price, reputation, successful execution rate and availability, it is an NP-complete problem. QoSGraphPlanExt can find a solution with globally optimized QoS value and no redundant services in exponential time complexity. Our BeamQoSGraphPlan is a heuristic search algorithm which uses beam search to find an optimal solution.
3. For multiple criteria problems, we use a non-preemptive model [9] to aggregate the QoS values into a utility value in order to compare the execution paths. Our BeamQoSGraphPlan can also be used for multiple criteria problem.

The paper is organized as follows. Section 2 gives background knowledge of QoS criteria, Graph Plan, and Dijkstra's algorithm. Section 3 presents our anytime QoS-aware service composition algorithm. We also discuss the properties of our algorithm in this section. Section 4 presents the results of the experiments with artificial data sets. Related work is reviewed in Section 5. We end up with a conclusion in Section 6.

2 Preliminaries

2.1 The QoS-Aware Service Composition Problem

We take the services as being stateless black boxes (no conversations) in this paper. The services expose themselves in WSDL descriptions which do not include state information[1]. We can associate semantic information to inputs and outputs using SAWSDL [28] or OWL [27].

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.

[1] For stateful services, we have developed a modelling technique to convert the sequential orders of each operation into its preconditions and postconditions [24]. Therefore, if we have QoS values for each operation in a stateful service, the methods developed in this paper can be applied to both stateful and stateless services.

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 $w : o_1, \ldots, w : o_n$.

An output of one service could become the input of another service if they are exactly the same concept, or the input concept subsumes the output concept. For example, assume one service needs "automobile" as one of its inputs; if another service outputs "automobile" or "car", the two services can be connected via their compatible parameters. In Section 4, we show the semantic relations can be "flattened". Therefore, in discussing the principle, we ignore the semantic relations, but consider that the parameters can be matched by their names.

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, $\sigma = w_1 || w_2 || \ldots || w_n$.

Some commonly used quality criteria for a Web service w and their aggregated values over σ are listed as below [11,2]:

– **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.

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

– **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 are negative, *i.e.*, the higher the value, the lower the quality. Response time and execution price are in this category. The other criteria are positive, *i.e.,*the higher the value, the higher the quality. We want to have a uniform way to compare the qualities, especially with the multiple criteria. We apply a Multiple Criteria Decision Making (MCDM) technique [26] to aggregate QoS value $Q(w)$. First, we scale the value of a quality i for a service w_j. For negative criteria (*i.e.,*response time and execution price), values are scaled according to Equation 9. For positive and non multiplication criteria (*i.e.,*throughput and reputation), values are scaled according to Equation 10. For positive and multiplication criteria (*i.e.,*successful execution rate and availability), values are scaled according to Equation 11. The logarithm is used for multiplication criteria so that the aggregated utility value for a network can be monotonic to the aggregated QoS value. For all the criteria, the higher the quality value, the lower the utility value $U_i(w_j)$. Please notice that other papers, like [38] and [5], do a similar conversion, but their aggregate functions have the opposite monotonic direction (the higher the quality value, the higher the utility value). This is because the classic Dijkstra's algorithm finds the "shortest distance" (lowest cost) over a graph. Therefore, we make our algorithms in order to lower the utility value to keep the same sense.

$$U_i(w_j) = \begin{cases} \frac{Q_i(w_j) - Q_i^{min}}{Q_i^{max} - Q_i^{min}} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{9}$$

$$U_i(w_j) = \begin{cases} \frac{Q_i^{max} - Q_i(w_j)}{Q_i^{max} - Q_i^{min}} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{10}$$

$$U_i(w_j) = \begin{cases} \frac{\ln(Q_i^{max}) - \ln(Q_i(w_j))}{\ln(Q_i^{max}) - \ln(Q_i^{min})} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{11}$$

The overall quality score for a Web service w_j is defined in Equation 12:

$$U(w_j) = \sum U_i(w_j) \times W_i \tag{12}$$

where $W_i \in [0, 1]$ and $\sum W_i = 1$. W_i represents the weight of criterion i.

For a network of services $\sigma = w_1, w_2, \ldots, w_n$, we want to get the aggregated utility value too. Equations 13 to 20 aggregate the single criterion values. We can easily prove that Equations 13 to 20 sort the networks in the order as Equations 3 to 8 do. That means if σ_1 is better than σ_2 according to one QoS criterion, σ_1 should be better than σ_2 according to its correspondent utility criterion.

$$U_1(w_1; \ldots; w_n) = \sum U_1(w_i) \tag{13}$$
$$U_1(w_1|| \ldots ||w_n) = \max U_1(w_i) \tag{14}$$
$$U_2(w_1; \ldots; w_n) = \max U_2(w_i) \tag{15}$$
$$U_2(w_1|| \ldots ||w_n) = \max U_2(w_i) \tag{16}$$
$$U_3(w_1, \ldots, w_n) = \sum U_3(w_i) \tag{17}$$
$$U_4(w_1, \ldots, w_n) = \frac{1}{n} \sum U_4(w_i) \tag{18}$$
$$U_5(w_1, \ldots, w_n) = \sum U_5(w_i) \tag{19}$$
$$U_6(w_1, \ldots, w_n) = \sum U_6(w_i) \tag{20}$$

If we want to compare a network of services $\sigma = w_1, \ldots, w_n$ under multiple criteria, we need to get the single criteria utility values first, then aggregate them into one general utility score. The aggregated utility value for σ is as Equation 21. We need to apply a normalization before we can aggregate the utility values. For throughput and reputation, the aggregated utility for single criterion is between [0,1]. For the other criteria, the aggregated utility for single criterion is between [0,n]. Thus we need to divide these values by n [15].

$$U(\sigma) = \sum_{i=1,3,5,6} \frac{1}{n} U_i(\sigma) \times W_i + \sum_{i=2,4} U_i(\sigma) \times W_i \tag{21}$$

where $W_i \in [0,1]$ and $\sum W_i = 1$.

We study QoS-aware service composition under both single criterion and multiple criteria. For single criterion problems, we can use either single QoS values or single utility values to compare the plans. For multiple criteria problems, we should first calculate the single utility values and then aggregate them into one general utility value using Equation 21.

If the QoS value is represented as a range, *e.g.*, [90-100], we need to define some arithmetic operators over range as following.

$$[x_1, y_1] + [x_2, y_2] = [x_1 + x_2, y_1 + y_2] \tag{22}$$
$$\max([x_1, y_1], [x_2, y_2]) = \max(y_1, y_2) \tag{23}$$
$$\min([x_1, y_1], [x_1, y_2]) = \min(x_1, x_2) \tag{24}$$
$$[x_1, y_1] * [x_2, y_2] = [x_1 * x_2, y_1 * y_2] \tag{25}$$

Based on these operators, we can refine functions 1 to 21 for range values. For example, assuming w_i has a range $[x_i, y_i]$ for a quality criterion, functions 1 to 9 are changed into follows (other functions can be changed in the same way):

$$Q_1(w_1; \ldots; w_n) = [\sum x_i, \sum y_i] \tag{26}$$
$$Q_1(w_1|| \ldots ||w_n) = \max y_i \tag{27}$$
$$Q_2(w_1; \ldots; w_n) = \min x_i \tag{28}$$
$$Q_2(w_1|| \ldots ||w_n) = \min x_i \tag{29}$$
$$Q_3(w_1, \ldots, w_n) = [\sum x_i, \sum y_i] \tag{30}$$
$$Q_4(w_1, \ldots, w_n) = \frac{1}{n}[\sum x_i, \sum y_i] \tag{31}$$
$$Q_5(w_1, \ldots, w_n) = [\prod x_i, \prod y_i] \tag{32}$$
$$Q_6(w_1, \ldots, w_n) = [\prod x_i, \prod y_i] \tag{33}$$
$$U_i(w_j) = \frac{[x_j - \min x_j, y_j - \min x_j]}{\max y_j - \min x_j} \tag{34}$$

In the rest of paper, our calculation uses single QoS values, not range. However, our methods can be extended to range values following the discussion above.

2.2 The Planning Technique and Graph Plan

AI planning [17] has been applied with success to service composition [22, 7].

Definition 3 Given a finite set $L = \{p_1, \ldots, p_n\}$ of proposition symbols, a *planning problem* [17] is a triple $P = ((S, A, \gamma),\ s_0,\ g)$, where:

- $S \subseteq 2^L$ is a set of states.
- A is a set of actions, an *action* a being a couple $(pre, effects^+)$ where $pre(a) \subseteq L$ and $effects^+(a) \subseteq L$ denote respectively the preconditions and the (positive) effects of a.
- γ is a state transition function such that, for any state s where $pre(a) \subseteq s$, $\gamma(s, a) = s \cup effects^+(a)$.
- $s_0 \in S$ and $g \subseteq L$ are respectively the initial state and the *goal*.

In Definition 3, $pre(a)$ is the set of the propositions as the precondition of action a. The definition in [17] takes into account predicates and constant symbols which are then used to define states (ground atoms made with predicates and constants). We directly use propositions here, because in the Web service models, we do not have predicates. In [17], an action also includes negative effects. Since in Web service models, we have only positive effects, we remove the negative effects definition for clarity.

Different algorithms have been proposed to solve planning problems and get plans from them, *e.g.*, depending on whether they are building the underlying graph structure in a forward (from initial state) or backward (from goal) way [17]. The study in this paper is based on an AI planning algorithm called Graph Plan [3]. Recent works have demonstrated the suitability of this model for ASC [24,40]. Graph Plan is particularly interesting for our idea of applying Dijkstra's algorithm to it, because the planning graph used is a compact representation of all the possible execution paths. This makes **it possible to do a systematic search on the planning graph**.

A planning graph G is a directed acyclic leveled graph (see Fig. 1). The levels alternate proposition layers P_i and action layers A_i. The initial proposition layer P_0 contains the initial propositions (s_0). An action a is put in layer A_i iff $pre(a) \subseteq P_{i-1}$ and then $effects^+(a) \subseteq P_i$. The multiple actions added into one layer are independent in the sense that they could possibly be executed parallelly. The planning graph actually explores multiple search paths at the same time when expanding the graph. The construction of the planning graph stops at a layer A_k iff the goal is reached ($g \subseteq A_k$) or in case of a fixpoint ($A_k = A_{k-1}$). In the former case there exists at least a solution, while in the latter there is not. Solution(s) can be obtained using backward search from the goal. In Graph Plan, the solution is layered as defined in Definition 4.

Definition 4 A *plan* is a sequence of sets of actions $\pi_1;\pi_2;\ldots;\pi_n$, in which each π_i ($i = 1,\ldots,n$) is a set of parallel actions (denoted with $||$). π_1 is applicable to s_0. π_i is applicable to γ (s_{i-2},π_{i-1}) when $i = 2,\ldots,n$. $g \subseteq \gamma(\ldots(\gamma(\gamma(s_0,\pi_1),\pi_2)\ldots\pi_n)$.

We can understand that a plan transfers the system state from its initial state s_0 to an end state s_n which contains the required goal g. The effects of the actions in an action layer provide the preconditions of the actions in the next action layer. The actions in one layer π_i are parallel to each other, *i.e.,*the effects of an action should not be the precondition of another action. Finally, there is no loop in a plan.

The Graph Plan 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 [3]. If the goal is reached, this means the problem has a solution. Then 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 [13]. We want to get a minimal set of services that can solve the problem.

Following [40], 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 [20]. Additionally, planning graphs enable us to retrieve plans with parallel invocations. These can be encoded using parallel operations (WS-BPEL flow).

Example 1 A set of available services with their input/output parameters and response time in milliseconds are listed in Table 1 (modified from [15]). The composition query is $(D_{in}, D_{out}) = (\{A, B, C\}, \{D\})$. We use the Graph Plan approach to solve this service composition problem. According to D_{in}, D_{out} and the available services in Table 1, we construct a planning graph as shown in Figure 1.

In Figure 1, the no-op actions are represented by dashed arrows. A no-op action simply inherits a true proposition from a previous proposition layer. It has no cost. A no-op action is preferred over a non no-op action during the plan extraction phase. At an action layer, all the enabled actions can be added, including those possibly added in the previous layers (the shaded actions in Fig 1. An action a at layer A_i takes the incoming arcs originating from its inputs at P_{i-1} and connects to its outputs at P_i. For example, w_1 at A_1 takes three arcs originating from its inputs A, B and C at P_0 and connects to its output J at P_1. To make the figure readable, we do not draw all the no-op arcs on A_2, neither do we draw the arcs connecting the shaded actions in the action layers after. Please notice that the graph reaches the fix point at layer P_4. After the planning graph reaches the fixed-point layer, we extract three solutions starting from goal D at P_4: $\{w_1; w_6\}$, $\{w_2; w_3; w_7\}$ and $\{w_2; w_4; w_8; w_7\}$.

Table 1 A set of available services

w_i	inputs	outputs	response time	utility value
w_1	A, B, C	J	800	1
w_2	B, C	E, F	100	0
w_3	C, E	H	600	0.71
w_4	C, F	G	100	0
w_5	K	H	600	0.71
w_6	J	D	100	0
w_7	H	D	300	0.29
w_8	G	H	100	0

Graph Plan can generate a plan with sequential and parallel actions. Graph Plan can generate plans with repeated sequential actions. However, Graph Plan cannot

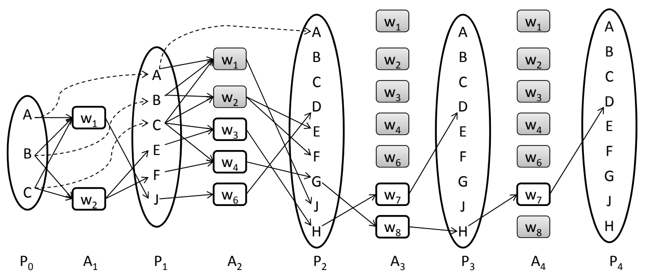

Fig. 1 The planning graph for Example 1.

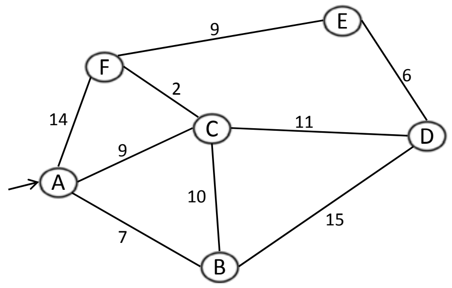

Fig. 2 The graph for Example 2.

generate a plan with loop or conditional construct. Conditional planning (to include conditional construct and loop construct in a plan) is a special topic to study in planning domain, which is not a problem we try to solve in this paper. Another work from us [24] proposed an encoding method to re-define actions and propositions in the exclusive choice construct into preconditions and effects of actions that can be used by planning graph. Using the encoding approach in [24], planning graph can also handle the exclusive choice pattern. In this paper, we concentrate on dealing with plans with no loop constructs and no condition constructs.

2.3 Dijkstra's Algorithm

Dijkstra's algorithm's goal is to find single-source shortest paths in a graph [16]. Dijkstra's algorithm is a systematic search algorithm. If the graph is finite, systematic search means that the algorithm will visit every reachable state, and keep track of states already visited to avoid infinite loops 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 a finite graph in this paper.

Suppose a graph $G = (V, E)$ has every edge $e \in E$ labeled with a distance $d(e)$. Assuming the edge e is from a vertex v, we can also write it in the state-space representation $d(v, e)$.

For each vertex v, we define a *cost-to-come* function $C : V \to [0, \infty]$. 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 $D(v)$.

Initially, $D^*(v_I) = 0$. Each time a vertex v' is visited, a distance is computed as $D(v') = D(v)^* + d(v, e)$, in which e is the edge from v to v'. Here, $D(v')$ represents the best cost-to-come known so far, but we do not write D^* because it is not yet known whether v' was reached optimally. In the search algorithm, we have a queue to record all the vertices to visit. If v' is visited again, which means a new path to v' is discovered, the cost-to-come value $D(v')$ may be updated if the new path has a lower value.

The complexity of Dijkstra's algorithm over a Graph $G = (V, E)$ is $O(|V|^2)$ without using a 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 [10].

We can only describe the principle of Djikstra's algorithm in this paper. The following is an example to explain how to find single-source shortest paths in a graph through Dijkstra's algorithm. More examples and the pseudo code can be found in [32].

Example 2 Figure 2 is a graph with the arcs labeled with their lengths. Node A is the source vertex. Table 2 presents the steps through Dijkstra's algorithm to find the shortest path from the origin A to any other destination vertex in the graph. In the table, $d(.)$ represents the distance of the node and $p(.)$ represents the parent of the node. Once we successfully find the shortest path from the origin A to a node, the node is added into "Solved nodes" in Table 2. For example, at "Step 5", "Solved nodes" is $ABCFD$ because nodes A, B, C, F and D have found their shortest paths.

3 QoS-Aware Planning Graph

3.1 Take the Advantages of Both Planning Graph and Dijkstra's Algorithm

The principle of Dijkstra's algorithm is to calculate the best cost-to-come value for a vertex. If we think of a proposition as a vertex of the planning graph, we could

Table 2 Dijkstra's algorithm

Step	Solved nodes	B		C		D		E		F	
		$d(B)$	$p(B)$	$d(C)$	$p(C)$	$d(D)$	$p(D)$	$d(E)$	$p(E)$	$d(F)$	$p(F)$
1	A	7	A	9	A	∞	-	∞	-	14	A
2	AB	-	-	9	A	22	B	∞	-	14	A
3	ABC	-	-	-	-	20	C	∞	-	11	C
4	$ABCF$	-	-	-	-	20	C	20	F	-	-
5	$ABCFD$	-	-	-	-	-	-	20	F	-	-
6	$ABCFDE$	-	-	-	-	-	-	-	-	-	-

use the same principle to calculate the best cost-to-come which is the best QoS value for each proposition. Then, we could get the overall cost-to-come for all the goal propositions. And during the search, we could record the best path which is the best plan.

The above idea is feasible, because the planning graph can be understood as a compact representation of all the execution alternatives. As all the applicable actions are considered on each layer, the planning graph is built to model the whole problem space until a solution is detected, rather than to solve a particular problem. Therefore, we could visit all the reachable system states over the planning graph. This makes Dijkstra's principle work over the planning graph.

Yet, we need to overcome several difficulties. First of all, Dijkstra's algorithm is for single source situations, *i.e.,*one edge represents one path between two vertices. While in a planning graph, a service takes multiple inputs which could possibly come from multiple services. Therefore, a cost value needs to be calculated from several edges, instead of one. Second, the planning graph presents both parallel and sequential connections between services, while a normal graph does not represent parallel connections. Third, different QoS criteria have different formulas for calculation. We need to find a way to calculate the aggregated QoS over the Planning graph. We present our solution in the following subsections.

3.2 Generation of Tagged Planning Graph

For simplicity, we present our algorithm using response time as the single quality criterion, *i.e.,*the $cost(a)$ in Algorithms 1-4 is either the QoS value (using Equations 1 and 2) or the utility value (using Equations 13 and 14) of response time for a service. And either way should get the same solution. The calculation of the other QoS criteria is discussed in Subsection 3.5.

We can consider a classic Planning Graph is a Directed Acyclic Graph (DAG) $G = (V, E)$. It has two kinds of vertices $V = V_A \cup V_P$, where V_A are the vertices representing actions and V_P are the vertices 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 $(V_A \times V_P)$ connect the actions with their output parameters.

We associate a real value to every vertex of a planning graph and obtain a Tagged Planning Graph (TPG):

Definition 5 A **Tagged Planning Graph** (TPG) is a Planning Graph $G = (V, E)$ with a cost function $cost(V)$ for vertices, $cost : V \mapsto \Re$, where $\Re$ is the real numbers.

The tags on the action vertices are the QoS values (or utility values) for executing the actions, *i.e.,cost*(a) is the cost of executing an action a. The tags of the proposition vertices are the QoS values (or utility values) for obtaining this proposition, *i.e.,cost*(p) is the cost of obtaining a proposition p.

Algorithm 1 called *QoSGraphPlan* is our main algorithm. It is modified from the standard GraphPlan algorithm [17] to calculate TGP and extract a solution. *QoSGraphPlan* repeats *ExpandGraph* (Algorithm 2) (line 2) until a fixed point (Definition 6) is detected (line 7). When all the goals g are satisfied and some of them are generated by non no-op actions (line 3), which means a new solution is found, the algorithm calls *ExtractPlan* (Algorithm 4) to extract the plan (line 4). *ExtractPlan* returns a solution only when it detects the found solution is better than the best solution found so far. This makes *QoSGraphPlan* an anytime algorithm. As time goes by, *QoSGraphPlan* can return better and better plans. When the fix point is reached, *QoSGraphPlan* terminates.

Algorithm 2 expands the TPG by one layer. Line 1 gets all the enabled actions for action layer i. The enabled actions are those whose inputs are in the previous proposition layer $i-1$. Each action has a tag t which is the cost value. The P_i layer contains the effects of A_i. We want to calculate the *cost-to-come* value for each $p \in P_i$. If an action a produces p, the cost of p is the maximum of the costs of all the preconditions of a plus $cost(a)$. If there are several actions to produce p, we choose the action which can produce the minimal *cost-to-come.* This is what $\min_a$ means and this action is recorded as the best parent of p. If there is more than one parent producing the best *cost-to-come* value, we

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

```
Data: $G = \langle P_0, A_1, P_1, ..., A_i, P_i\rangle$ is a planning graph; i=1;
 1: repeat
 2:    $G = ExpandGraph(G)$;
 3:    if $g \subseteq P_i$ and $g$ generated by non no-ops then
 4:       print $ExtractPlan(G, g)$;
 5:    end if
 6:    $i = i + 1$;
 7: until $Fixedpoint(G)$
 8: if $g \not\subseteq P_i$ then
 9:    print $\emptyset$;
10: end if
```

Algorithm 2 $ExpandGraph(\langle P_0, A_1, ..., A_i, P_i\rangle)$

```
1: $A_i = \{(a, t)|pre(a) \subseteq P_{i-1}, a \in A, t = cost(a)\}$;
2: $P_i = \{(p, t)|\exists a \in A_i : p \in effects(a), t = \min_a(\max(cost(pre(a))) + cost(a))$, record $a$ that generates $t$ as the best parent of $p\}$;
3: for each $a \in A_i$ do
4:    link $a$ with precondition arcs and negative effects arcs to $pre(a)$ and $effect^-(a)$ in $P_{i-1}$;
5:    link $a$ with each of its $effects(a)$ in $P_i$;
6: end for
7: return $\langle P_0, A_1, ..., A_i, P_i\rangle$;
```

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

```
1: if $P_i = P_{i-1}$ then
2:    return true;
3: else
4:    return false;
5: end if
```

Algorithm 4 $ExtractPlan(\langle P_0, A_1, ..., A_n, P_n\rangle, g)$

```
Data: $U$ is the QoS value for the current best plan
 1: $U' = \max(cost(l)), \forall l \in g$;
 2: if $U' > U$ then
 3:    return $\emptyset$
 4: else
 5:    $U = U'$;
 6: end if
 7: for $i = n, \ldots, 1$ do
 8:    Select an action set $\pi_i = \{a|a$ is the best parent of $l, \forall l \in g\}$;
 9:    $g = \{pre(a)|\forall a \in \pi_i\}$;
10: end for
11: return $\pi$
```

can choose either one, because both paths are equally the best. Lines 4 and 5 create the arcs between actions and propositions.

Algorithm 3 checks if a fixed point layer is reached.

Definition 6 A fixed point layer in a TPG is a layer k such that for $\forall i, i \geq k$, layer i is identical to layer k, $i.e., P_i = P_k$ and $A_i = A_k$.

$P_i = P_k$ means the vertex-tag pairs are identical between P_i and P_k. Formally, $\forall(p, t) \in P_i, (p, t) \in P_k$ and $\forall(p', t') \in P_k, (p', t') \in P_i$. $A_i = A_k$ has similar meaning. Theorem 2 in the following subsection shows we just need to check whether $P_i = P_{i-1}$ at a layer i.

Algorithm 4 first calculates the cost for the whole plan in U'. For response time, it is the maximal cost of the all the goals $U' = \max(cost(l))$ (line 2). It uses U to record the best cost value known so far. Only if the new cost U' is lower than U, the algorithm extracts the new plan (line 7-9), otherwise, it returns $\emptyset$ (line 3). Since when the TPG is built we record the best parent of a proposition, the extraction of a new plan consists in just retrieving the recorded best parent for each involved proposition.

Example 3 Figure 3 shows the TPG for the problem in Example 1. Each proposition node or action node is labeled by its tag, denoted as "QoS value/utility value", in the TPG. At proposition layers, propositions are separated by dashed lines. w_1 at layer A_1 has a tag 800/1 because the response time of w_1 is 800, w.r.t the utility value of 1. J is the output of w_1 and only w_1 at layer A_1 produces J. According to line 2 of Algorithm 2, $\max(cost(pre(w_1))) + cost(w_1) = 800$. Therefore, J at layer P_1 has the response time of 800, w.r.t the utility value of 1. Table 3 lists the value of "QoS value/utility value"for each proposition over P_0 to P_4. At the fixed-point layer P_4, the goal D has its best response time 600. Using either the QoS values or the utility values, the best solution is obtained through backtracking: $\{w_2; w_4; w_8; w_7\}$.

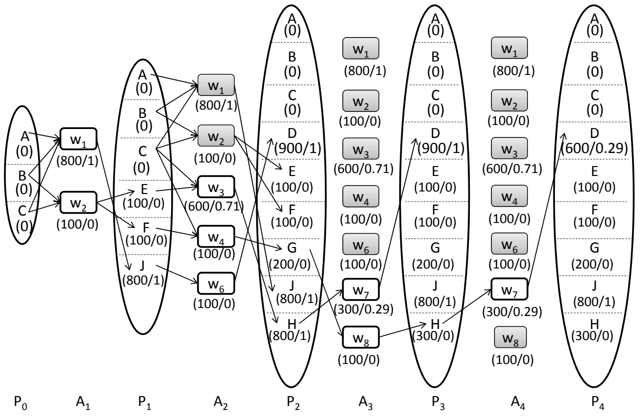

Fig. 3 The tagged planning graph for Example 1.

Use *QoSGraphPlan* for throughput as single criterion. If we use QoS value according to Equations 3 and 4, line 2 in Algorithm 2 should use $t = \min_a(\min(cost(pre(a)), cost(a)))$ to calculate the cost

Table 3 The QoS value (numerator) and the utility value (denominator) for a proposition at each proposition layer $p \in P_i$ $(i = 1, \ldots, 4)$ in the TPG for Example 1

Proposition	P_0	P_1	P_2	P_3	P_4
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	-	-	900/1	900/1	600/0.29
E	-	100/0	100/0	100/0	100/0
F	-	100/0	100/0	100/0	100/0
G	-	-	200/0	200/0	200/0
J	-	800/1	800/1	800/1	800/1
H	-	-	700/0.71	300/0	300/0

value. If we use utility value according to Equations 15 and 16, $t = \max_a(\max(cost(pre(a)), cost(a)))$ should be used. When we calculate the cost for the whole plan in line 1 of Algorithm 4, $U' = \min(cost(l)), \forall l \in g$ should be used, if using QoS value, or $U' = \max(cost(l)), \forall l \in g$ should be used, if using utility value.

The *QoSGraphPlan* algorithm works well for response time and throughput when they are considered as a single criterion. *QoSGraphPlan* has polynomial time complexity (proofs in the next section). *QoSGraphPlan* considers the best plan as the plan with the lowest cost. Practically, *QoSGraphPlan* searches for the best path to generate each proposition in the goal, and puts all the best paths together as a best solution. However, it is not necessary to use all the best paths to generate the goal propositions, because the QoS value for the whole plan is determined by the worst path. It is possible that we can relax the best paths to some paths that could share some services, which means lower execution cost with the same best response time or throughput. We discuss the redundancy problem in Subsection 3.4. Redundancy removal only make sense when multiple criteria are considered. For single criterion optimization, redundancy removal is not necessary.

3.3 The Properties of QoSGraphPlan

We present the properties of our QoSGraphPlan in this subsection.

Theorem 1 *The time to expand a TPG to layer k is polynomial to the size of the planning problem.*

Proof: for a planning problem (A, s_0, g) has a total of n propositions and m actions, then $\forall i : |P_i| \leq n$. This is because even though a proposition may be associated with different tags, a proposition can only appear once in P_i. Thus $|P_i| \leq n$. Further, $|A_i| \leq m + n$ which include possibly n no-op actions. Therefore, the size of a TPG with k layers is $|s_0| + (m + 2n)k$ $\square$.

Theorem 2 *Every TPG has a fixed point layer k, which is the smallest k such that $P_{k-1} = P_k$.*

Proof: A_{k+1} depends only on P_k. Thus $P_{k-1} = P_k$ implies $A_{k+1} = A_k$, and consequently $P_{k+1} = P_k$. Therefore, for $\forall i > k, A_i = A_k, P_i = P_k$. $\square$

Theorem 3 *QoSGraphPlan has polynomial time complexity.*

Proof: Theorem 1 shows the time to expand a TPG is polynomial to the size of the problem. Theorem 2 proves the expansion of a TPG stops at a fixed point. Now we only need to prove the solution extraction by Algorithm 4 is polynomial. The complexity to get a solution by Algorithm 4 lies in retrieving the parents of the goals on each layer, until reaching the initial layer. As the best parents are recorded during the construction phase, it takes $|g| \leq |D|$ operations to $|g|$ parents at each layer. It takes $n \leq |A|$ loops to do the retrieval on n layers. Therefore, *QoSGraphPlan* is polynomial. $\square$

QoSGraphPlan is also an anytime algorithm. When it has finished, the best response time value and a correspondent solution are produced. If the problem has no solution, our algorithm can report no solution, as the graph plan algorithm.

Understand the scheduling of services over TPG. When a service is associated with response time in TPG, a question that arises is *when can the services on the next action layers start?* Is it after finishing all the services in the previous action layer? One should understand that TPG constrains the input and output dependency among the services. If the inputs of a service are produced, this service can start. TPG does not show the time constraints among the services. Therefore, it is possible that a service starts before all the services in the previous action layer are finished. We use ; and $\|$ to represent the connections of the services in TPG. But this representation does not express the input and output dependency, nor the starting sequences in the sense of scheduling.

Example 4 Figure 4 shows a planning graph with six services and their input and output parameters. The response time of each service is shown in the underneath parenthesis. For clarity, we do not draw the duplicated services and the no-ops at the action layers. Assume we want two goals d_6 and d_7. Service w_5 can possibly start at the time point 40 when w_1 or w_3 is not finished, as at the time point 40 its input is ready. The best solution to this problem is $\{(w_1; w_4)||(w_3; w_6)\}$ and the optimal response time is $\max(T(d_6), T(d_7)) = \max(90, 250) = 250$.

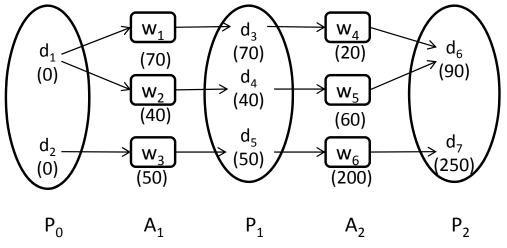

Fig. 4 An example to explain the scheduling of services over TPG

3.4 Redundant Activities

QoSGraphPlan calculates the lowest cost for each proposition, including the goals. *QoSGraphPlan* simply puts all the best paths to produce the individual goals together as a solution. This is the best solution in the sense that each goal is generated with the lowest cost. However, it is not necessary to use all the best paths to generate the goal propositions, because the QoS value for the whole plan is determined by the worst path. It is possible that the less optimal paths can still keep the same QoS value. Please check the following example.

Example 5 Figure 5 shows a planning graph with three services w_1, w_2, w_3 and their input and output parameters. The initial layer has d_1 and d_2 and the goal layer has two goals d_3 and d_5. There is one no-op action to connect d_3 in P_1 and P_2 layers. For clarity, we do not draw the duplicated services at the action layers.

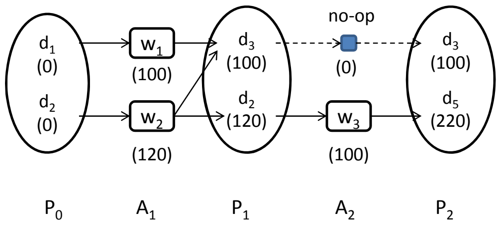

Fig. 5 An example to explain redundant services

By our TGP technique, the best value for d_3 is 100 and the best path to produce it is $\{w_1\}$. Similarly, the best value for d_5 is 220 and the best path to produce it is $\{w_2; w_3\}$. *QoSPlanGraph* puts the two paths together to get the solution $\{(w_1||w_2); w_3\}$. The response time for the whole plan is determined by the longest path which is T=220. If we consider the whole plan, we do not need to maintain the best path for d_3. This means we can remove w_1 from the solution. After the removal, we will get d_3 at T=120 which does not change the QoS value for the whole plan. We give a redundancy definition as below.

Definition 7 A plan without redundant services is a plan for which removing any action causes unsatisfied goals or increased utility value.

Redundancy is due to the optimization done on a single criterion. For a single criterion, redundancy removal does not change the optimal QoS value for the whole plan, thus is not necessary. However, redundancy removal implies reducing execution price. A full discussion on multiple criteria QoS optimization is in Subsection 3.6. In the rest of this subsection, we present a redundancy removal algorithm which can remove redundant services while keeping single QoS criterion for a solution unchanged. Now, we focus on the solution from *QoSGraphPlan* (Def. 8).

Definition 8 A **solution tagged planning graph** STPG is a subgraph of TPG with all the actions in a solution and the propositions connecting these actions.

A solution graph removes the actions which are not in the solution and the propositions which do not connect actions in the solution from TPG.

Definition 9 A **reproduced proposition** is a proposition which has more than one parent action in a STPG.

Proposition 1 *A necessary condition for an action a to be redundant is that all its post conditions in STPG are reproduced propositions, i.e.,$\forall p \in (post(a) \cap STPG)$, p is a reproduced proposition.*

With Proposition 1, if all the post conditions of an action a can also be produced by at least another action in the STPG, it is possible to remove a from the plan. However, this is only a necessary condition of redundancy. Redundancy also depends on the QoS values of the other actions. Please check the following example.

Example 6 Figure 6 is similar to Figure 5, only w_4 is added. The goals are $\{d_5, d_6\}$. The best value for d_6 is 240 which is the highest value among the goals. The best solution is $\{(w_1||w_2); (w_3||w_4)\}$. There are no redundant services, because if any of them is removed, either some goals are not satisfied or the QoS value increases. If we change the QoS value for w_4 to T=100, the best value for d_6 is 200. Then the QoS value for the entire plan becomes 220. And w_1 is redundant. This is because if we remove w_1, the QoS value for the whole plan does not change.

Example 6 shows that not only the connection relations but also the QoS values determine whether a service is redundant. It is not possible to determine whether a service is redundant except by trying it out. Algorithm *RemoveRedundancy* (Alg. 5) scans the STPG

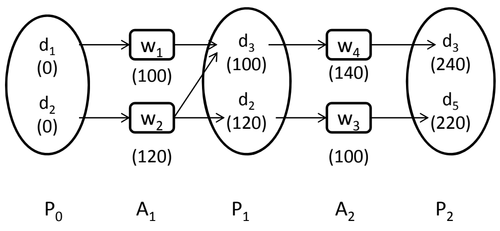

Fig. 6 An example to explain redundant services

from the goal layer towards the initial layer to probe whether a service is redundant by removing it from the STPG and computing whether the QoS value is changed. It uses Proposition 1 to pick the possible redundant services in a layer (line 2). The second condition in line 2 happens after some services are removed which may leave some propositions that are not used by any services (ref. description about line 11). We try to remove a possible redundant service and compute the QoS value for the goals (line 4). If the QoS value for the plan changed, the removed service is added back (line 5-6). If not, the arcs pointing to it are removed (line 8). A proposition that has no descendants is removed as well (line 11). A lean solution is returned in line 13.

Algorithm 5 *RemoveRedundancy*($\langle P_0, A_1, ..., A_n, P_n\rangle$)

```
⟨P_0, A_1, ..., A_n, P_n⟩) is a STPG
 1: for $i = n, \ldots, 1$ do
 2:     $A'_i = \{a | \forall a \in A$, $a$ is a possible redundant services $\vee$
          $a$ that has no descendants };
 3:     for $\forall a \in A'_i$ do
 4:         remove $a$ from the graph and calculate QoS values
              for the goals;
 5:         if the QoS for the plan worsened then
 6:             add $a$ back;
 7:         else
 8:             remove the arcs pointing to $a$;
 9:         end if
10:     end for
11:     remove $\forall p \in P_i$ that has no descendants, remove the
          arcs pointing to $p$;
12: end for
13: return $\langle P_0, A_1, ..., A_n, P_n\rangle$)
```

RemoveRedundancy scans the STPG once from the goal layer towards the initial layer. Its complexity is polynomial.

Theorem 4 *The complexity of RemoveRedundancy is polynomial.*

Proof: It needs n loops to scan the STPG graph $\langle P_0, A_1, ..., A_n, P_n\rangle$ once. To remove one service a from a layer i, we need to re-compute the QoS value from the layer i to the top layer n, which needs one scan. Maximally there are $|A|$ services at a layer to remove. Therefore, it needs maximally $|A| \times n$ loops to remove all the services at a layer. Therefore, we need $|A| \times n \times n$ loops in total. n's upper bound is $|A|$. Therefore, the complexity of *RemoveRedundancy* is $|A|^3$. □

RemoveRedundancy results in a lean solution without redundant services. In the cases that there are multiple removable services, *RemoveRedundancy* removes the one it confronts first. Which service is the best to remove is beyond discussion in this subsection, because a second QoS criterion should be used to find the optimal one.

For response time and throughput we have the following proposition.

Proposition 2 *For* ***response time*** *and* ***throughput*** *as a single criterion, Algorithms 1 to 5 can get composition solutions without redundancy, as well as the globally optimized QoS value in polynomial time.*

3.5 Calculate the Other Single QoS Criteria

The previous subsection calculates the optimal response time and throughput when they are used as a single criterion. In this section, we show how to calculate **execution price**, **successful execution rate**, and **availability**, when they are used as a single criterion. Different from response time and throughput, each service contributes to these QoS values of the plan equally, regardless of sequential or parallel connections. Please check the following example.

Example 7 Reuse Figure 6 for execution cost. Assume all the numbers represent the execution costs. We can compute $\forall p \in post(a)$, for execution price $cost(p) = \max(cost(pre(a)) + cost(a)$. QoSGraphPlan reports a best solution $\{(w_1||w_2); (w_4||w_3)\}$ for a best execution cost 460. If we use w_2 to generate both d_3 and d_4 and remove w_1 from the solution, we reduce the execution cost to 360.

Example 7 shows that if we use *QoSGraphPlan* for execution price, it may not get the correct optimal QoS value. This is because *QoSGraphPlan* calculates the best path for each goal proposition independently and puts together these paths as the optimal solution. For the criteria in this subsection, all the services in the solution contribute to the QoS of the whole plan equally. Therefore, removal of any services affects the QoS of the whole plan.

We can still use *RemoveRedundancy* to remove redundancy. The result of *RemoveRedundancy* is still a lean solution. However, *RemoveRedundancy* removes the redundant services in the sequence of inspection. When there are choices to remove different services, *RemoveRedundancy* does not do optimization. Therefore, the solution obtained after *RemoveRedundancy* may not have globally optimized the QoS value either. We have the following proposition.

Proposition 3 *For* ***execution price***, ***reputation***, ***successful execution rate***, *and* ***availability*** *as single criteria, Algorithms QoSGraphPlan and RemoveRedundancy can get composition solutions without redundancy in polynomial time, but they do not guarantee to get a globally optimized QoS value.*

In order to extend our method to be able to calculate the correct optimal QoS value for **execution price**, **reputation**, **successful execution rate** and **availability**, we modify the tags of the propositions to record all the possible paths. We use multiple tags for one proposition p. A tag t_j represents one execution path that leads to p, and has a list of parents and a QoS value $t_j = (\{t_j.parent_k\},\ t_j.v)$. Please check the following example.

Example 8 Figure 7 shows multiple tags are used for calculating execution price on a TPG. Each tag of a proposition corresponds to one path leading to the proposition. A tag records the parent actions and the cost to generate the proposition.

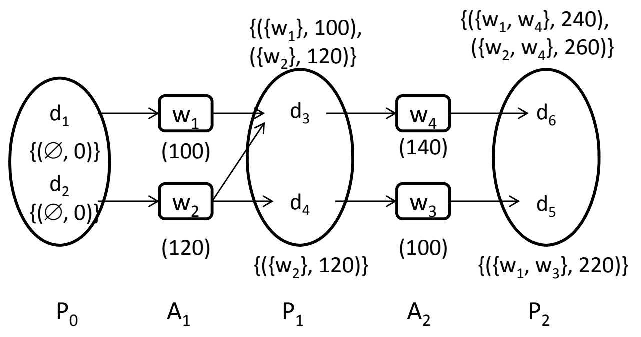

Fig. 7 An example to explain multiple tags.

In the multiple tag situation, we use Algorithm 6 to replace Algorithm 2 for expanding the TPG. Algorithm 6 calculates all the tags in line 2, which is the only difference to Algorithm 2.

We call the extended version *QoSGraphPlanExt* which uses Algorithms 1, 6, 3, and 4. *QoSGraphPlanExt* probes all the combinations, which can be huge. Assume a proposition at layer i has k tags in average and a service can be enabled by l paths from layer i. Then an output of this service has $k \times l$ tags. Therefore, a proposition at layer $i+1$ has $k \times l$ tags. If the graph expands to $i+2$ layer, a proposition at $i+2$ layer has $k \times l^2$ tags. If a graph has n layers and $k = 1$ at layer 0, a proposition has l^n tags at layer n. l and n are bounded by $|A|$ (A is the set of services). Therefore, *QoSGraphPlanExt* has exponential complexity. We propose a beam search algorithm in Subsection 3.7 to solve the exponential problem. We have the following proposition for the properties of *QoSGraphPlanExt*.

Algorithm 6 $ExpandGraphMultiTag(\langle P_0, A_1, ..., A_i, P_i\rangle, g)$

```
1: $A_i = \{(a,t)|pre(a) \subseteq P_{i-1}, a \in A, t = cost(a)\}$;
2: $P_i = \{(p,\{t_j\})|\exists a \in A_i : p \in effects(a), \{t_j\}$ is all the possible tags$\}$;
3: for each $a \in A_i$ do
4:     link $a$ with precondition arcs to $pre(a)$ in $P_{i-1}$;
5:     link $a$ with each of its $effects(a)$ in $P_i$;
6: end for
7: return $\langle P_0, A_1, ..., A_i, P_i\rangle$;
```

Proposition 4 *QoSGraphPlanExt gets a composition solution without redundant services and with the best QoS value for execution price, reputation, successful execution rate, and availability as single criterion in exponential time.*

Example 9 Suppose the quality criteria given in Table 1 is the **execution cost**. Figure 8 shows the multiple tags for each $p \in P_i$ ($i = 0, \ldots, 4$) in the expanded tagged planning graph for Example 1. At proposition layers in the TPG, propositions are separated by dashed lines. The tag of J at P_1 is $\{(\{w_1\}, 800/1)\}$ because there is only one execution path $\{w_1\}$ up to layer P_1 that leads to J. The best solution is $\{w_2; w_4; w_8; w_7\}$ with the minimal execution cost of 600, w.r.t. the utility value of 0.29.

3.6 Under Multiple Criteria

When we need to consider multiple QoS criteria, we aggregate them into a utility value according to Equation 21. We use the utility values to compare the different paths. As different QoS criteria have different formulas to calculate their values, we need to calculate the individual QoS values separately at each search step, before aggregating them. Therefore, for a proposition in the TPG, the label can be as shown in following example.

Example 10 Assume a proposition p has two paths to reach $\{w_1, w_2\}$ and $\{w_1, w_3, w_4\}$. Table 4 shows the QoS values and the aggregated utility value for the two paths. We can represent the tags as $\{(\{w_1, w_2\}, 10, 20, 30, \ldots, 0.6), (\{w_1, w_3, w_4\}, 15, 20, 35, \ldots, 0.8)\}$. Compared by their utility values, $\{w_1, w_2\}$ is better than $\{w_1, w_3, w_4\}$.

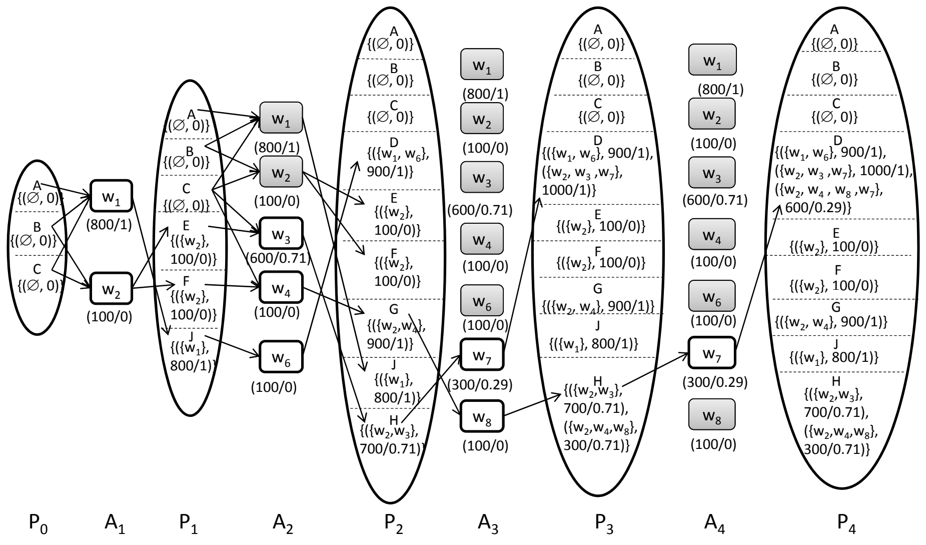

Fig. 8 The multiple tags used for optimizing execution cost.

Table 4 An example of multiple criteria tag

paths	Q1	Q2	Q3	...	Utility
$\{w_1, w_2\}$	10	20	30	...	0.6
$\{w_1, w_3, w_4\}$	15	20	35	...	0.8

As we try all the combinations, we can get the best solution which has no redundant services to remove at the end. When the combinations are huge, we can use heuristics to tackle the problem as developed in the next subsection.

3.7 BeamQoSGraphPlan with beam search

To solve the combination explosion problem, we incorporate *QoSGraphPlan* with beam search. Beam search uses breadth-first search to build its search tree [31]. At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of heuristic cost. However, it only stores a predetermined number of states at each level (called the beam width). The greater the beam width, the fewer states are pruned. With an infinite beam width, no states are pruned and beam search is identical to breadth-first search. The beam width bounds the memory required to perform the search. Since a goal state could potentially be pruned, beam search sacrifices completeness (the guarantee that an algorithm will terminate with a solution, if one exists) and optimality (the guarantee that it will find the best solution).

We use beam search to keep only a few best tags for each proposition. We propose a heuristic function in our application as denoted by Equation 35. The heuristic function is a weighted sum of two functions. The first function $U(\{t_j.parents\})$ is the utility value of the path. $\left|\bigcup_{a\in t_j.parents} effect(a) \cap g\right|$ is the number of goal propositions that the outputs of services on the path can satisfy. $\frac{|g|-\left|\bigcup_{a\in t_j.parents} effect(a)\cap g\right|}{|g|}$ is the percentage of unsatisfied goals, which acts as an estimation of the distance from current proposition to a goal state. A t_j with lower heuristic value is a better choice. Suppose K is the beam width, we use Q_p^K represent the top K tags sorted by $h(t_j)$.

BeamQoSGraphPlan uses beam search to keep only top K tags for each search step. K is the beam width. We use Algorithm 7 to replace Algorithm 2 to expand the plan graph. Line 2 includes the function to select the top K tags for a proposition. Therefore, *BeamQoSGraphPlan* includes Algorithm 1, 7, 3, and 4.

$$
\begin{aligned}
|h(t_j)| &= k_1 \times \sum U_i(\{t_j.parents\}) \\
&+ k_2 \times \frac{|g| - \left|\bigcup_{a\in t_j.parents} effect(a) \cap g\right|}{|g|} \\
&\text{where } k_1 + k_2 = 1
\end{aligned}
\tag{35}
$$

Algorithm 7 *ExpandGraphBeamWidth*$(\langle P_0, A_1, ..., A_i, P_i\rangle, g)$

```
1: $A_i = \{(a,t) | pre(a) \subseteq P_{i-1}, a \in A, t = cost(a)\}$;
2: $P_i = \{(p, Q_p^K) | \exists a \in A_i : p \in effects(a)$ , $Q_p^K$ is the set
   of top $K$ tags associated with proposition $p\}$;
3: for each $a \in A_i$ do
4:     link $a$ with precondition arcs to $pre(a)$ in $P_{i-1}$;
5:     link $a$ with each of its $effects(a)$ in $P_i$;
6: end for
7: return $\langle P_0, A_1, ..., A_i, P_i\rangle$;
```

BeamQoSGraphPlan can be used to solve the tag explosion problem for both single criterion and multiple criteria problem. It is a heuristic algorithm. It does not guarantee optimal QoS and its solution may include redundant services.

3.8 Summary of the Algorithms Developed

As a summary, we list the properties of all the algorithms in this paper as below.

QoSGraphPlan:

– *QoSGraphPlan* uses Algorithms 1, 2, 3, and 4 to first construct a TPG and then extract a solution from the TPG.
– Best used for throughput and response time as single criterion (best QoS value, with redundant services).
– If used for execution price, reputation, successful execution rate, and availability as single criterion, its solution may have redundant services and may not have the optimal QoS value.
– Complexity: polynomial.
– Use *RemoveRedundancy* (Alg. 5) to remove redundancy.

QoSGraphPlanExt:

– *QoSGraphPlanExt* uses Algorithm 1, 6, 3, and 4 to first construct a TPG with multiple tags and then extract a solution from the multiple-tag TPG.
– Best used for execution price, reputation, successful execution rate, and availability as single criterion.
– Complexity: exponential.
– Solution does not have redundant services and guarantees the optimal QoS value.

BeamQoSGraphPlan:

– *BeamQoSGraphPlan* uses Algorithms 1, 7, 3, and 4 to incorporate *QoSGraphPlan* with beam search.
– Best used for execution price, reputation, successful execution rate, and availability as single criterion, or all the multiple criteria cases.
– Heuristic algorithm.
– Solution may have redundant services and may not have the optimal QoS value.
– Use *RemoveRedundancy* (Alg. 5) to remove redundancy.

RemoveRedundancy:

– *RemoveRedundancy* as presented in Algorithm 5 removes redundant services from a solution to obtain a lean solution without redundancy.
– Used for redundant service removal for single or multiple criteria.
– No optimization on which service to remove.
– No guarantee to get the optimal QoS value.

4 Empirical Results

4.1 Data set

The data set used in our evaluation is generated by the test set generator in Web Service Challenge 2009 (WSC09) [2]. The data generator generates a service composition problem through the generation of Web services in a WSDL file, ontology concepts in an OWL file, and QoS values for Web services in a WSLA file. The WSDL file is annotated with a simple extension mechanism to link to the ontology definition in the OWL file, instead of using full-fledged SAWSDL [28]. The Web service parameters are instances (“things” in an OWL file) of the semantic concepts in OWL files. The user can control the generated dataset by specifying the number of services, the number of concepts, and the number of solutions and their length (in action steps). Given those parameters, the generator randomly creates a set of concepts and selects a subset of these concepts as the goals. Then, the generator returns several groups of solutions at given lengths. When generating a group of solutions, the generator prepares a set of inputs and outputs at each time step. A set of services are then generated, each of which can independently use these inputs/outputs. Thus, a group of solutions are generated. Within a group, some services can directly substitute others as they use the same input set and produce the same outputs. The generator randomly adds a lot of “padding” Web services around the services used in solutions. These “padding” services do not have the outputs that can be used by the services within a solution. Each Web service in the data set has a throughput and a response time defined in a WSLA file.

4.2 Implementation

We have implemented all the algorithms presented in this paper. We have also developed a verification tool to check the correctness of the obtained solutions. We have used a technique similar to those developed in [33] to **flatten the semantics and index the data.** This has proved to be important in speeding up the algorithms. It works as the following example.

In the example in Figure 9, there are 4 concepts. “Machine” subsumes the concept “Vehicle”, and “Vehicle” subsumes “Car” and “Motorcycle”. Web service A has an output “Ford 1986 Red” which is an instance of "Car" and Web service B accepts an input "An old Vehicle" which is an instance of "Vehicle". By checking the semantic relationships, we can know that the output of Web service A can be acceptable by Web service B because "Car" is also a kind of "Vehicle".

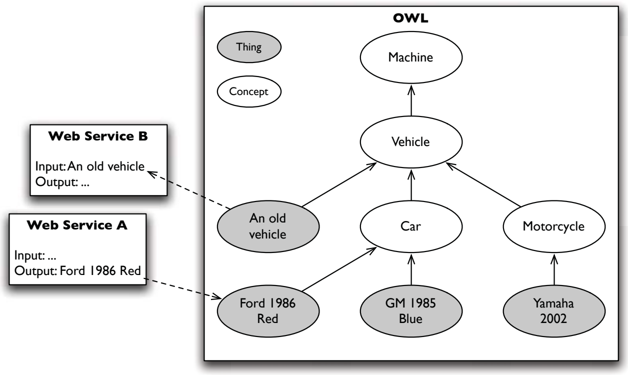

Fig. 9 Semantic Relationship Between Web Service Input/Output parameters

Service	Outputs of the Services
A	Car, Vehicle, Machine
B	. . .
...	...

Input Concept	Services
Car	. . .
Vehicle	B
Machine	...

Table 5 Top: Example of the Output Indexing Table; Bottom: Example of the Reverse Indexing Table

Rather than checking the relationship map in OWL every time we need to find a list of invokable services, we build two indexing tables as shown in Table 5. The indexing tables are stored as hash tables so that we can look up the services or the parameters in constant time.

4.3 Empirical Results

As the WSC09 data sets and the results are posted at [2], we are able to compare our results with the first place winning paper [14] and the second place winning paper [34] in terms of response time and throughput of the solutions. WSC09 has five data sets. 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. Every data set has a WSLA file to describe response time and throughput of services. Other QoS values, such as execution price and reputation, are not available. For the experimental purpose, we take response time and throughput as some other QoS values. We run the experiments on a laptop with Intel(R) Core(TM)2 2.60GHz Duo CPU and 3.00GB of RAM. The algorithms are implemented in Java.

In Table 6, we use *QoSGraphPlan* to work on throughput and response time as single criterion. To refine the solutions, we use *RemoveRedundancy* to remove redundant services. We show whether the correct best QoS values can be calculated (the checkmarks), how many services are in the solution (the lines of #Services), and how many services are redundant (lines of #Redunt). We can see that *QoSGraphPlan* can find the correct QoS values for all the five datasets, as the first place winner does. Our method generates solutions without redundant services or very few redundant services (≤ 4) in the solutions on all data sets.

The first place winners [14] generate zero redundant services in some datasets, but much more redundant services in the other datasets, especially when they compute throughput. [14] uses Dijkstra's principle to calculate the optimal value while searching all composition alternatives. It uses a table to record all the enabled services at a time step. All the enabled services and parameters have a current best quality value, which is a similar idea to ours. However, planning graph is a much more matured graph designed for planning than the graph in [14]. For example, for a planning problem, one should be able to reuse the same action multiple times in the plan, otherwise one may not find an existing solution. [14] seems to filter out this possibility because the actions are not reused in their graph. Also, without using the concept of no-op, [14]'s graph loses the information about which services could have produced a proposition, which is important in order to remove the redundant services.

The second place winners [34] fail to find the correct QoS for Dataset4 and produce comparably more redundant services on all the datasets than our method and the first place winner. It is because [34] uses a simple breadth first search which could get only sequential solutions. Therefore, [34] is not able to get the optimal QoS value correctly, if some services can be concurrently executed.

According to the comparison, *QoSGraphPlan* algorithm can find a solution that contains fewer redundant services and has the optimal throughput or response time on all data sets. *RemoveRedundancy* algorithm working on an optimal solution makes the solution contain no redundant services.

In Table 7, we show the composition time with redundant services (Comp T1, time used by *QoSGraph-*

	Dataset1	Dataset2	Dataset3	Dataset4	Dataset5
Resp. Time	✓/✓/✓	✓/✓/✓	✓/✓/✓	✓/✓/✓	✓/✓/✓
#Services	8/5/18	21/20/52	10/10/18	42/93/133	32/32/4772
#Redunt.	3/0/13	1/0/32	0/0/8	2/53/93	0/0/4740
Throughput	✓/✓/✓	✓/✓/✓	✓/✓/✓	✓/✓/-	✓/✓/✓
#Services	5/7/9	21/25/36	30/26/81	65/73/159	32/45/4772
#Redunt.	0/2/4	1/5/16	4/0/55	3/11/94	0/13/4740

Table 6 Results with the WSC09 Data Sets: our method/Paper [14]/Paper [34]

	Dataset1	Dataset2	Dataset3	Dataset4	Dataset5
Comp T1 (ms)	90/75	470/449	531/535	2209/4176	1787/1951
Comp T2 (ms)	157/105	517/494	533/1502	2248/8318	1791/2028

Table 7 Our composition time: T1 with redundant service (Resp. Time/Throughput); T2 without redundant services (Resp. Time/Throughput).

Solution		Dataset1	Dataset2	Dataset3	Dataset4	Dataset5
Before removing redundant services	Total execution price	80000	344000	82000	729000	319000
	#Services	7	38	12	85	45
	Comp T1 (ms)	219	2740	1675	5977	5032
After removing redundant services	Total execution price	67000	208000	62000	339000	218000
	#Services	5	20	10	42	32
	Comp T2 (ms)	225	2744	1678	5987	5045

Table 8 Total execution price on the WSC09 Data Sets

Plan) and without redundant services (Comp T2, time used by *QoSGraphPlan* and *RemoveRedundancy*) for both response time and throughput. The composition time T1 and T2 are the average of 10 runs, since we run the same experiment over each dataset 10 times. We can see that the computation time of removing redundant services, i.e. T2-T1, is comparatively small compared to the time spent in finding a solution with redundant services (Comp T1) on all data sets. As no source code is provided by the WSC09 teams, we cannot compare our composition time with theirs' on one machine.

In Table 8, we use *BeamQoSGraphPlan* on execution price as single criterion[2]. We show the solution with the minimal total execution price for each data set, how many services are in the solution (the lines of #Services) and composition time before and after removing redundant services. We use *RemoveRedundancy* to remove redundant services from the solution. As for the heuristic function in *BeamQoSGraphPlan*, we only consider the QoS of execution price. In Equation 35, we set $k_1 = 0.8$, $k_2 = 0.2$, and beam width $K = 5$ for the heuristic function .

According to the results in Table 8, *BeamQoSGraphPlan* algorithm can find a solution with the minimal execution price on all datasets. *RemoveRedundancy* algorithm can remove redundant services from the solution. Redundancy removal reduces the total execution price of a solution.

In Table 9, we use *BeamQoSGraphPlan* on execution price and reputation as multiple criteria[3]. In Equation 21, we set $W_1 = 0.5$ for the execution price and $W_2 = 0.5$ for the reputation to calculate the aggregated utility value of execution price and reputation. In Equation 35, we set $k_1 = 0.8$, $k_2 = 0.2$ and beam width $K = 5$ to calculate the heuristic function. Table 9 shows the utility value, the number of services (the lines of #Services) in the best solution, and the composition time before and after removing redundant services (Comp T1 and Comp T2 respectively). The composition time T1 and T2 are the average of 10 runs, since we run the same experiment over each dataset 10 times. We use *RemoveRedundancy* to remove redundant services from the solution.

According to the results in Table 9, *BeamQoSGraphPlan* algorithm can find the solution with the optimal aggregated QoS values of execution price and reputation on all datasets. *RemoveRedundancy* algorithm can remove redundant services from the optimal solutions.

[2] We take the value of throughput of a service as its execution price for this experiment.

[3] We take the value of throughput of a service as its execution price, and the value of response time as its reputation for this experiment.

Solution		Dataset1	Dataset2	Dataset3	Dataset4	Dataset5
Before removing redundant services	Utility value	0.391	0.393	0.327	0.405	0.357
	#Services	10	27	12	83	44
	Comp T1 (ms)	323	4086	1905	7801	5190
	Total execution price	93000	228000	92000	761000	352000
	Average reputation	331	307.04	351.67	313.73	330.68
After removing redundant services	Utility value	0.391	0.374	0.314	0.392	0.345
	#Services	10	18	10	43	34
	Comp T2 (ms)	326	4089	1908	7811	5204
	Total execution price	93000	151000	68000	381000	244000
	Average reputation	331	324.45	342	318.37	321.47

Table 9 Result considering aggregated value of execution price and reputation on the WSC09 Data Sets

5 Related Work

We study a QoS-aware service composition problem that is to connect SOAP services into a network by matching their parameters to achieve some business goals so that the resulted network can satisfy functional goals and QoS optimization at the same time. This problem is also called vertical composition in [12].

Another QoS related problem is the service selection problem [39,37]. Different from QoS-aware service composition, this kind of problem has a predefined business process template and each task in the business process can be fulfilled by a set of services with varied QoS. The objective is to select a set of services that can optimize the QoS of the entire process. This problem is also called horizontal composition in [12]. Please notice in both problems, only SOAP services, *i.e.,*services without internal behavior, are used.

For both problems, people need some methods to compare the services under multiple QoS criteria. Most papers use some multiple criteria decision-making (MCDM) techniques to aggregate multi-criteria QoS values into an overall score [39,37]. Our paper follows this method. A skyline technique can be used to calculate a set of dominating services [4], instead of getting one best service based on the overall score. [36] considers computing service skyline from uncertain QoS values. These techniques are listed in Table 10.

To solve the QoS-aware composition problem, people have used optimization algorithms like Dynamic Programming (DP) [14,15] and Integer Programming (IP) [9] to find a QoS optimized solution. [14,15] build a graph of service connections and use DP on the graph. They combine all the optimal paths to reach individual functional goals to be the optimal solution. This kind of optimal solutions can contain multiple redundant services that removing them does not worsen the QoS value. If considering redundant services can make the plan more expensive, removing them is needed. [15] does not consider the possibility of reusing the same service in a plan. Maybe the authors are more capable in optimization but not so familiar with planning. IP is based on a given linearized formulation. It involves proper assignment of variables and some tweaks may be needed to make the relation among the variables linear. People can use standard IP solver to find a solution. If modeling properly, IP can avoid the problem of redundant services. IP is good at calculating the solution from scratch. IP is generally NP-hard, though some IP problems are solvable in polynomial time. Our method is based on planning algorithm and combine planning with optimization. It takes the advantage of known theories in both planning and optimization. For example, the actions are reusable in planning and the redundant services have no meaning in planning, while using only optimization might forget these features. Our research studies the individual criteria more carefully and find out that response time and throughput optimization can be solved in polynomial time, which is not reported in literatures using IP. Another advantage to introduce planning graph is the possibility to reuse the graph to answer different queries very fast without calculating composition from scratch. In our previous research [35], we developed such a technique without QoS. We hope this paper and [35] can pave the road to QoS aware plan adaptation. The techniques to solve the QoS-aware composition problem as discussed above are listed in Table 10.

The service selection problem is a combinatorial optimization problem. Integer programming is a powerful tool to solve it [39]. As this is an NP-complete problem, heuristic search can be applied to search the problem space only partially [37,1]. Genetic Algorithm (GA) is another way to partially search the problem space [6]. And the advantage of GA compared to integer programming is that GA can deal with nonlinear constraints of QoS requirements. [12] models the problem as a constraint satisfaction problem (CSP). Rooted on AI, CSP can model hard constraints, *i.e.,*the functional requirements in ASC, and soft constraints, *i.e.,*

Multiple QoS criteria	Overall score	[39,37]
	Skyline	[4,36]
QoS-aware Service Composition (functional+QoS)		DP [14,15], IP [9], our method: planning+optimization
Service selection		CSP [12,18], IP [39], GA [6], heuristic search [37,1]

Table 10 QoS related service composition methods

optimizing QoS in ASC. The penalty of violating a hard constraint is infinite, vs a finite penalty to sub-optimal QoS values. Therefore, the CSP algorithms search for a solution which minimizes the penalty. [12] proposes an interactive algorithm for solving the problem with user inputs. Their model can be used to solve the vertical composition problem as well. [18] uses fuzzy constraints to model service clients and providers' preferences and services can communicate with each other to find an optimal solution in a distributed manner. All the techniques to solve service selection problem as discussed above are listed in Table 10.

6 Conclusion

In this paper, we present a new way to solve the QoS-aware service composition problem. We use Dijkstra's algorithm on the planning graph to optimize the QoS, satisfying the functions goals at the same time. We extend Dijkstra's algorithm to handle multiple source graphs like the planning graph. We discuss how to calculate the optimal QoS values for different single criterion problems, as well as multiple criteria problem. For throughput and response time as single criterion, we have a polynomial algorithm to get the optimal QoS value, and a solution without redundant services. For the other single criterion problems and the multiple criteria problems, we have only an exponential algorithm. In this case, we use beam search which is a heuristic algorithm to get feasible solutions.

References

1. Rainer Berbner, Michael Spahn, Nicolas Repp, Oliver Heckmann, and Ralf Steinmetz. Heuristics for qos-aware web service composition. In *Proc. of ICWS*, pages 72–82, 2006.
2. Steffen Bleul. Web service challenge rules. `http://ws-challenge.georgetown.edu/wsc09/downloads/WSC2009Rules-1.1.pdf`, 2009.
3. Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence Journal*, 90(1-2):225–279, 1997.
4. Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proc. of ICDE*, pages 421–430, 2001.
5. Athman Bouguettaya, Qi Yu, Xumin Liu, and Zaki Malik. Service-centric framework for a digital government application. *IEEE T. Services Computing*, 4(1):3–16, 2011.
6. Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proc. of GECCO*, pages 1069–1075, 2005.
7. K.S.May Chan, Judith Bishop, and Luciano Baresi. Survey and comparison of planning techniques for web service composition. Technical report, Dept Computer Science, University of Pretoria, 2007.
8. Min Chen and Yuhong Yan. Redundant service removal in qos-aware service composition. In *Proc. of IEEE ICWS*, pages 431–439, 2012.
9. LiYing Cui, Soundar Kumara, and Dongwon Lee. Scenario analysis of web service composition based on multi-criteria mathematical programming. *INFORMS Service Science*, 3(3), 2011.
10. Michael Lawrence Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved optimalization problems. *Journal of ACM*, 34, 1987.
11. Joyce El Haddad, Maude Manouvrier, and Marta Rukoz. Tqos: Transactional and qos-aware selection algorithm for automatic web service composition. *IEEE T. Services Computing*, 3(1):73–85, 2010.
12. Ahlem Ben Hassine, Shigeo Matsubara, and Toru Ishida. A constraint-based approach to horizontal web service composition. In *Proc. of ISWC*, pages 130–143, 2006.
13. Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
14. Zhenqiu Huang, Wei Jiang, Songlin Hu, and Zhiyong Liu. Effective pruning algorithm for qos-aware service composition. In *Proc. of CEC*, pages 519–522, 2009.
15. Wei Jiang, Charles Zhang, Zhenqiu Huang, Mingwen Chen, Songlin Hu, and Zhiyong Liu. Qsynth: A tool for qos-aware automatic service composition. In *Proc. of ICWS*, pages 42–49, 2010.
16. Steven M. LaValle. *Planning Algorithms*. Cambridge, 2006.
17. Dana Nau Malik Ghallab and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers, 2004.
18. Xuan Thang Nguyen, Ryszard Kowalczyk, and Manh Tan Phan. Modelling and solving qos composition problem using fuzzy discsp. In *Proc. of ICWS*, pages 55–62, 2006.
19. OASIS. Uddi version 2.04 api specification. `http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm`, 2007.
20. OASIS. Web services business process execution language (ws-bpel). `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`, 2007.
21. Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing:

a Research Roadmap. *International Journal of Cooperative Information Systems*, 17(2):223–255, 2008.
22. Joachim Peer. Web Service Composition as AI Planning – a Survey. Technical report, University of St.Gallen, 2005.
23. Marco Pistore, Paolo Traverso, and Piergiorgio Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. of ICAPS*, pages 2–11, 2005.
24. Pascal Poizat and Yuhong Yan. Adaptive composition of conversational services through graph planning encoding. In *Proc. of IsoLA*, pages 35–50, 2010.
25. Jinghai Rao and Xiaomeng 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.
26. Evangelos Triantaphyllou. *Multi-Criteria Decision Making: A Comparative Study*. 2000.
27. W3C. Owl web ontology language overview. `http://www.w3.org/TR/owl-features/`. Retrieved 2011-06-30, 2004.
28. W3C. Semantic annotations for wsdl and xml schema (sawsdl). `http://www.w3.org/TR/sawsdl/`. Retrieved 2011-06-30, 2007.
29. W3C. Soap version 1.2 part 1: Messaging framework (second edition). `http://www.w3.org/TR/soap12-part1/\#intro`. Retrieved 2011-06-30, 2007.
30. W3C. Web services description language (wsdl) version 2.0. `http://www.w3.org/TR/wsdl20/`. Retrieved 2011-06-30, 2007.
31. Wikipedia. Beam search. `http://en.wikipedia.org/wiki/Beam_search`. Retrieved 2011-09-02.
32. Wikipedia. Dijkstra's algorithm. `http://en.wikipedia.org/wiki/Dijkstra'27_algorithm`. Retrieved 2011-06-30.
33. Yixin Yan, Bin Xu, and Zhifeng Gu. Automatic service composition using and/or graph. In *Proc. of CEC/EEE*, pages 335–338, 2008.
34. Yixin Yan, Bin Xu, Zhifeng Gu, and Sen Luo. A qos-driven approach for semantic service composition. In *Proc. of CEC*, pages 523–526, 2009.
35. Yuhong Yan, Pascal Poizat, and Ludeng Zhao. Repair vs. recomposition for broken service compositions. In *Proc. of ICSOC*, pages 152–166, 2010.
36. Qi Yu and Athman Bouguettaya. Computing service skyline from uncertain qows. *IEEE T. Services Computing*, 3(1):16–29, 2010.
37. Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web*, 1(1), 2007.
38. Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality driven web services composition. In *Proc. of WWW*, pages 411–421, 2003.
39. Liangzhao Zeng, Boualem Benatallah, Anne H. H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Software Eng.*, 30(5):311–327, 2004.
40. Xianrong Zheng and Yuhong Yan. An Efficient Web Service Composition Algorithm Based on Planning Graph. In *Proc. of ICWS*, pages 691–699, 2008.