Anytime QoS Optimization over the PlanGraph for Web Service Composition

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

Yubin Yang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing 210093, China

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. In this paper, we 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. In addition, we make our algorithm an anytime algorithm that has the advantage of getting better solutions if it keeps 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 [28], WSDL [29], and UDDI [18]. With these open standards, Web services are automatically invokeable 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 [20, 24]. 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 [22, 23, 39]. 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.

In addition to satisfying business goals, recent research moves to select the best services to optimize the QoS, such as throughput and response time, of the target business process. Without QoS consideration, the planning algorithms normally search for a shortest plan. This actually implies the execution time/cost for each action is uniform. With QoS consideration, a shortest plan may not be the preferred one, because a longer plan may have faster response time, or less cost, or higher throughput. Therefore, we need to modify the classic planning algorithm to find a QoS optimized solution.

On the other hand, people use algorithms like Dynamic Programming [14, 15] and Integer Programming [8] 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 also tend to solve the QoS aware service composition problem in two steps. The first step is to choose a control flow which becomes the template of the target business process; then the services are selected for each task in the control flow in the second step. The second step is the so-called service selection problem which is NP-complete [38, 36]. We find that this kind of problem makes sense only if we have to use a predefined control flow. Otherwise, we should decide control flow and services at the same time in QoS aware service composition. This is because it is possible that some other control flows with some other services can have better QoS than the predefined control flow can achieve. We prove in this paper that single criterion QoS aware service composition for throughput and response time is a polynomial time problem. This means the two step approach is not necessary.

In this paper, we present a QoS aware Graph Plan algorithm which uses the principle of Dijkstra’s algorithm to systematically search the best QoS path on a Plan Graph. Our motivation is to combine the AI planning model with an optimization algorithm to optimize the QoS during the search for satisfying functional requirements. We extend Dijkstra’s algorithm from handling single input/output to handling multiple inputs and outputs and parallel branches on a graph, like the Plan Graph. Our approach 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. In addition to satisfying functional requirements, our algorithm can get a globally QoS optimized solution. Moreover, our algorithm is an anytime algorithm [30] that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

©2012 ACM 978-1-4503-0857-1/12/03 ...$10.00.
can return better and better solutions the longer it keeps running. 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. We can also associate semantic information to inputs and outputs using SAWSDL [27] or OWL [26]. OWL represents knowledge as a set of concepts within a domain and the relationships between those concepts. The data sets used in the data experiments in Section 4 do contain semantic relations among the input and output concepts.

Definition 1. Given a set $D$ of concepts, a service $w$ is a tuple $(\text{in}(w), \text{out}(w), Q(w))$, where $\text{in}(w) \subseteq D$ (resp. $\text{out}(w) \subseteq D$) 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 $w_1, w_2, \ldots, w_n$. 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 criteria are response time, throughput, execution price, reputation and availability. We focus only on response time and throughput in this paper. The extension to the other QoS criteria is discussed in 3.5. The definition of response time and throughput for a service $w$ and the aggregated value over $\sigma$ [11, 2] are as follows:

- **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.
  \[ Q_1(w_1; \ldots; w_n) = \sum Q_1(w_i) \] (1)
  \[ Q_1(w_1 || \ldots || w_n) = \max Q_1(w_i) \] (2)

- **Throughput** $Q_2(w)$: the average rate of successful message delivery over a communication channel, e.g., 10 successful invocations per second.
  \[ Q_2(w_1; \ldots; w_n) = \min Q_2(w_i) \] (3)
  \[ Q_2(w_1 || \ldots || w_n) = \min Q_2(w_i) \] (4)

Definition 2. A **service composition problem** is a tuple $(W, D_{in}, D_{out}, Q)$, 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.

Response time is a negative criterion, i.e., the higher the value, the lower the quality. Throughput is 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 (MCEDM) technique [25] to aggregate QoS value $Q(w)$. First, we scale a quality value for a service $w_i$. For response time, the values are scaled according to Equation 5. For throughput, the values are scaled according to Equation 6. The lower the utility value $U_1(w_i)$, the higher the quality for all the criteria. Please notice that other papers, like [37] and [5], do a similar conversion. But they normally use an increasing function of $Q_i(w_i)$. We use a decreasing function for its convenience in using Dijkstra’s algorithm, because the classic Dijkstra’s algorithm finds the “shortest distance” over a graph.

\[ U_1(w_j) = \begin{cases} 
\frac{Q_1(w_j) - Q_1^{\min}}{Q_1^{\max} - Q_1^{\min}} & \text{if } Q_1^{\max} - Q_1^{\min} \neq 0 \\
1 & \text{if } Q_1^{\max} - Q_1^{\min} = 0 
\end{cases} \] (5)

\[ U_2(w_j) = \begin{cases} 
\frac{Q_2(w_j) - Q_2^{\min}}{Q_2^{\max} - Q_2^{\min}} & \text{if } Q_2^{\max} - Q_2^{\min} \neq 0 \\
1 & \text{if } Q_2^{\max} - Q_2^{\min} = 0 
\end{cases} \] (6)

The overall quality score for each Web service:

\[ Q(w_j) = \sum U_i(w_j) \times W_i \] (7)

where $W_i \in [0, 1]$ and $\sum W_i = 1$. $W_i$ represents the weight of criterion $i, i = 1, 2$.

For a network of services $\sigma$, we would like to do the same conversion. The following equations for defining $U_1(\sigma)$ are the same as those defining $U_i(\sigma)$, just changing $Q_i(w_i)$ to $U_i(w_i)$, except in Equations 10 and 11, $\max$ replaces $\min$, because $U_2$ decreases when $Q_2$ increases. Thus, the max value of $U_1$ corresponds to the $\min$ value of $Q_1$.

\[ U_1(w_1; \ldots; w_n) = \sum U_1(w_i) \] (8)
\[ U_1(w_1 || \ldots || w_n) = \max U_1(w_i) \] (9)
\[ U_2(w_1; \ldots; w_n) = \max U_2(w_i) \] (10)
\[ U_2(w_1 || \ldots || w_n) = \max U_2(w_i) \] (11)

The aggregated utility value for $\sigma$ is:

\[ U(w_1, \ldots, w_n) = \sum U_i(w_1, \ldots, w_n) \times W_i \] (12)

With Equations 8 to 12, we could compare two networks of services by the individual utility values $U_i$ or by the overall score $U$. The lower the value, the higher the quality of service. We uniform the increasing and decreasing sense of the quality criteria. But the calculation of the precise values is still different for different criteria. In Section 3, we use response time to present our algorithm, then different QoS criteria are discussed.

2.2 The Planning Technique and Graph Plan

AI planning [10] has been applied with success to static service composition [21, 7], 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 Graph Plan [3]. Recent works have demonstrated the suitability of this model for ASC [23, 39]. Graph Plan is particularly interesting for
our idea of applying Dijkstra’s algorithm to it, because the planning graph built (cf. below) is firstly a graph, secondly a compact representation of the problem world, rather than a model for a specific goal g. This makes it possible to do a systematic search on the planning graph.

The Graph Plan approach contains two phases. The planning graph construction phase builds the planning graph from P0. 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 [39], it is possible to map a service composition problem (W, D_in, D_out, Q) to a planning problem P = ((S, W, γ), D_in, D_out) with service inputs being mapped to action preconditions (in(w) → pre(w)) and outputs to positive effects (out(w) → effects out(w)). Plans can be encoded in any orchestration language with assignment, sequence, and parallel operators, e.g., WS-BPEL [19]. Additionally, planning graphs enable 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 and 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 can construct a planning graph as in Figure 1.

In Figure 1, the no-op actions are represented by dashed arrows. A no-op action inherits a true proposition from a previous proposition layer. It has no cost. A no-op action is preferred than an non no-op action during the plan extraction phase. At an actions layer, all the enabled actions can be added, including those possibly added in the previous layers (the shaded actions in Fig 1. To make the figure readable, we do not draw all the no-op arcs on A2, 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 A3. There are three solutions: \{w1, w6\} \{w2, w3, w7\} and \{w2, w4, w6, w7\}.

<table>
<thead>
<tr>
<th>(w_i)</th>
<th>in (w_i)</th>
<th>out (w_i)</th>
<th>(Q_i)</th>
<th>(w_i)</th>
<th>in (w_i)</th>
<th>out (w_i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A, B, C</td>
<td>J</td>
<td>800</td>
<td>w5</td>
<td>K</td>
<td>H</td>
<td>600</td>
</tr>
<tr>
<td>B, C</td>
<td>E, F</td>
<td>100</td>
<td>w6</td>
<td>J</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>C, E</td>
<td>H</td>
<td>600</td>
<td>w7</td>
<td>H</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>C, F</td>
<td>G</td>
<td>100</td>
<td>w8</td>
<td>G</td>
<td>H</td>
<td>100</td>
</tr>
</tbody>
</table>

### 2.3 Dijkstra’s Algorithm

Dijkstra’s algorithm’s goal is to find single-source shortest paths in a graph [17]. 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 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 a finite graph.

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 \(C: V \rightarrow [0, \infty]\). 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 \(D(v)\).

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 that is 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 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 [9].

We can only describe the principle of Dijkstra’s algorithm in this paper. More examples and the pseudo code can be found in [31].

### 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 use the same principle to calculate the best cost-to-come for a proposition, which is the best-so-far QoS value for producing a proposition. Then, we could get the overall cost-to-come for all the goal propositions. And during the search, we could...
record the best-so-far 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 paths. As all the possible 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 a graph to solve a particular problem. Therefore, we could visit all the possible system states over the planning graph. This makes Dijkstra’s principle work over the planning graph.

Yet, we need to overcome some difficulties. First of all, Dijkstra’s algorithm is for single source situations, i.e., one edge represents one path between two vertices. While in planning graph, a service takes multiple inputs which could possibly come from multiple services. Therefore, the edges on the planning graph are not independent from one another. Second, the planning graph presents both parallel and sequential connections between services, which is different from a normal graph. 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 subsection.

3.2 Generation of Tagged Planning Graph

The classic Planning Graph \( G = (V, E) \) is a Directed Acyclic Graph (DAG). It has two kinds of vertices \( V = V_A \cup V_T \) where \( V_A \) are the vertices representing actions and \( V_T \) are the vertices representing propositions. Edges \( E = (V_T \times V_A) \cup (V_A \times V_T) \) connect the vertices. The edges \( (V_T \times V_A) \) connect the input parameters with the actions, while \( (V_A \times V_T) \) connect the actions with their output parameters.

The Tagged Planning Graph (TPG) associates a real value on the vertices of the planning graph:

**Definition 3.** The Tagged Planning Graph (TPG) is a Planning Graph \( G = (V, E) \) and each vertex has an associated real value. \( V_T = (V, T) \), where \( \forall t \in T \) is a real value.

The tags on the action vertices are the QoS values for executing the actions. \( cost(a) \) is a function to get the tag value for an action \( a \). And the tags of the proposition vertices are the QoS values for obtaining this proposition. \( cost(p) \) is a function to get the tag value for a proposition \( p \). For the sake of simplicity, we present our algorithm using response time as the quality criterion, i.e., the \( cost(a) \) in the Algorithm 1-4 is the QoS value of response time for service \( a \). And its calculation follows Equations 1 and 2. The calculation of the other QoS criteria is discussed in Subsection 3.5.

Algorithm 1 QoSGraphPlan is our main algorithm. It is modified from the standard Graph Plan Algorithm [10] to calculate TGP and extract a solution. QoSGraphPlan repeats ExpandGraph (Algorithm 2) until a fix point is detected. QoSGraphPlan is an anytime algorithm. When the goals \( g \) are generated, and not all of them are generated by no-op actions, it means a new solution is found. The algorithm calls ExtractPlan (Algorithm 4) to extract the plan. ExtractPlan returns a solution only when it detects the found solution is better than the best solution found so far. As time goes by, QoSGraphPlan can return better and better plans. When the fix point is reached, the algorithm 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 the cost of \( cost(a) \) (line 2). 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 are more than one parent producing the best cost-to-come value, we can choose either one, because both paths are equally the best. Lines 3 and 4 create the arcs between actions and propositions.

Algorithm 3 checks if a fixed point layer is reached.

Definition 4. A fixed point layer in a TPG is a layer k such that for ∀i, i ≥ 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, ∀(p, t) ∈ P_i, (p, t) ∈ P_k and ∀(p', t') ∈ P_k, (p', t') ∈ 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 uses U to record the QoS value for the best solution known so far. If only the new solution has a less QoS value, the algorithm extracts the new plan (lines 7-9), otherwise, it returns ∅.

Example 2. Figure 2 shows the TPG for the problem in Example 1. We can see the QoS values of D are updated during the search. And the best solution is \{w_2; w_4; w_6; w_7\}.

![Figure 2: The tagged planning graph for Example 1.](image)

### 3.3 The Properties of QoSGraphPlan

We present the properties of our QoSGraphPlan in this subsection. The proofs of the theorems are removed due to paper length limits. The proofs of Theorem 1 and 2 are similar to [3].

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

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

As we record the best parents when we expand TPG, the complexity to get a solution is just to retrieve the parents of the goals on each layer, until reaching the initial layer. Thus, QoSGraphPlan is a polynomial time algorithm. 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? One should understand that TPG just constrains the earliest starting points of the services, not the finishing time. If the service does not produce inputs for the services in the following layer, the service can last after the services in the following layer have started.

#### 3.4 Redundant Activities

In our algorithm, the optimal QoS values for the goals are calculated one by one. Different goals may have different optimal QoS values, which means the paths to produce these values are not the same. Algorithm 1-4 simply put all the paths together as a solution. This guarantees the solution produces the best QoS value. But it is possible that the solution has redundant services.

Definition 5. A plan without redundant services is a plan for which removing any service causes unsatisfied goals or worse QoS value.

Example 3. Figure 3 has four services w_1, w_2, w_3, w_4. The goals are \{d_5, d_6\}. When T=140 for w_4, the best value for d_6 is 240 which is the highest value among the goals. The 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 the QoS value for w_4 is changed to T=100, the best value for d_6 is 200. Then the QoS value for the whole plan becomes 220. And w_1 is redundant. This is because if w_1 is removed, the QoS value for the whole plan does not change.

![Figure 3: An example to explain redundant services.](image)
3.5 The Other Quality Values over TQG
For throughput (Eq. 10, 11), \( \max() \) function is used. Therefore, \( \forall p \in \text{post}(a) \), \( \text{cost}(p) = \max(\text{cost}(\text{pre}(p)), \text{cost}(a)) \). If there are multiple services that produce \( p \), \( \text{cost}(p) = \max_a(\text{cost}(\text{pre}(a)), \text{cost}(a)) \). The redundant services can be removed in the same way as described in the above subsection.

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

For execution price, successful execution rate and availability, each service contributes to the QoS value in the same way, no matter how they are connected (i.e., sequential or parallel). We are able to extend our approach for these criteria as well. Due to page limits, we do not present this part of the results in this paper.

4. EMPIRICAL RESULTS

4.1 Data set and implementation
The data set used in our evaluation is generated by the WSC test set generator [2]. The data generator generates service composition problems through generating Web services in WSDL documents and ontology concepts in OWL documents. The WSDL files are annotated with a simple extension mechanism to link to the ontology definition in OWL documents, instead of using full-fledged SAWSDL [27]. 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 generates a set of concepts and selects a subset of the concepts as the goals. Then, the generator generates 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 generates 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 service has its response time and throughput.

QoS GP (QoS aware Graph Plan) is a tool set we have implemented in Java. QoS GP includes the algorithms developed in this paper and a verification tool. We use the techniques similar to [33] to flatten the semantics and index the data. These techniques are proved to be important in speeding up the algorithms. Rather than checking the relationship map in OWL every time we need to find a list of invokable services, we build two indexing tables that are stored as hash tables so that we can look up the services or the parameters in constant time.

4.2 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 the quality 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.

In Table 2, we show whether the correct QoS values can be calculated (the checkmarks), how many services are in the solution (the lines of #Services), and how many services are redundant services (lines of #Redunt). We can see that our method can find the correct QoS values for all the five datasets, as the first place winner does. The second place winner fails to find the correct QoS for Dataset4. Our method normally produces no redundant services in the solution, or very few redundant services (lev4) on all data sets. The first place winner can generate zero redundant services in some datasets, but much more redundant services in the other datasets, especially when it computes throughput. The second place winner produces comparably more redundant services on all the datasets and on some of the datasets the number is much higher than our method and the first place winner. According to the comparison, our method can find a solution with the optimal QoS value and much fewer redundant services on all data sets.

In Table 3, we show the composition time with redundant services (Comp T1) and without redundant services (Comp T2) for both response time and throughput with our method. We can see that the computation cost of removing redundant services, i.e., T2-T1, is comparatively small compared to the time spent to find 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.

5. RELATED WORK
We study a kind of service composition problem that is to connect SOAP services into a network by matching their parameters 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) [2]. The
WSC09 data set has response time and throughput as the QoS criteria. WSC09 only evaluates the composition results by the accuracy of reporting the optimal values and satisfaction of the business goals. Our algorithm can get the right results. In addition we consider other properties, like redundancy as well.

[14], the first place winner of WSC09, uses Dijkstra’s principle to calculate the optimal value while searching all the paths. 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. Their idea is similar to our approach. However, planning graph is a much more mature 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, as we can see from their posted results in WSC09, [14]’s solution contains redundant services as well. Without using the concept of no-op, [14]’s graph loses the information about which services could have produced a proposition, which is important to remove the redundant services. The second place paper [34] 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 [15] from the authors of [14] considers a subgraph of the service connection graph could be a solution. This is an incorrect idea because firstly the service connection graph contains states unreachable 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 spaces [16].

[32] considers the hierarchical structure of composite services. Hierarchical Task Network (HTN) planning is enhanced to decompose a task in multiple ways and hence be able to find more than one plan, taking both functional and non-functional properties into account.

Another QoS related problem is the service selection problem [38, 36]. 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 combinatorial optimization problem can be modeled as a multi-dimension multi-choice 0-1 knapsack problem. Integer programming is a powerful tool to solve it [38]. As this is an NP-complete problem, heuristic search can be applied to search the problem space only partially [36, 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] calls service selection a horizontal composition problem. It models the problem as a constraint satisfaction problem (CSP).

Instead of aggregating multi-criteria QoS values into an overall score according to MCDM technique, skyline technique can be used to calculate a set of dominating services [4]. [35] considers computing service skyline from uncertain QoS values.

6. CONCLUSION

We present a QoS aware service composition algorithm using the principles of graph plan and Dijkstra’s algorithm. Our approach can satisfy the functional requirements and optimize QoS at the same time. We also discuss how to remove the redundant services from a solution. Our approach can work for different QoS criteria, though details are not fully presented in this paper.

7. ACKNOWLEDGMENTS

This work is supported by project “Building Self-Manageable Web Service Process” of Canada NSERC Discovery Grant, RGPIN/298362-2007, and by the International Science and Technology Cooperation Program of China (Grant No. 2010DFA11030), the Natural Science Foundation of China (Grant Nos. 61035003, 61021062, 60875011), and the Natural Science Foundation of Jiangsu, China (Grant No. BK2011005, BK2010054).

8. REFERENCES


"http://ws-challenge.georgetown.edu/wsc09/


