QoS-aware Service Composition and Redundant Service Removal

Min Chen

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy
Concordia University
Montréal, Québec, Canada

January 2015

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ms. Min Chen**

Entitled: **QoS-aware Service Composition and Redundant Service Removal**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this university and meets the accepted standards with respect to originality and quality.

Singed by the final examining committee:

______________________ Dr. Chun Wang	Chair
______________________ Dr. Weiming Shen	External Examiner
______________________ Dr. Abdelwahab Hamou-Lhadj	External to Program
______________________ Dr. Volker Haarslev	Examiner
______________________ Dr. Nematollaah Shiri	Examiner
______________________ Dr. Yuhong Yan	Supervisor

Approved ______________________________

Chair of Department or Graduate Program Director

____________ 20_____ ______________________________

Faculty of Engineering and Computer Science

Abstract

Automatic Service Composition (ASC) 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 business process results. It is a challenge to fulfill functional goals and achieve QoS optimization at the same time.

In this thesis, we propose to combine a planning algorithm called GraphPlan, with a systematic search algorithm like Dijkstra's algorithm to achieve functional goals and QoS optimization at the same time. The GraphPlan algorithm has the advantages of easily modeling business logic, reusing the actions in one plan, and planning parallel actions in a plan. The planning graph generated by the GraphPlan algorithm is a compact representation of all execution paths, which makes it feasible to apply Dijkstra's principle. Two methods have been proposed to combine the Graphplan with Dijkstra's algorithm.

In the first method, we extend Dijkstra's algorithm from working on a single source graph to working on the extended planning graph whose nodes have multiple sources. The advantage of this method is that it gets an optimal plan with the best QoS value for the single criteria of throughput or response time in polynomial time. However, this method does not provide a uniform graph structure (*i.e.,* an extended planning graph with single or multiple tag, to generate an optimal plan for all kinds of quality criteria).

In the second method, we improve the idea of combining the Graphplan with Dijkstra's algorithm by providing a uniform graph structure to generate a QoS optimal solution for all kinds of quality criteria. A Layered Weighted Graph (LWG) is generated and provides a uniform structure for the easy use of Dijkstra's algorithm to find an optimal plan for all kinds of quality criteria. By using multi-objective shortest path algorithms, this method can be easily extended to solve QoS optimization on multiple QoS criterion for service composition problem.

In this thesis, we also study redundant service removal to further optimize QoS optimal solutions. The removal of redundant services does not worsen the QoS value of the optimal solution. Fewer numbers of services indicates less execution costs to invoke these services. A redundant service removal problem is modeled as an optimization problem such that the optimal solution without redundancy is found.

Acknowledgments

I would like to express my deep gratitude to my supervisor Dr. Yuhong Yan for accepting me as her PhD student so that I had an opportunity to study in the Faculty of Engineering and Computer Science at Concordia University. She led me into the domain of automated reasoning. Numerous support, valuable suggestions, and patient guidance by Dr. Yuhong Yan led to my successful study at Concordia University.

I might never start my academic career if Dr. Volker Haarslev, Dr. Abdelwahab Hamou-Lhadj, and Dr. Nematollaah Shiri had not insisted on their approving of me. I feel very grateful to their valuable comments about my work.

Finally, I wish to extend my deepest appreciation to my family members. No words can express my eternal gratitude to them.

Contents

Contents **vii**

List of Algorithms **x**

List of Tables **xi**

List of Figures **xiii**

1 Introduction **1**

1.1 Motivation 2

1.2 Problem Statement 4

1.3 Thesis Overview 5

2 Background and Literature Review **7**

2.1 Background 7

2.1.1 Web Service and Service Modeling 8

2.1.2 Quality of Service 10

2.1.3 QoS-aware Service Composition 12

2.2 Literature Review 13

2.2.1 Web Service Composition Using the GraphPlan Algorithm . . 16

2.2.2 QoS-aware Service Composition Algorithms 20

2.2.3 Other Related Techniques 27

3 Anytime QoS-aware Service Composition over GraphPlan 32

3.1 Introduction 32

3.2 Motivation 34

3.3 Calculation of the QoS 35

3.4 Generation of Tagged Planning Graph 37

3.4.1 The Properties of QoSGraphPlan 42

3.5 Redundant Activities 44

3.6 Calculate the Other Single QoS Criteria 49

3.7 Under Multiple Criteria 52

3.8 BeamQoSGraphPlan with Beam Search 53

3.9 Summary of the Algorithms Developed 55

3.10 Empirical Results 57

3.10.1 Data Set 57

3.10.2 Implementation 57

3.11 Summary 63

4 QoS-aware Service Composition over GraphPlan through Graph Reachability 64

4.1 Introduction 64

4.2 The Framework 66

4.2.1 Planning Graph Labeling 66

4.2.2 Graph Conversion 68

4.2.3 Plan Generation 71

4.2.4 Main Algorithm 71

4.3 Planning Graph Labelling 71

4.4 Graph Conversion 74

4.5 Plan Generation 80

4.6 Other Single QoS Criteria 86

4.7 Experimental Results 87
4.8 Summary 89

5 Redundant Service Removal in QoS-aware Service Composition 90
5.1 Introduction 90
5.2 Motivation 91
5.3 Analysis of Redundant Service Removal 95
5.4 Model Redundant Service Removal Problem 97
5.5 Redundant Service Removal in QoS-aware Service Composition . . . 101
5.6 Experimental Results 106
5.7 Summary 108

6 Conclusion 109
6.1 Summary 109
6.2 Future Work 111

Bibliography 112

List of Publications 121

List of Algorithms

1 $Dijkstra(Graph, source)$ 29

2 $QoSGraphPlan(A, s_0, g)$ 39

3 $ExpandGraph(\langle P_0, A_1, ..., A_i, P_i\rangle)$ 39

4 $Fixedpoint(\langle P_0, A_1, ..., A_i, P_i\rangle)$ 39

5 $ExtractPlan(\langle P_0, A_1, ..., A_n, P_n\rangle, g)$ 40

6 $RemoveRedundancy(\langle P_0, A_1, ..., A_n, P_n\rangle)$ 48

7 $ExpandGraphMultiTag(\langle P_0, A_1, ..., A_i,\ P_i\rangle, g)$ 51

8 $QoSGraphPlanExt(A, s_0, g)$ 51

9 $ExpandGraphBeamWidth(\langle P_0, A_1, ..., A_i,\ P_i\rangle, g)$ 54

10 $BeamQoSGraphPlan(A, s_0, g)$ 55

11 $QoSWSC(A, s_0, g)$ 71

12 $PLPGGeneration(A, s_0, g)$ 73

13 $GraphConversion(PLPG, s_0, g)$ 76

14 $PlanGeneration(LWG, A, g)$ 83

15 $RedundancyRemoval(sol)$ 102

16 $FindReduntS(sol)$ 103

17 $ReduntSolver(sol, reduntS, Q_{opt})$ 103

18 $RemoveUselessServices(newSol)$ 104

19 $CheckInvokability(newSol)$ 104

List of Tables

2.1 A set of available services 20

2.2 Dijkstra's algorithm 30

3.1 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 42

3.2 An example of multiple criteria tag 53

3.3 Top: example of the output indexing table; bottom: example of the reverse indexing table 59

3.4 Results with the WSC09 Data Sets: our method/Paper [32]/Paper [74] 59

3.5 Our composition time: T1 with redundant services (Resp. Time/Throughput); T2 without redundant services (Resp. Time/Throughput). . . . 61

3.6 Total execution price on the WSC09 Data Sets 61

3.7 Results considering aggregated value of execution price and reputation on the WSC09 Data Sets 62

4.1 A set of available services 73

4.3 Changes made in Algorithm 14 for different single criteria (r,r' are real values, $k, k' \geq 0$ are integers) 88

4.2 Results with the WSC09 Data Sets: 89

5.1 Services in the solution shown in Figure 5.2 92

5.2 Response time before and after redundancy removal for Example 19 . 98
5.3 Redundancy removal for Example 11 106
5.4 Redundancy removal results for the solutions with optimal response time: our method/random removal method in Section 3.5 107
5.5 Redundancy removal results for the solutions with optimal throughput: our method /random removal method in Section 3.5 107

List of Figures

2.1 Web service roles and operations [2] 9

2.2 The planning graph for Example 1. 20

2.3 System architecture [74] 22

2.4 Algorithm overview: 3 steps search [74] 24

2.5 Definition of variables and parameters [16] 25

2.6 The graph for Example 2. 29

3.1 The tagged planning graph for Example 1. 41

3.2 An example to explain the scheduling of services over TPG 44

3.3 An example to explain redundant services 45

3.4 An example to explain redundant services 47

3.5 An example to explain multiple tags. 50

3.6 The multiple tags used for optimizing execution cost. 52

3.7 Semantic relationship between Web service input/output parameters . 58

4.1 The framework of our approach 67

4.2 An example of PLPG 68

4.3 The LWG converted from the PLPG in Figure 4.2 70

4.4 The partially labelled planning graph for Example 11. 74

4.5 The Layered Weighted Graph (LWG) for Example 12. 81

4.6 Plan Generation for Example 12. 85

5.1 The tagged planning graph for **Example 14** 93

5.2 The TPG for the solution for **Example 14** 93

5.3 An EDAG with labelled levels for Example 11 96

5.4 The EDAG after w_5 is removed from Fig. 5.3 97

Chapter 1

Introduction

A Web service is a self-described, self-contained software module that completes tasks, solves problems, or conducts transactions on behalf of a user or application. Available Web services are posted across the Internet using a set of open standards such as WSDL (*Web Service Description Language*) [67], SOAP (*Simple Object Access Protocol*) [66], and UDDI (*Universal Description Discovery and Integration of Web Services*) [51]. With these open standards, Web services are automatically invokable and interoperable. An increasing number of companies and organizations implement their core business and outsource their application services over the Internet. However, facing large and various users' requests, it is impossible to fulfill every request by a single existing service. For example, a user wants to make a travel plan by booking a flight, a hotel room, and a taxi from the airport to the hotel in the destination city.

Automatic Service Composition (ASC) is the generation of a business process to fulfill business goals that cannot be fulfilled by individual services. A business process is implemented as a composite service that typically assembles and invokes many pre-existing services (*e.g.,* travel-booking service is a composite service which invokes two individual services, *i.e.,* a flight booking service and a hotel booking service). The travel-booking service is able to complete the business task (*e.g.,* making a travel plan, which is impossible for an individual service like the flight booking service or the hotel booking service).

In addition to the fulfilled functional requirements, users are also concerned about how well non-functional requirements are fulfilled. The performance of services is reflected by the non-functional properties of services (*i.e.,* Quality of Services (QoS)). QoS is specified in the service description using the standard language (*e.g.,* Web Service Level Agreement (WSLA) [36]). To quantitatively measure qualities of services, several related aspects are often considered, such as response time, throughput, and execution price. Achieving QoS optimization during ASC is called QoS-aware service composition.

We study how to fulfill the functional goals and achieve QoS optimization simultaneously in QoS-aware service composition.

1.1 Motivation

Automated Service Composition (ASC) is studied under different assumptions [54, 59]. The most useful and practical problem is to connect stateless 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 thesis.

AI Planning algorithms are frequently used to solve ASC problems [56, 57, 81]. First, AI planning algorithms build a unique problem space where connections between actions are propositions and expressed in a compact way. 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 path. If the planning algorithms stop at a later time after the first feasible solution is found, it is very possible to find other solutions corresponding to longer paths.

In addition to satisfying business goals, recent research moves towards 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 vary. Therefore, we need to modify the classic planning algorithm to become QoS aware to find a QoS optimized solution.

People have used algorithms like Dynamic Programming (DP) and Integer Programming (IP) to find a QoS optimized solution [16, 32, 34]. However, Dynamic Programming (DP) and Integer Programming (IP) are based on given linearized formulations. Formulizing the business constraints into linear algebra is not straightforward. It involves the proper assignment of variables. Some adjustments may be needed to make the relationship among variables linear. Hence, it is not straightforward to model service composition problems into DP or IP. Some of the models (*e.g.,* [34]) does not consider the possibility of reusing the same service in a plan.

People 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; services are selected for each task in the control flow in the second step. This second step is the so-called service selection problem which is NP-complete [78, 80]. This kind of problem makes sense only if we have to use a predefined control flow. Otherwise, we should decide control flow and services selection at the same time in QoS-aware service compositions. This is because it is possible that some other control flows with some other services can have a better QoS than the predefined control flow can achieve.

Our motivation is to keep the advantage of the AI planning model, such as easily modeling business logic, reusing the actions in one plan, planning parallel actions in a plan, and extending AI planning from the ASC solver into the QoS-aware service composition solver with the help of an optimization algorithm for the QoS. We are motivated by combining a AI planning technique, called the Graphplan, with the Dijkstra's algorithm. From the optimization perspective, Dijkstra's algorithm is an optimization algorithm since it traverses a graph to find the optimal paths (*i.e.,* the shortest paths) from a single source to the destination vertices. If we map the QoS value into the distance, we can use Dijkstra's algorithm to obtain the shortest path that corresponds to a path with the optimal QoS value in terms of certain QoS criteria.

1.2 Problem Statement

According to the type of information that can be used in service composition, Web service composition problems can be viewed from three different levels [37], a signature level, a behavior description level, and a QoS description level.

In a signature level, WSDL is the standard used to describe Web services. A service signature is made up of a set of operations that can be described in terms of their inputs and outputs. Since a service in a signature is taken as a black box with multiple inputs and multiple outputs, this type of service can be regarded as a stateless service. Given a set of initial input parameters and a set of stateless services, a Web service composition problem generates a sequence of sets of services to produce the expected output parameters. Moreover, this type of service composition is based on the assumption that each component service is an atomic service that only contains one operation. If a service has more than one operation, each operation is regarded as an atomic service to take part in service composition.

On a behavior description level, services consist of stateful business processes, which specify multi-phase interactions/conversations with other services. Conversational services can be expressed using the standard behavioral description languages, such as *Web Service Business Process Execution Language* (WS-BPEL) [15] and *Web Ontology Language for Services* (OWL-S) [63]. The behavioral description language uses constructs (*e.g.,* sequential, conditional and flow) to describe the business process of conversational service. Given a set of behavioral descriptions of Web services and goal descriptions, the Web service composition problem on a behavior description level automatically generates a business to fulfill underspecified requirement, such as data requirements or capability requirements. Data requirements specify the expected data produced by a composite service. Capability requirements give a set of capabilities that are specified in the expected order to be fulfilled.

On a QoS description level, the non-functional property (*i.e.,* Quality of Services (QoS)) is specified in the service description using standard language such as *Web Service Level Agreement* (WSLA). QoS has several criteria including response time, throughput, and execution price.

In this thesis, our work is done from a signature level and a QoS description level. We use the planning technique called GraphPlan to study QoS-aware service composition so that functional goals and QoS optimization can be achieved simultaneously. Graphplan can generate a plan with sequential and parallel actions. As for stateful business processes at a behavior description level, conditional planning (to include conditional constructs and loop constructs in a plan) is a special topic to study in planning domain. [57] 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 this encoding approach, the planning graph can also handle the exclusive choice pattern. Though the planning graph does not have a loop construct, actions are re-used when the planning graph is constructed. The repeated sequential actions can be considered as an unfolded loop.

1.3 Thesis Overview

The rest of this thesis is organized as follows.

- Chapter 2 provides the background and literature review. This chapter introduces the necessary background for the rest of the thesis. In particular, we review Web service composition using the GraphPlan algorithm, QoS-aware service composition algorithms, and related techniques that will be used in our research.

- We study the QoS-aware service composition problem in Chapter 3 and Chapter 4. We are motivated to combine the GraphPlan technique with Dijkstra's algorithm to solve QoS-aware service composition problems. Two algorithms are respectively proposed in Chapter 3 and Chapter 4. Experimental results show both algorithms can find a solution with optimal QoS values.

- Chapter 5 studies the redundant service removal problem. We model the redundant service removal problem as an optimization problem such that the optimal solution without redundancy is found.

- Chapter 6 concludes the thesis with the summary and the discussion of the future work.

Chapter 2

Background and Literature Review

2.1 Background

Service-Oriented Computing (SOC) is an promising paradigm that refers to computing in Service-Oriented Architecture (SOA)[19, 20, 21, 49] which utilizes Web services with standard interfaces as the constructs to build rapid and distributed software applications. Automated Service Composition (ASC) which composes the existing services to fulfill some complex functionality is one of the most studied topic in SOC. The most useful and practical problem is to connect stateless SOAP services into a network by matching their parameters so that given a set of input parameters this network of services can produce a set of required output parameters. This is also the composition problem studied in this thesis. Quality of Service (QoS) is the non-functional properties of services. When discussing service composition technologies in SOC, QoS is another important factor which influences the process of service composition, as QoS is a significant concern for users. QoS-aware service composition targets satisfying not only functional goals but also QoS requirements such as the optimization of QoS values in terms of certain QoS criteria. In this thesis, our research focuses on the studying QoS-aware service composition problem.

2.1.1 Web Service and Service Modeling

The term “Web service” has been defined in different ways. Papazoglou [53] describes that a Web service is a self-describing, self-contained software module available via a network, such as the Internet, which completes tasks, solves problems, or conducts transactions on behalf of a user or an application. Another widely accepted definition is specified by the World Wide Web Consortium (W3C) that a Web service is defined as “a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols” [2].

The definition of Web service also reveals the work mechanism of Web services. Web services work in the service-oriented architecture (SOA). The overview of SOA is shown in Figure 2.1 [2]. There are three primary roles of SOA that communicate with each other through three operations. The three primary roles are the service provider, the service registry, and the service requester (client) and the three operations are publish, find, and bind.

- The service provider is the owner of Web services and implements the functionality of the service. The service provider is responsible for the publishing of Web services on the service registry, including describing the information of services and registering that information with the service registry hosted by a service discovery agency.

- The service requester is the client who sends the request regarding the completion of certain functions to the service registry, searching for the services that satisfy the requested functions, and subsequently invoking the services.

- The service registry is a directory of service descriptions. The service requester uses the service registry to locate the service and obtain information for services.

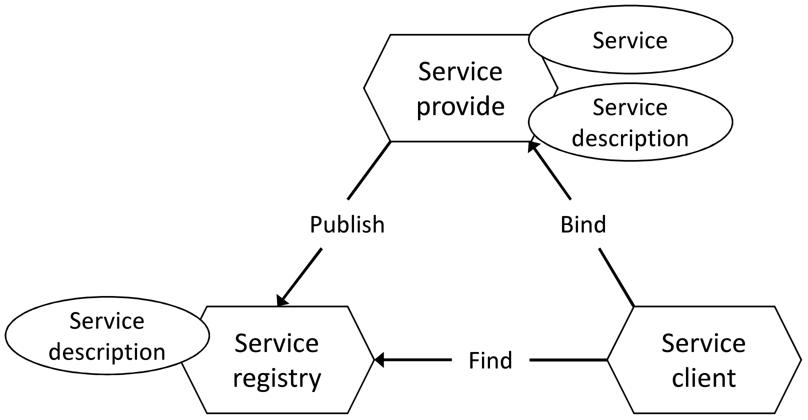


Figure 2.1: Web service roles and operations [2]

This work mechanism of Web services in Figure 2.1 [2] is enabled by a number of Web services supported by a set of open standards. These open standards include the Web Service Description Language (WSDL) [67], the Simple Object Access Protocol (SOAP) [66], and the Universal Description Discovery and Integration of Web services (UDDI) [51]. With these open standards, Web services are automatically invokable and interoperable.

Service modeling relies on many factors [17]. In this thesis, we consider the services as stateless black boxes (no conversations). Conversations describe two-way communication between a client and a Web service so as to maintain the state between calls to the Web service. The services expose themselves in WSDL descriptions that do not include state information. We can associate semantic information to inputs and outputs using SAWSDL [65] and ontology supporting annotations described using OWL [64].

Definition 1 *Given a set D of concepts, service w is a tuple $(in(w), out(w))$, where $in(w) \subseteq D$ (resp. $out(w) \subseteq D$) denotes the inputs (resp. the outputs) of w.*

For each service w with n operations $o_1, \ldots, o_n$ we may operate or function as if we had n services $w{:}o_1$, …, $w{:}o_n$. Currently, most services posted online are this kind of service, that is, numerous services listed by `webservicelist.com` and `xmethods.net`. Their functions span from checking stock prices, weather, driving directions, to calculating currency exchange rates, or mortgages.

An output of one service could become an input of another service if they are exactly the same concept or if the input concept subsumes the output concept. For example, assume one service needs "vehicle" as one of its inputs. If another service outputs "vehicle" or "car", the two services can be connected via their compatible parameters.

2.1.2 Quality of Service

Quality of Service (QoS) is the non-functional property of a service. Normally, QoS is specified in the service description using the standard language, such as a Web Service Level Agreement (WSLA) [36]. If the QoS is given, the service model used in our research corresponds to Definition 2.

Definition 2 *Given a set D of concepts, a **service** w is a tuple $(in(w), out(w), Q(w))$, where $in(w) \subseteq D$ (resp. $out(w) \subseteq D$) denotes 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 operate as if we had n services $w : o_1, \ldots, w : o_n$. We use $\sigma = w_1, w_2, \ldots, w_n$ to represent a network of connected services. If they are connected in sequence, $\sigma = w_1; w_2; \ldots; w_n$, or in parallel, $\sigma = w_1||w_2|| \ldots ||w_n$.

Some commonly used quality criteria for a Web service w and their aggregated values over σ are listed as below [7, 28]:

- **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.

$$Q_1(w_1; \ldots; w_n) = \sum Q_1(w_i) \tag{2.1}$$

$$Q_1(w_1||\ldots||w_n) = \max Q_1(w_i) \tag{2.2}$$

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

$$Q_2(w_1;\ldots;w_n) = \min Q_2(w_i) \tag{2.3}$$

$$Q_2(w_1||\ldots||w_n) = \min Q_2(w_i) \tag{2.4}$$

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

$$Q_3(w_1,\ldots,w_n) = \sum Q_3(w_i) \tag{2.5}$$

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

$$Q_4(w_1,\ldots,w_n) = \frac{1}{n}\sum Q_4(w_i) \tag{2.6}$$

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

$$Q_5(w_1,\ldots,w_n) = \prod Q_5(w_i) \tag{2.7}$$

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

$$Q_6(w_1,\ldots,w_n) = \prod Q_6(w_i) \tag{2.8}$$

Please note that some of the above QoS criteria are negative, such as the higher the value, the lower the quality. Response time and execution price are in this category. The other criteria are positive, such as the higher the value, the higher the quality. Throughput, reputation, successful execution rate, and availability are in this category. When considering QoS-aware service composition for multiple QoS criteria, people normally apply the Multiple Criteria Decision Making (MCDM) technique [61] to solve this problem. A Simple Additive Weighting (SAW) [41] is a common method used to aggregate multiple QoS values into a simple overal QoS value. There are basically two phases to use SAW. The first phase is called the scaling phase. Let the utility value $U_i(w_j)$ be the scaled value of a quality i for a service w_j. For negative criteria, values are scaled according to Equation 2.9. For positive criteria, values are scaled according to Equation 2.10.

$$U_i(w_j) = \begin{cases} \frac{Q_i(w_j) - Q_i^{min}}{Q_i^{max} - Q_i^{min}} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{2.9}$$

$$U_i(w_j) = \begin{cases} \frac{Q_i^{max} - Q_i(w_j)}{Q_i^{max} - Q_i^{min}} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{2.10}$$

The second phase is called weighting phase. The overall quality score for a Web service w_j is defined in Equation 2.11:

$$U(w_j) = \sum U_i(w_j) \times W_i \tag{2.11}$$

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

2.1.3 QoS-aware Service Composition

Nowadays, an increasing number of companies and organizations implement their core businesses and outsource their application services over the Internet. Sometimes, no single service can satisfy the functionality requested to accomplish a specific business goal. In such cases, a composite service which typically assembles and invokes many pre-existing services to complete the business goal is necessary.

Automatic service composition is the generation of a business process to fulfill business goals that cannot be fulfilled by individual services. Service composition is defined as follows.

Definition 3 *A service composition problem is a tuple* $(W, D_{\text{in}}, D_{\text{out}})$*, where* W *is a set of services,* D_{in} *are the provided inputs, and* D_{out} *are the expected outputs.*

QoS-aware service is the generation of a business process to both fulfill business goals and achieve QoS optimization. In this thesis, we focus on the QoS-aware service composition problem, as defined in Definition 4, where functional goals and QoS optimization can be achieved at the same time.

Definition 4 ***A QoS-aware 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.*

2.2 Literature Review

Some existing work decomposes QoS-aware service composition into two sub-problems, for example the service composition problem and QoS optimization problem. Since functional goals need to be satisfied first, service composition needs to be performed preceding QoS optimization. In such a case, the study of service composition and QoS optimization become separate research branches, for example with automatic service composition and service selection problems.

Automatic service composition has been widely studied [3, 18, 23, 33, 75]. Most research is based on workflow techniques [12, 58, 60, 68] or AI planning techniques [45, 46, 47, 48, 71, 81]. For the former, a composite service is regarded as a workflow [13]. The definition of a composite service includes specifying the control flow and data flow among atomic services. For the latter, each Web service can be specified by its preconditions and effects in the planning context. The workflow methods are mostly used when the request has given the defined the process model and look for the atomic services to implement the process model. AI planning methods are used when the request has no process model but does has a set of expected outputs, constraints, or preferences. In recent years, service composition has been studied in cloud computing environments [5, 26, 27, 35] with the rapid development of cloud computing [4, 62].

A service selection problem [78, 80] 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 [80]. As this is an NP-complete problem, a heuristic search can be applied to search the problem space only partially [6, 78]. A Genetic Algorithm (GA) is another way to partially search the problem space [11]. The advantage of GA compared to integer programming is that the GA can deal with nonlinear constraints of QoS requirements.

A service selection problem is also called a horizontal composition problem (versus vertical composition)[29]. It models the problem as a constraint satisfaction problem (CSP). Rooted on AI, a CSP can model hard constraints, for example the functional requirements in ASC, and soft constraints, for example optimizing QoS in ASC. The penalty of violating a hard constraint is infinite, versus a finite penalty to sub-optimal QoS values. Therefore, the CSP algorithms search for a solution which minimizes the penalty. Hassine et al. [29] proposes an interactive algorithm for solving the problem with user inputs. Their model can be used to solve the vertical composition problem as well. Nguyen et al. [50] use fuzzy constraints to model service clients and providers' preferences. Services can communicate with each other to find an optimal solution in a distributed manner. In [72], a model for QoS-satisfied predictions is proposed based on the hidden Markov model (HMM) [43].

Instead of aggregating multi-criteria QoS values into an overall score according to an MCDM [39, 44], a skyline technique can be used to calculate a set of dominating services [9]. Yu et al. [76] considers computing a service skyline from uncertain QoS values. Qi et al. [77] generates a service skyline composed of a specific set of services that other possible service sets cannot dominate with respect to the QoS parameters.

In our work, we study the QoS-aware service composition problem where QoS optimization and functional goals can be achieved at the same time. The Web Service Challenge [7] targets this type of QoS-aware service composition problem, such as the approaches proposed by the first place winning paper [32] and the second place winning paper [74] of WSC09 [7]. However, both approaches generate more redundant services. A later paper [34] from the authors of [32] considers that a subgraph of the service connection graph could be a solution. This is an incorrect idea because the service connection graph contains states unreachable from the initial state that do not need to be constructed. It also removes the possibility of reusing actions in a plan. A heuristic search is also a commonly used approach for large problem spaces [38].

Another QoS-aware service composition related problem is redundant service removal. In most cases, there are redundant services that can be removed from the optimal solution generated by QoS-aware service composition approach. The optimal solution after redundancy removal will have less execution costs. Redundant service removal problems have been addressed in ASC. Kwon et al. [40] proposes a non-redundant Web services composition based on a two-phase algorithm. Hewett et al. [30] also mentions redundant service removal in the composition of Web services. However, as far as we know, almost all previous work on redundant service removal problem is studied in the context of ASC rather than QoS-aware service composition. It is difficult to study redundant service removal problem in QoS-aware service composition due to the functional overlapping of the chosen services. In our work, redundancy removal problems are studied in the context of QoS-aware service composition to achieve bi-objectives: functional goals and QoS optimization.

In the following, we will review the techniques in the related research area. Web service composition using the GraphPlan algorithm is presented in Section 2.2.1. In Section 2.2.2, we review QoS-aware service composition algorithms. Finally, other related techniques are presented in Section 2.2.3.

2.2.1 Web Service Composition Using the GraphPlan Algorithm

AI planning [24] has been applied with success to service composition [14, 55], because a service composition problem can be mapped onto an AI planning problem. Following [81], it is possible to map a service composition problem (W, D_{in}, D_{out}) 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 [52]. Additionally, planning graphs enable the retrieval of plans with parallel invocations. These can be encoded using parallel operations (WS-BPEL flow).

Definition 5 *Given a finite set $L = \{p_1, \ldots, p_n\}$ of proposition symbols, a planning problem [24] 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 tuple $(pre, effects^{+})$ 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 $s \in S$ 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 5, $pre(a)$ is the set of the propositions as the precondition of action a. The definition in [24] takes into account predicates and constant symbols which are then used to define states (ground atoms made with predicates and constants). A predicate can be regarded as a Boolean-valued function on a subject which has the value of true or false. A proposition is understood as a statement which affirms or denies a predicate of a subject. We directly use propositions here because in the Web service models we do not have predicates. In [24], 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 to get plans from them such as depending on whether they are building the underlying graph structure in a forward (from the initial state) or backward (from the goal) approach [24]. The study in this thesis is based on an AI planning algorithm called GraphPlan [8]. Recent works have demonstrated the suitability of this model for ASC [57, 81]. GraphPlan 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. 2.2). 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 in parallel. The planning graph actually explores multiple search paths at the same time while expanding the graph. The construction of the planning graph stops at a layer P_k if the goal is reached ($g \subseteq P_k$) or in case of a fixed point layer ($P_k = P_{k-1}$). A fixed point layer in a planning graph G is a layer k such that $P_k = P_{k-1}$ and $A_k = A_{k-1}$. In the former case there exists at least a solution, while in the latter there is not. Solution(s) can be obtained using a backward search from the goal. In GraphPlan, the solution is layered as defined in Definition 6.

Definition 6 *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. For example the effects of an action should not be the precondition of another action. Finally, there is no loop in a plan.

The GraphPlan approach contains two phases.

1. 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 in the size of the planning problem [8]. If the goal is reached, the problem has a solution. We come to the conclusion that the existence of a solution is detected in polynomial time.

2. The second phase is to extract a solution using a backward search from the goal layer. Normally the second phase is more costly. In the most general cases (*i.e.,* if the problem has negative effects), the backward search phase may require backtracking and the complexity is NP-complete. If the problem only has positive effects such as a service composition problem modeled as graph planning problem, we see that backtracking is not needed [31]. Since negative effects cause the existence of mutually exclusive for a pair of actions or propositions. No negative effects means no exclusions exist, and that backtracking

does not happen. During the backward search, we want to find a solution that consists of a minimal set of services. Once the planning graph is constructed, the length of the solution is known (*i.e.,* the number of action layers in the planning graph minus one). We can use a minimal hitting set algorithm [25] to obtain the minimal number of services in each layer of solution. A hitting set for a collection of sets C is a set $H \subseteq \cup_{s \in C} S$ such that $H \cap S \neq \{\}$ for each $S \in C$. A hitting set problem is known as a NP-complete problem. What we need is the hitting set with the smallest cardinality: a hitting set for C is minimal if and only if no proper subset of it is a hitting set for C.

Following [81], it is possible to map a service composition problem (W, D_{in}, D_{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 [52]). 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 are listed in Table 2.1 (modified from [34]). The composition query is* $(D_{in}, D_{out}) = (\{A, B, C\}, \{D\})$*. We use the GraphPlan approach to solve this service composition problem. According to* D_{in}*,* D_{out} *and the available services in Table 2.1, we construct a planning graph as shown in Figure 2.2.*

In Figure 2.2, 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 2.2). An action a at layer A_i *takes the incoming arcs originating from its inputs at* P_{i-1} *and connects to its*

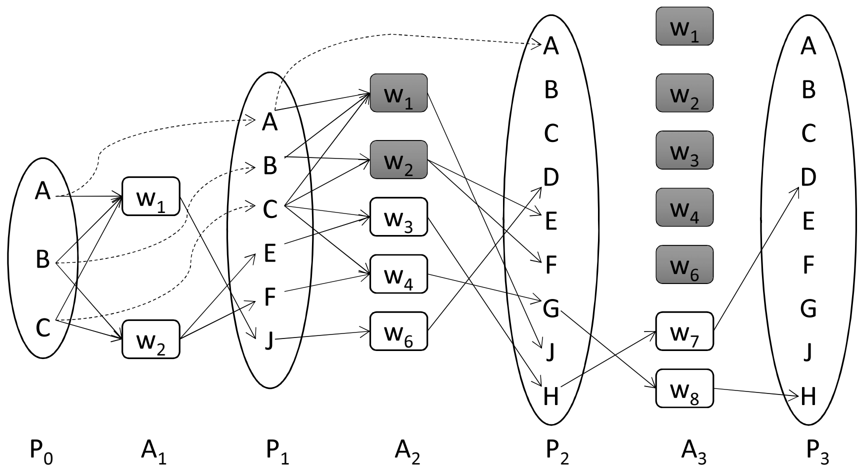


Figure 2.2: The planning graph for Example 1.

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 achieves the goal D at layer P_3. After the planning graph achieves the goal, we extract three solutions starting from goal D at P_3: $\{w_1; w_6\}$, $\{w_2; w_3; w_7\}$.

Table 2.1: A set of available services

w_i	inputs	outputs	w_i	inputs	outputs
w_1	A, B, C	J	w_5	K	H
w_2	B, C	E, F	w_6	J	D
w_3	C, E	H	w_7	H	D
w_4	C, F	G	w_8	G	H

2.2.2 QoS-aware Service Composition Algorithms

Several algorithms have been proposed to solve the QoS-aware service composition problem to fulfill functional goals and achieve QoS optimization at the same time. We describe them below.

Effective Pruning Algorithm

The first place winning paper [74] of Web Service Challenge 2009 (WSC09) [7] proposed an effective pruning algorithm to for QoS-aware service composition to achieve functional goals and QoS optimization at the same time. Figure 2.3 presents the system architecture of this algorithm. The solid arrows shows the workflow of initialization. The initialization process (a) takes the input of the system including the service description (WSDL file), the semantic annotation (OWL file), and QoS description (WSLA file) to build the concept forest (b) and the service repository (c). The concept forest and the service repository are used to build an inverted index table [1] as the fundamental data structures. Challenge client sends the request to the system (e1) and the request is forward to the filter (e2) that generates a sub-graph. Making use of multiple threads techniques, two threads are used to search the sub-graph to find the composition results satisfying the highest throughput and the lowest response time separately (g). The composition results are sent back to the challenge client (i).

In the system, the filter plays an important role in the system. The filter runs a filter algorithm followed by a modified dynamic programing. The filter algorithm generates a layered service graph. The layered service graph is a network of services where the network is expanded layer by layer according to the matching relation between input concepts and output concepts of services. When there is a layer of services whose output concepts contain all the expected output concepts, the sub-graph stops at this layer. In such a way, the filter algorithm filters out the rest of services which are not necessary to be added into the layered service graph. However, we found that the filter algorithm does not consider the possibilities of reusing services when building the layered service graph. Therefore, the effective pruning algorithm [74] may not find the optimal solution.

When constructing the layered service graph, modified dynamic programming is proposed to find out the optimal throughput and response time for each concept and

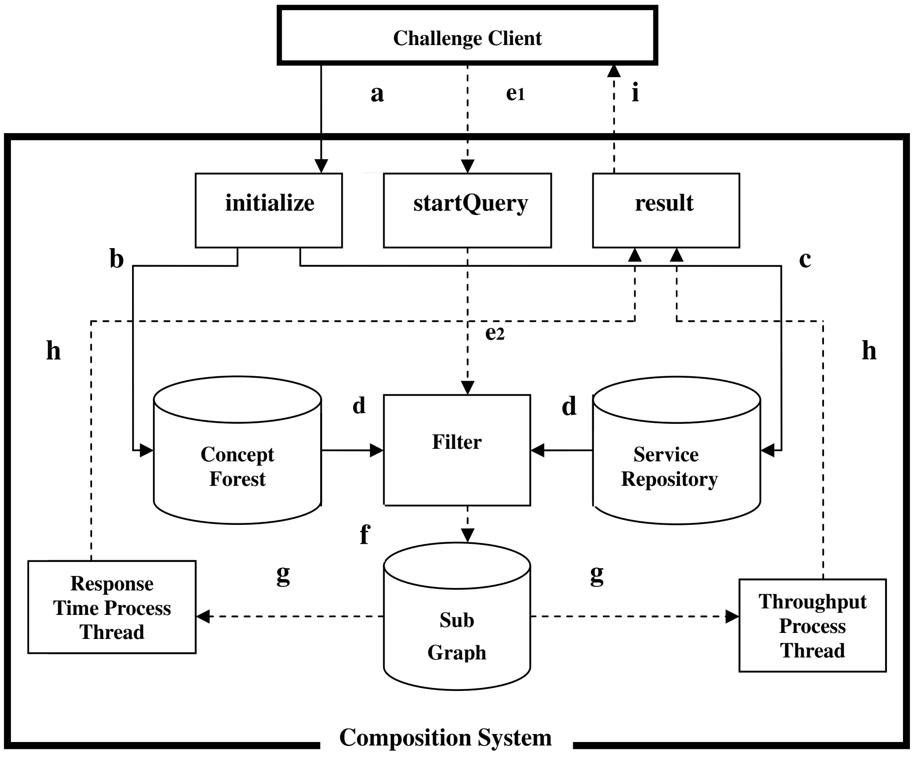


Figure 2.3: System architecture [74]

each service respectively through a concept map. In such a way, an inverted index table mapping each concept with two services that provide this concept with the optimal throughput and the optimal response time respectively is constructed. The backward search makes use of the inverted index table to search the optimal plan backward from the requested service to the service in layer 1. The principle of the backward search is that it will always find an optimal service at layer k $(1 \leq k < i)$ for a concept as an input of a service at layer i. However, it causes the optimal plan to contain more redundant services. A plan with an optimal QoS value does not mean any sub-path of this plan needs to be optimal. More services indicate more execution costs.

A QoS-driven Approach

The second place winning paper [74] of WSC09 [7] proposed a QoS-driven approach for QoS-aware service composition. Figure 2.4 shows the overview of the algorithm. The algorithm divides the composition task into three steps, breadth-first search, semantic optimization, and QoS optimization.

During the breadth-first search, the algorithm first creates pairs of services {P,S} where service P can directly invoke service S. Then the algorithm scans the set of service pairs to generate a graph represented by a sequence of layers. Starting from the provided parameters by the request, services are added into the first layer. The output parameters of services in the current layers plus the available parameters invoke services in the next layer. The breadth-first search continues this process until all the available parameters contain the required parameters.

The second step of semantic optimization is to decrease the number of services. The removal of redundant services to maintain the minimum number of services can still produce the requested output parameters.

The last step of QoS optimization is to select services from many candidate services, since the same functionality can be provided by different service characterized with various QoS.

We find the QoS-driven approach has two shortcomings. Firstly, it is doubtful that the QoS-driven approach can find the optimal solution. In the layered graph, a service does not belong to the first $k-1$ layers if this service belongs to the k^{th} layer. It eliminates the possibility of re-using services in a solution, which may find a non-optimal solution. Secondly, the solution found by the QoS-driven approach may contain more redundant services. In the third step of the QoS optimization, if a parameter can be provided by several services, the search selects a service with the optimal QoS value that provides this parameter. Since several parameters can be produced by the same service and several services may have the same QoS values, the QoS optimization may find an optimal solution containing some redundant services.

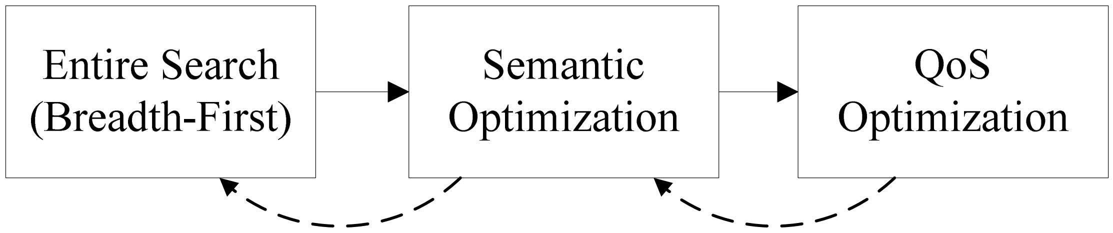


Figure 2.4: Algorithm overview: 3 steps search [74]

After the redundant services are removed, the still functional solution works and has the optimal QoS value.

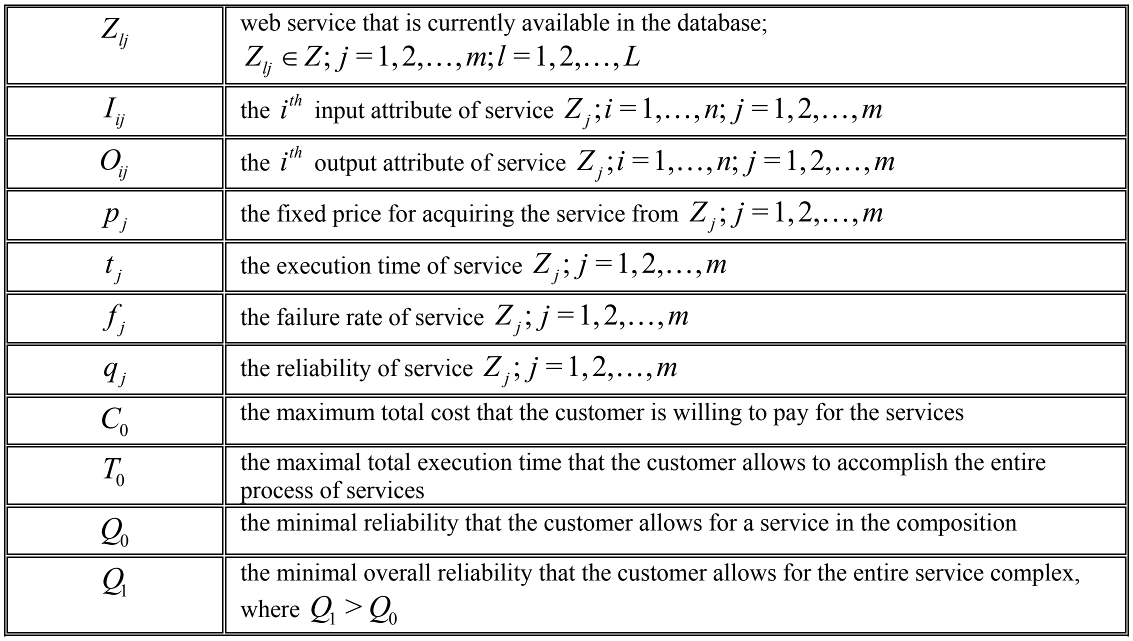
Goal Programming Approach

Cui et al. [16] analyzes scenarios of Web service composition and models QoS-aware service composition as a goal programming problem. Cui et al. [16] targets at solving QoS-aware service composition in accordance with customers' preferences.

In order to build the mathematical modeling, Cui et al. [16] first defined the parameters and variables to be used in the model. Z, I, and O are the variables representing a Web service, an input attribute, and an output attribute respectively. m is the number of services. n is the number of attributes for each service where $n = \max\{|I|, |O|\}$. L is the maximal number of composition levels. Z_{ij} means the status of the j^{th} Web service in the i^{th} level of the composition where $j = 1, \ldots, m$ and $i = 1, \ldots, L$. The other parameters and variables are defined as shown in Figure 2.5. Among all the variables, Z_{ij} is the decision variable. If $Z_{ij} = 1$, Web service Z_j is selected in the i_{th} level. Otherwise, $Z_{ij} = 0$.

Based on the defined variables and parameters, a set of objective functions is defined. The objective functions are defined in terms of the optimized objectives. If considering the execution price, the objective is to minimize the overall price of the services in the composition.

$$\min \sum_{l=1}^{L} \sum_{j=1}^{m} Z_{lj} \cdot p_j$$



Z_{lj}	web service that is currently available in the database; $Z_{lj} \in Z; j = 1,2,\ldots,m; l = 1,2,\ldots,L$
I_{ij}	the i^{th} input attribute of service $Z_j; i = 1,\ldots,n; j = 1,2,\ldots,m$
O_{ij}	the i^{th} output attribute of service $Z_j; i = 1,\ldots,n; j = 1,2,\ldots,m$
p_j	the fixed price for acquiring the service from $Z_j; j = 1,2,\ldots,m$
t_j	the execution time of service $Z_j; j = 1,2,\ldots,m$
f_j	the failure rate of service $Z_j; j = 1,2,\ldots,m$
q_j	the reliability of service $Z_j; j = 1,2,\ldots,m$
C_0	the maximum total cost that the customer is willing to pay for the services
T_0	the maximal total execution time that the customer allows to accomplish the entire process of services
Q_0	the minimal reliability that the customer allows for a service in the composition
Q_1	the minimal overall reliability that the customer allows for the entire service complex, where $Q_1 > Q_0$

Figure 2.5: Definition of variables and parameters [16]

If the execution time is under consideration, the objective is to minimize the total process time for executing the entire series of services. Since services at the same level are assumed to be executed in parallel, the execution time η_l of the services at l^{th} level is the maximum service execution time of the l^{th} level, *i.e.,* $\eta_l = \max_j\{t_j \cdot Z_{lj}\}$. Therefore, the total execution time of services in the composition is $\sum_{l=1}^{L} \max_j\{t_j \cdot Z_{lj}\}$. The total execution time can be reformulated into the following linear function:

$$\min \sum_{l=1}^{L} \eta_l$$

subject to

$$\begin{aligned} \eta_l - t_j \cdot Z_{lj} &\geq 0 \\ j = 1,\ldots,m; l &= 1,\ldots,L \end{aligned}$$

If reliability is under consideration, the reliability of the service composition is the summation of the reliability of services included in the composition. The highest reliability is the objective to be optimized. Since maximizing the overall reliability corresponds to minimizing the product of the failure rates of the services in the composition, the objective function to optimize the reliability is:

$$\min \prod_{Z_j \in S} f_j$$

where $\prod_{Z_j \in S} f_j$ is the overall failure rate of the services in the composition.

Also, a set of constraints are defined based on the defined variables and parameters. The hard constraints including input constraints, output constraints, and the relationship of the outputs and the inputs between the levels are the functional constraints that the service composition must satisfy.

Input constraint specifies that the inputs of the compositions should be included in the inputs provided by the composition.

$$\sum_{l=1}^{L} \sum_{j=1}^{m} I_{ij} \cdot Z_{lj} \geq I_{i0} \qquad i = 1, 2, \ldots, n.$$

Output constraint require that the outputs of the services in the composition should contain all the expected output parameters.

$$\sum_{l=1}^{L} \sum_{j=1}^{m} O_{ij} \cdot Z_{lj} \geq O_{i0} - I_{i0} \qquad i = 1, 2, \ldots, n.$$

All the inputs of the services in the first level must be contained in the initial inputs.

$$\sum_{j=1}^{m} I_{ij} \cdot Z_{1j} \leq I_{i0} \qquad i = 1, 2, \ldots, n.$$

Moreover, all the inputs of services at k^{th} level must be contained in the initial inputs and the outputs produced by services in previous levels.

$$\sum_{j=1}^{m} I_{ij} \cdot Z_{k+1,j} - \sum_{l=1}^{k}\sum_{j=1}^{m} O_{ij} \cdot Z_{lj} \leq I_{i0}$$
$$k = 1, 2, \ldots, L-1; i = 1, 2, \ldots, n.$$

Other constrains are soft constraints. Normally, soft constraints are the requirements regarding QoS. For example, the constraint on the total cost should not exceed a certain value C_0.

$$\sum_{l=1}^{L}\sum I_{ij} \cdot Z_{1j} \leq C_0$$

In terms of the objective functions and constraints, [16] formulates three multi-criteria scenarios according to the customers requirements.

2.2.3 Other Related Techniques

Dijkstra's Algorithm

Dijkstra's algorithm's goal is to find the shortest paths from a single source in a graph [42]. Dijkstra's algorithm is a systematic search algorithm. If the graph is finite, a 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, the 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. We are dealing with a finite graph in this thesis.

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, 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 [22].

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. Algorithm 1 is the pseudo code of Djikstra's algorithm [70].

Example 2 *Figure 2.6 is a graph with the arcs labeled with their lengths. Node A is the source vertex. Table 2.2 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 Table 2.2, $d(.)$ represents the distance of the node and $p(.)$ represents the parent of the node. The numbers in the first column of Table 2.2 show the number of steps to perform Dijkstra's algorithm. Once we successfully find the shortest path from the origin A to a node, the node is added into "Solved nodes" in Table 2.2. For example, at step "5", "Solved nodes" is $ABCFD$ because nodes A, B, C, F and D have found*

Algorithm 1: $Dijkstra(Graph, source)$

```
1: $dist[source] := 0$;// Distance from source to source
2: for each vertex $v$ in $Graph$ do
3:    if $v \neq source$ then
4:       $dist[v] := infinity$;// Unknown distance function from source to v
5:       $previous[v] := undefined$;// Previous node in optimal path from source
6:    end if
7:    add $v$ to $Q$; // All nodes initially in $Q$ (unvisited nodes)
8: end for
9: while $Q$ is not empty do
10:    $u :=$ vertex in $Q$ with $min\ dist[u]$;// Source node in first case
11:    remove $u$ from $Q$;
12:    for each neighbor $v$ of $u$ do
13:       $alt := dist[u] + length(u, v)$;
14:       if $alt < dist[v]$ then
15:          $dist[v] := alt$;
16:          $previous[v] := u$;
17:       end if
18:    end for
19: end while
20: return $dist[\,], previous[\,]$;
```

their shortest paths.

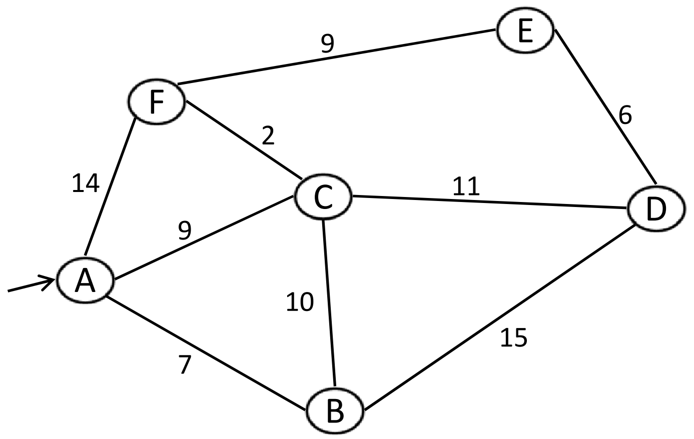


Figure 2.6: The graph for Example 2.

Table 2.2: Dijkstra's algorithm

	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	$ABCFDE$	-/-	-/-	-/-	-/-	-/-

Linear Programming

Linear programming (LP) is an important optimization technique which is in the form of constrained optimization. Constrained optimization is harder than unconstrained optimization because constrained optimization needs to satisfy several requirement in addition to finding the optimal point of the function. More formally, linear programming is defined as the problem of maximizing or minimizing a linear objective function, subject to linear constraints. The constraints maybe equalities or inequalities.

For example, find numbers x_1 and x_2 that maximize the sum of $x_1 + x_2$ subject to the constraints that

$$\begin{aligned} x_1, x_2 &\geq 0 \\ x_1 + 3x_2 &\leq 6 \\ 3x_1 + 2x_2 &\leq 12 \\ x_1 + x_2 &\leq 2 \end{aligned}$$

In this problem there are two unknown variables and five linear constrains. All constrains are inequalities and linear because each inequality is in some linear function of the variables. The objective function is to maximize $x_1 + x_2$.

Linear programming can be used in various fields of study, such as in business and economics. It has proved to be very useful in the optimal resource allocation problem. In this thesis, we use linear programming to solve redundant service removal problems in QoS-aware service composition.

Chapter 3

Anytime QoS-aware Service Composition over GraphPlan

3.1 Introduction

In this chapter, we study the kind of QoS-aware service composition problem that fulfills functional goals and achieves QoS optimization at the same time. To fulfill functional goals is the objective of automatic service composition problems. AI Planning [24] has been applied with success to solve service composition problems [14, 55]. Different algorithms have been proposed to solve planning problems and get plans from them such as depending on whether they are building the underlying graph structure in a forward (from initial state) or backward (from goal) way [24]. 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 time for each service is a unit. Thus, the shortest path has the shortest execution time, hence the shortest response time. With QoS consideration, the 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 vary. Therefore, we need to modify the classic planning algorithm to become QoS aware to find a QoS optimized solution.

Dynamic Programming (DP) and Integer Programming (IP) have been used to find a QoS optimized solution [16, 32, 34]. However, they are based on a given linearized formulation. To formulize the business constraints into linear algebra is not a straightforward process. It involves proper assignment of variables and some adjustments may be needed to make the relation among the linear variables. Hence, it is not straightforward to model service composition problem into DP or IP. IP is a known NP-hard problem. With an IP solver, the optimal solution without redundant services can be obtained. We also find that some of the models ([34] does not consider the possibility of reusing the same service in a plan).

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. This second step, called the service selection problem, is NP-complete [78, 80]. 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 selection 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 a predefined control flow can achieve.

In this chapter, we propose 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 use of the GraphPlan algorithm keeps the advantages of the AI planning model for easily modeling business logic, reusing the actions in one plan, and planning parallel actions in a plan.

The rest of this chapter is organized as follows. Section 3.2 gives the motivation to combine Dijkstra's algorithm with the GraphPlan algorithm. In Section 3.3, we provide a way to uniform the calculation of different QoS criterion for the purpose of aggregating multiple QoS values into one overall QoS value. In Section 3.4, a QoSGraphPlan algorithm is proposed to get the optimal solution with the best QoS value for a single criterion of throughput or response time in polynomial time. We also discuss the properties of our algorithm in this section. Section 3.5 discussed the redundant activities in the optimal solution. For the other QoS criteria, such as execution time, reputation, successful execution rate, and availability, Section 3.8 extends QoSGraphPlan with beam search to get the optimal solution with best QoS value for both single criterion problems and multiple criteria problems. Section 3.10 presents the results of the experiments with artificial data sets.. Finally, the conclusion is given in Section 3.11.

3.2 Motivation

The planning graph is a key structure built by the GraphPlan technique to represent the problem space. Since Dijkstra's algorithm is a graph traversal algorithm, we are motivated to combine Dijkstra's and the GraphPlan technique by using Dijkstra's algorithm on the planning graph.

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 value that is the best QoS value for each proposition. Then, we could get the overall cost-to-come value 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.

3.3 Calculation of the QoS

Some of the QoS criteria are negative (*i.e.*, the higher the value, the lower the quality). Response time and execution price are in this category. The other QoS criteria are positive (*i.e.,* the higher the value, the higher the quality). Throughput, reputation, successful execution rate, and availability are in this category. 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 [61] 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 2.9. For positive and non multiplication criteria (*i.e.,* throughput and reputation), values are scaled according to Equation 2.10. For positive and multiplication criteria (*i.e.,* successful execution rate and availability), values are scaled according to Equation 3.1. 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 [79] and [10] 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 calculate the utility value to make sure that lower utility value corresponds to better quality.

$$U_i(w_j) = \begin{cases} \frac{\ln(Q_i^{max}) - \ln(Q_i(w_j))}{\ln(Q_i^{max}) - \ln(Q_i^{min})} & \text{if } Q_i^{max} - Q_i^{min} \neq 0 \\ 1 & \text{if } Q_i^{max} - Q_i^{min} = 0 \end{cases} \tag{3.1}$$

Then, we use Equation 2.11 to calculate the overall quality score for each Web service.

For a network of services $\sigma = w_1, w_2, \ldots, w_n$, we also want to get the aggregated utility value. Equations 3.2 to 3.9 aggregate the single criterion values. We can easily prove that Equations 3.2 to 3.9 sort the networks in the order that Equations 2.3 to 2.8 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.

$$U_1(w_1; \ldots; w_n) = \sum U_1(w_i) \tag{3.2}$$

$$U_1(w_1 || \ldots || w_n) = \max U_1(w_i) \tag{3.3}$$

$$U_2(w_1; \ldots; w_n) = \max U_2(w_i) \tag{3.4}$$

$$U_2(w_1 || \ldots || w_n) = \max U_2(w_i) \tag{3.5}$$

$$U_3(w_1, \ldots, w_n) = \sum U_3(w_i) \tag{3.6}$$

$$U_4(w_1, \ldots, w_n) = \frac{1}{n} \sum U_4(w_i) \tag{3.7}$$

$$U_5(w_1, \ldots, w_n) = \sum U_5(w_i) \tag{3.8}$$

$$U_6(w_1, \ldots, w_n) = \sum U_6(w_i) \tag{3.9}$$

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 3.10. 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 [34].

$$U(\sigma) = \sum_{i=1,3,5,6} \frac{1}{n} U_i(\sigma) \times W_i + \sum_{i=2,4} U_i(\sigma) \times W_i \tag{3.10}$$

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

In the anytime QoS-aware service composition approach, 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 3.10.

If the QoS value is represented as a range (*e.g.*, [90-100]), we use a value (*e.g.*, the middle value between 90 and 100) as a representative of the set of values. Hence, we can still use the above formulation to calculate the QoS of a network of connected services.

3.4 Generation of Tagged Planning Graph

For simplicity, we present our algorithm using response time as the single quality criterion. For example, the $cost(a)$ in Algorithms 2-5 is either the QoS value (using Equations 2.1 and 2.2) or the utility value (using Equations 3.2 and 3.3) of response time for a service. Either way should get the same solution. The calculation of the other QoS criteria is discussed in Subsection 3.6.

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 7 *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 2 called *QoSGraphPlan* is our main algorithm. It is modified from the standard GraphPlan algorithm [24] to calculate TGP and extract a solution. *QoSGraphPlan* repeats *ExpandGraph* (Algorithm 3) (line 2) until a fixed point (Definition 8) 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 5) 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 3 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

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

```
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 3: $ExpandGraph(\langle P_0, A_1, ..., A_i, P_i \rangle)$

```
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 effects arcs to $pre(a)$ and $effects(a)$ 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 4: $Fixedpoint(\langle P_0, A_1, ..., A_i, P_i \rangle)$

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

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 can choose either one because both paths

Algorithm 5: *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 $\pi$
```

are equally the best. Lines 4 and 5 create the arcs between actions and propositions.

Algorithm 4 checks if a fixed point layer is reached.

Definition 8 *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 $\forall (p',t') \in P_k, (p',t') \in P_i$. $A_i = A_k$ have 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 5 first calculates the cost for the whole plan in U'. For the 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 of retrieving the recorded best parent for each involved proposition.

Example 3 *Figure 3.1 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 the 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 3,* $\max(cost(pre(w_1)))+cost(w_1) = 800$*. Therefore,* J *at layer* P_1 *has the response time of* 800*, w.r.t the utility value of* 1*. Table 3.1 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:* $\{w_2; w_4; w_8; w_7\}$*.*

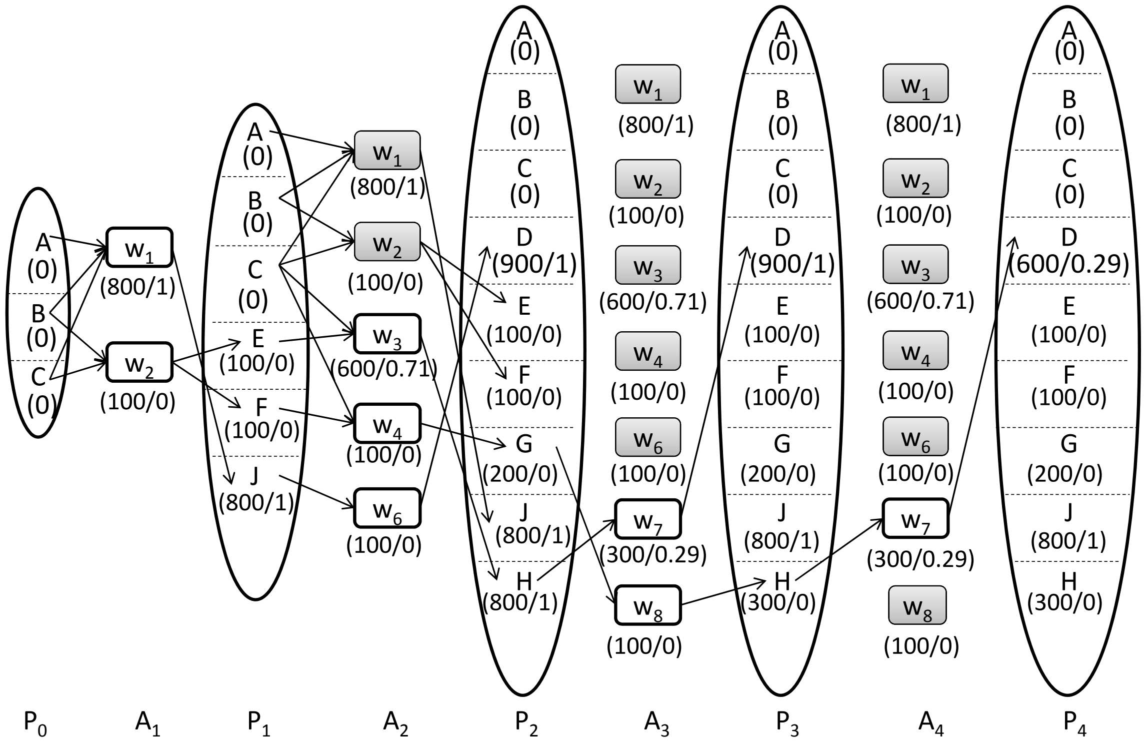


Figure 3.1: The tagged planning graph for Example 1.

Using *QoSGraphPlan* for throughput as single criterion. If we use QoS value according to Equations 2.3 and 2.4, line 2 in Algorithm 3 should use $t = \min_a(\min(cost(pre(a)), cost(a)))$ to calculate the cost value. If we use utility value according to Equations 3.4 and 3.5, $t = \max_a(\max(cost(pre(a)), cost(a)))$ should be used. When we calculate the cost for the whole plan in line 1 of Algorithm 5,

Table 3.1: 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

$U' = \min(cost(l)), \forall l \in g$ should be used, if we are using QoS value, or $U' = \max(cost(l)), \forall l \in g$ should be used, if we are using utility value.

The *QoSGraphPlan* algorithm works well for response time and throughput when they are considered as a single criterion. The *QoSGraphPlan* has polynomial time complexity (proofs in the next section). The *QoSGraphPlan* considers the best plan as the plan with the lowest cost. Practically, the *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 costs with the same best response time or throughput. We discuss the redundancy problem in Subsection 3.5. Redundancy removal is only necessary when multiple criteria are considered. For single criterion optimization, redundancy removal is not necessary.

3.4.1 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 5 is polynomial. The complexity to get a solution by Algorithm 5 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.

Understanding the scheduling of services over TPG. When a service is associated with the response time in TPG, a question that arises: 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 will start before all the services in the previous action layer are finished. We use ; and || to represent the connections of the services in TPG. However, this representation does not express the input and output dependency, nor the starting sequences in the sense of scheduling.

Example 4 *Figure 3.2 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* $\max(T(d_6), T(d_7)) = \max(90, 250) = 250$.

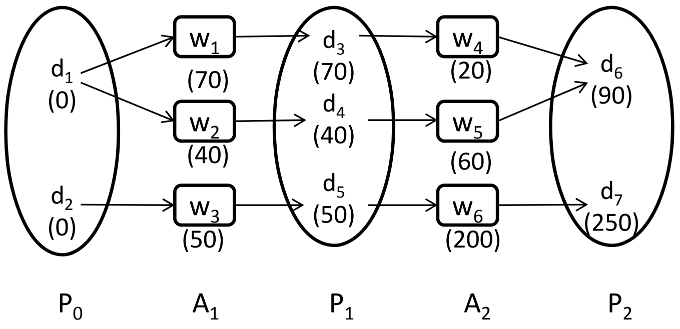


Figure 3.2: An example to explain the scheduling of services over TPG

3.5 Redundant Activities

QoSGraphPlan calculates the lowest cost for each proposition, including the goals. *QoSGraphPlan* simply combines 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. Please check the following example.

Example 5 *Figure 3.3 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.*

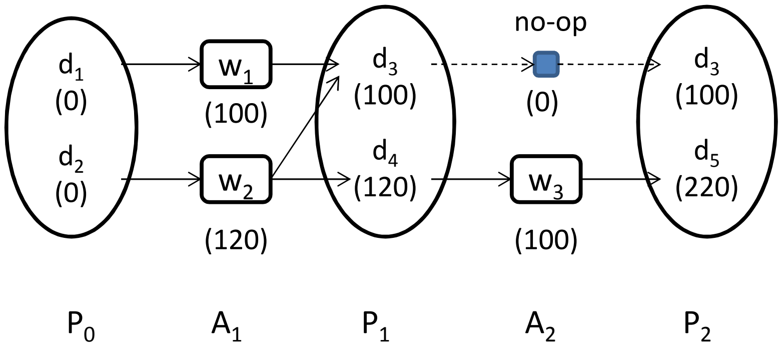


Figure 3.3: An example to explain redundant services

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

Redundancy exsits when we directly combine the best paths for each goal proposition. Without considering the execution price, redundancy removal does not change the optimal QoS value for the whole plan. However, redundancy removal implies reducing execution price. A full discussion on multiple criteria QoS optimization is in Subsection 3.7. 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. 10).

Definition 10 *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 11 *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 one other 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 3.4 is similar to Figure 3.3, 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 will not change.

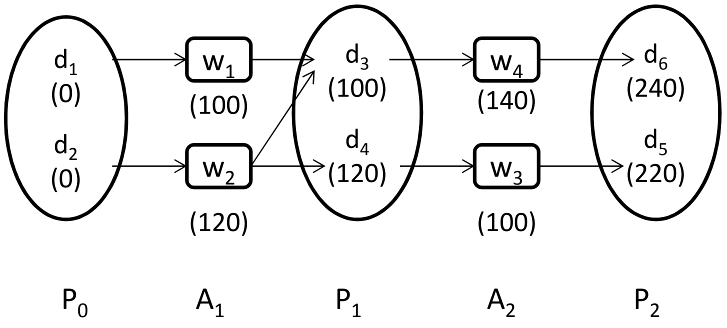


Figure 3.4: An example to explain redundant services

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. 6) scans the STPG 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 changes, 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.

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.*

Algorithm 6: *RemoveRedundancy*($\langle P_0, A_1, ..., A_n, P_n\rangle$))

```
Data: $\langle P_0, A_1, ..., A_n, P_n\rangle$) is a STPG
 1: for $i = n, \dots, 1$ 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 $\langle P_0, A_1, ..., A_n, P_n\rangle$)
```

Proof: Algorithm *RemoveRedundancy* 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 $|A|^3$. $\square$

RemoveRedundancy results in a lean solution without redundant services. In the cases where 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 2 to 6 can get composition solutions without redundancy, as well as the globally* *optimized QoS value in polynomial time.*

3.6 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 see the following example.

Example 7 *Reusing Figure 3.4 for execution cost. Assuming 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 the 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 getting 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 see the following example.

Example 8 *Figure 3.5 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.*

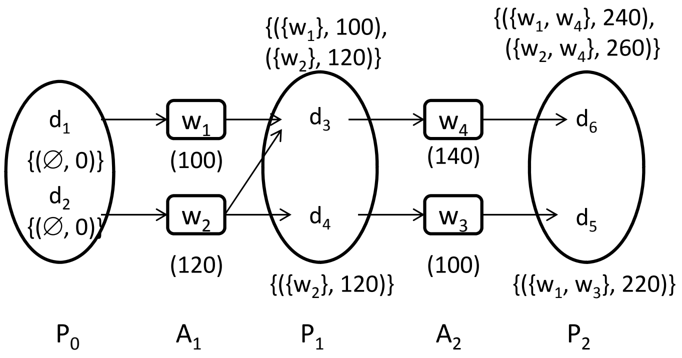


Figure 3.5: An example to explain multiple tags.

In the multiple tag situation, we use Algorithm 7 to replace Algorithm 3 to expand the TPG. Algorithm 7 calculates all the tags in line 2, which is the only difference to Algorithm 3.

QoSGraphPlanExt is the extended version of *QoSGraphPlan* which uses Algorithm 7 to generate the TPG. *QoSGraphPlanExt* probes all the combinations, which

Algorithm 7: *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 $\langle P_0, A_1, ..., A_i, P_i\rangle$;
```

Algorithm 8: *QoSGraphPlanExt*(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:    $G = ExpandGraphMultiTag(G, g)$;
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
```

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. 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 3.8 to solve the exponential problem. We have the following proposition for the properties of *QoSGraphPlanExt*.

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.*

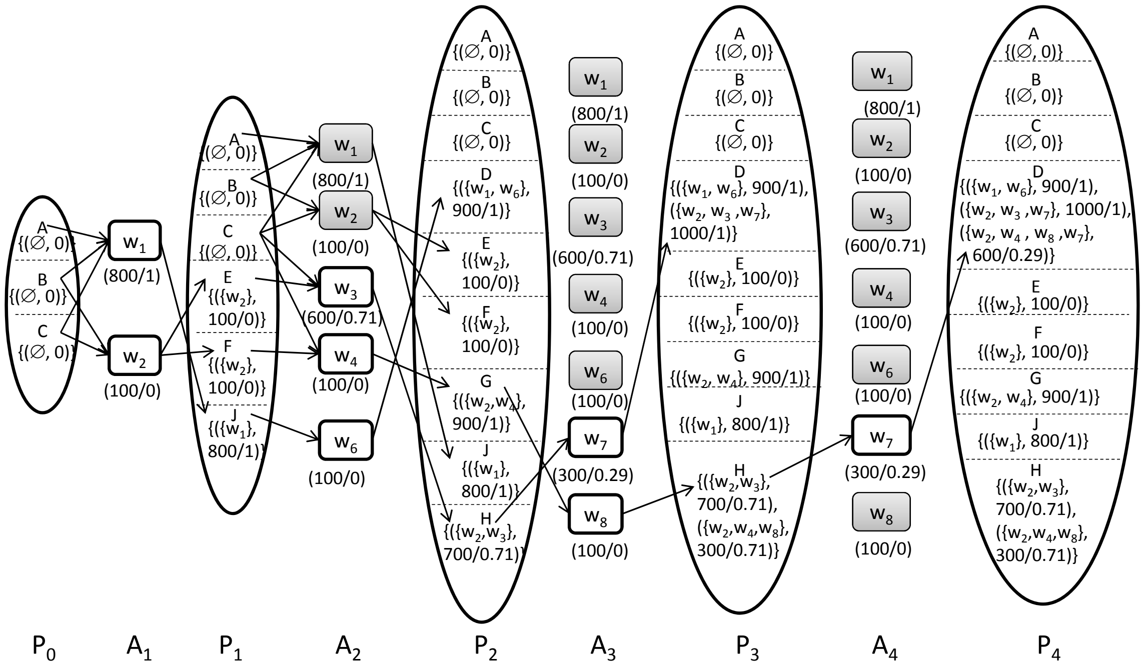


Figure 3.6: The multiple tags used for optimizing execution cost.

Example 9 *Suppose the quality criteria given in Table 2.1 is the execution cost. Figure 3.6 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.

3.7 Under Multiple Criteria

When we need to consider multiple QoS criteria, we aggregate them into a utility value according to Equation 3.10. 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 proposition* p *has two paths to reach* $\{w_1, w_2\}$ *and* $\{w_1, w_3, w_4\}$*. Table 3.2 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* $\{w_1, w_3, w_4\}$*.*

Table 3.2: 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.

3.8 BeamQoSGraphPlan with Beam Search

To solve the combination explosion problem, we incorporate *QoSGraphPlan* with beam search. Beam search uses a breadth-first search to build its search tree [69]. At each level of the tree, beam search 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 that are pruned. With an infinite beam width, no states are pruned and the beam search is identical to the 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 of the best tags for each proposition. We propose a heuristic function in our application as denoted by Equation 3.11. 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 to 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 9 to replace Algorithm 3 to expand the plan graph. Line 2 of Algorithm 9 includes the function to select the top K tags for a proposition. Therefore, *BeamQoSGraphPlan* includes Algorithm 2, 9, 4, and 5.

$$\begin{aligned} |h(t_j)| &= k_1 \times \sum U_i(\{t_j.parents\}) \\ &\quad + k_2 \times \frac{|g| - \left|\bigcup_{a\in t_j.parents} effect(a) \cap g\right|}{|g|} \\ &\quad \text{where } k_1 + k_2 = 1 \end{aligned} \tag{3.11}$$

Algorithm 9: *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

Algorithm 10: $BeamQoSGraphPlan(A, s_0, g)$

```
Data: $G = \langle P_0, A_1, P_1, ..., A_i, P_i \rangle$ is a planning graph; i=1;
1: repeat
2:     $G = ExpandGraphBeamWidth(G, g)$;
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
```

single criterion and multiple criteria problem. It is a heuristic algorithm. It does not guarantee optimal QoS and its solution may include redundant services.

3.9 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 2, 3, 4, and 5 to first construct a TPG and then extract a solution from the TPG.
- *QoSGraphPlan* is 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. 6) to remove redundancy.

QoSGraphPlanExt:

- *QoSGraphPlanExt* uses Algorithm 2, 7, 4, and 5 to first construct a TPG with multiple tags and then extract a solution from the multiple-tag TPG.
- It is best used for execution price, reputation, successful execution rate, and availability as single criterion.
- Complexity: exponential.
- The solution does not have redundant services and guarantees the optimal QoS value.

BeamQoSGraphPlan:

- *BeamQoSGraphPlan* uses Algorithms 2, 9, 4, and 5 to incorporate *QoSGraphPlan* with the beam search.
- It is 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. 6) to remove redundancy.

RemoveRedundancy:

- *RemoveRedundancy* as presented in Algorithm 6 removes redundant services from a solution to obtain a lean solution without redundancy.
- It is used for redundant service removal for single or multiple criteria.
- The solution after redundancy removal is not cost-optimized.
- No guarantee to get the optimal QoS value.

3.10 Empirical Results

3.10.1 Data Set

The data set used in our evaluation is generated by the test set generator in Web Service Challenge 2009 (WSC09) [7]. 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 [65]. 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 for 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.

3.10.2 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 [73] 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 3.7, 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”.

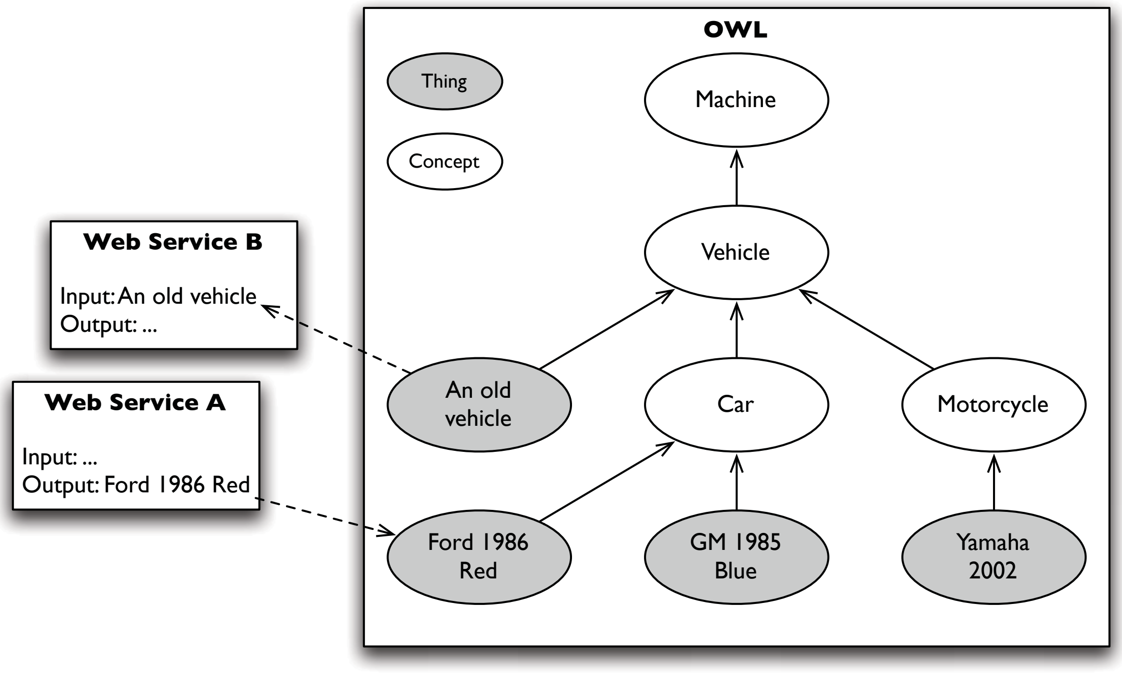


Figure 3.7: Semantic relationship between Web service input/output parameters

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 3.3. The indexing tables are stored as hash tables so that we can look up the services or the parameters in constant time.

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

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

Table 3.3: Top: example of the output indexing table; bottom: example of the reverse indexing table

Empirical Results

As the WSC09 data sets and the results are posted at [7], we are able to compare our results with the first place winning paper [32] and the second place winning paper [74] in terms of the response time and the 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 the values of 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.

Table 3.4: Results with the WSC09 Data Sets: our method/Paper [32]/Paper [74]

	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

In Table 3.4, 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 [32] generate zero redundant services in some datasets, but much more redundant services in the other datasets, especially when they compute throughput. [32] 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, the planning graph is a much more matured graph designed for planning than the graph in [32]. 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. [32] seems to filter out this possibility because the actions are not reused in their graph. Also, without using the concept of no-op, [32]'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 [74] 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. This is because [74] uses a simple breadth first search which could get only sequential solutions. Therefore, [74] is not able to get the optimal QoS value correctly, if some services can be concurrently executed.

According to the comparison, the *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.

Table 3.5: Our composition time: T1 with redundant services (Resp. Time/Throughput); T2 without redundant services (Resp. Time/Throughput).

	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

In Table 3.5, we show the composition time with redundant services (Comp T1, time used by *QoSGraphPlan*) and without redundant services (Comp T2, time used by *QoSGraphPlan* and *RemoveRedundancy*) for both response time and throughput. 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.

Table 3.6: Total execution price on the WSC09 Data Sets

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

In Table 3.6, we use *BeamQoSGraphPlan* on the execution price as single criterion[1]. 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*

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

to remove redundant services from the solution. As for the heuristic function in *BeamQoSGraphPlan*, we only consider the QoS of execution price. In Equation 3.11, 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 3.6, the *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.

Table 3.7: Results considering aggregated value of execution price and reputation on the WSC09 Data Sets

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

In Table 3.7, we use *BeamQoSGraphPlan* on execution price and reputation as multiple criteria[2]. In Equation 3.10, 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 3.11, we set $k_1 = 0.8$, $k_2 = 0.2$ and beam width $K = 5$ to calculate the heuristic function. Table 3.7 shows the utility value, the number of services (the lines of #Services) in the best solution, and the composition time

[2]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.

before and after removing redundant services (Comp T1 and Comp T2 respectively). We use *RemoveRedundancy* to remove redundant services from the solution.

According to the results in Table 3.7, *BeamQoSGraphPlan* algorithm can find the solution with the optimal aggregated QoS values of execution price and reputation on all datasets. The *RemoveRedundancy* algorithm can remove redundant services from the optimal solutions.

3.11 Summary

In this section, 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 problems. 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 the beam search which is a heuristic algorithm to get feasible solutions. 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.

Chapter 4

QoS-aware Service Composition over GraphPlan through Graph Reachability

4.1 Introduction

In this chapter, we improve the idea of combining the GraphPlan algorithm with Dijkstra's algorithm to solve QoS-aware service composition problems. One disadvantage of the method proposed in Chapter 3 is that there is no uniform graph structure to solve the problem for all kinds of QoS criteria. The tagged planning graph with single tags is built for single quality criterion of throughput or response time while the tagged planning graph with multiple tags is built for other single quality criterion or multiple criteria. In this chapter, we propose a method in which Dijkstra's algorithm works with a uniform structure to find an optimal solution for any single QoS criterion or multiple QoS criteria.

We find that it is hard to directly use Dijkstra's algorithm on the planning graph. The way that the planning graph is extended for the use of Dijkstra's algorithm suits for the calculation of certain quality criterion (*e.g.*, response time and throughput).

For the calculation of other quality criterion (*e.g.,* execution cost, availability, and reputation), the approach of using Dijkstra's algorithm on the extended planning graph has exponential time complexity.

Definition 12 ***Graph reachability*** *is the ability to reach one vertex from other vertices within a graph.*

If a vertex is able to be reached by one vertex, we call it **simple graph reachability**. There also exists the case that a vertex is reachable from several parent vertices together. In such case, we call it **complex graph reachability**, which is the case in the planning graph.

The planning graph can be traversed if actions and propositions are considered as vertices. The QoS value can be mapped to the *cost-to-come* value of the vertex. It is possible to use Dijkstra's algorithm to traverse the graph to search for a plan with the optimal QoS value.

In this section, we are motivated to change the graph reachability by graph conversion. Graph conversion generates a converted graph with simple graph reachability from a extended planning graph with complex graph reachability. The objective of Dijkstra's algorithm is to find costs of the shortest paths from a single vertex to a single destination vertex. Since each vertex in a graph with simple graph reachability is able to be reached from another single vertex, it is convenient to use Dijkstra's algorithm on a converted graph with simple graph reachability. To do so, we need to overcome some difficulties. First of all, a suitable formula to calculate the *cost-to-come* value for each vertex is important. It is because the QoS value is mapped into the *cost-to-come* value. The calculation of QoS value for different QoS criteria follows different formula. Secondly, two types of vertices in the planning graph (*i.e.,* action vertices and proposition vertices) lead to complex graph reachability. An action vertex and a proposition vertex correspond to a service and an input/output parameter accordingly. A service is invokable when all its inputs are available. Edges link the input parameters of a service to this service itself. Hence, an action vertex may be reachable from several propositions at the same time. Complex graph reachability makes it difficult to apply Dijkstra's algorithm. We need to find a way to change the graph reachability in planning graph for easily using Dijkstra's algorithm.

The rest of this chapter is organized as follows. Section 4.2 gives the framework of this method. Section 4.3, Section 4.4, and Section 4.5 elaborate each step of the proposed method using response time as the single QoS criterion. In Section 4.6, we discuss how to get the optimal solution for other single QoS criterion. Section 4.7 presents the results of the experiments with artificial data sets. We end up with a conclusion in Section 4.8.

4.2 The Framework

The improvement is achieved by graph conversion that bridges the GraphPlan technique and Dijkstra's algorithm. For the sake of simplifying graph reachability, graph conversion builds a converted graph, called the Layered Weighted Graph (LWG), from the extended planning graph constructed by the GraphPlan technique. LWG contains all the information presented in the planning graph. Also, LWG has simple graph reachability which makes it easy to use Dijkstra's algorithm. The framework of our approach is presented in Figure 4.1.

4.2.1 Planning Graph Labeling

In the first step, a Partially Labelled Planning Graph (PLPG) is generated to represent the problem space. The PLPG extends the classic planning graph in the way that each proposition is associated with a set of labels. PLPG only labels proposition vertices rather than action vertices in the classic planning graph. This is why PLPG is called partially labeled. However, TPG proposed in Section 3.4 tags both proposition vertices and action vertices in the classic planning graph.

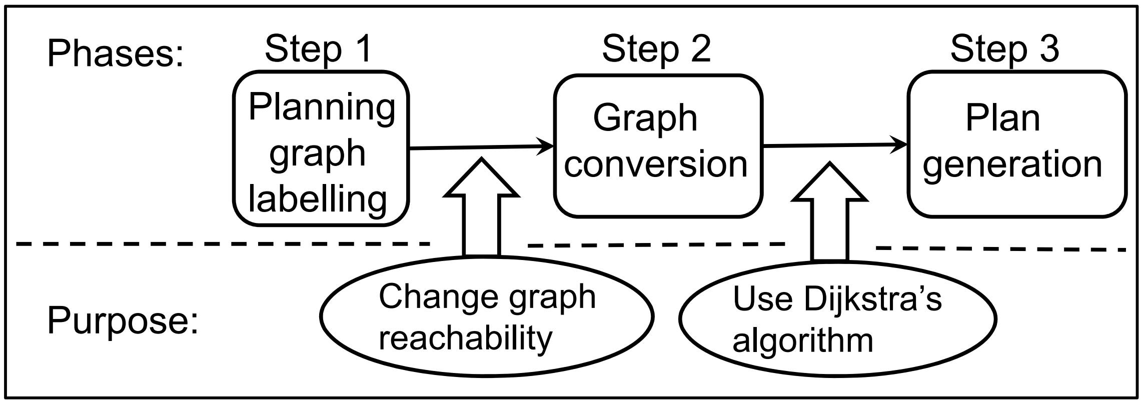


Figure 4.1: The framework of our approach

Definition 13 *A **label** t is a triple (a, L_a, C_a), where a is an action enabled at action layer A_k, L_a is the layer number ($L_a = k$), and C_a is a real number representing the cost for a to be enabled.*

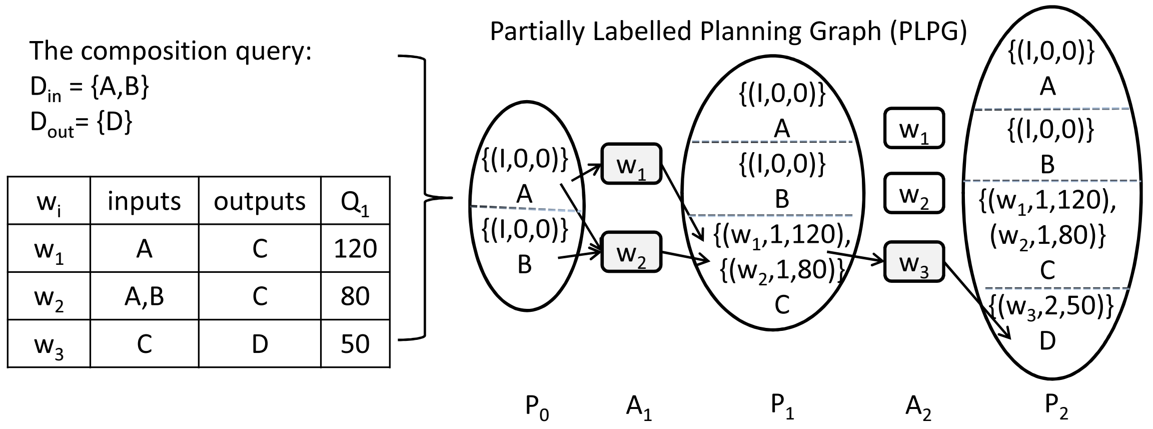
A label describes a situation when an action is enabled (*i.e.,* the action, the layer number where this action is enabled, and the cost to enable this action). In order to easily get the information contained in t, functions $act(t)$, $layer(t)$, and $cost(t)$ are defined such that $act(t) = a$, $layer(t) = L_a$, and $cost(t) = C_a$. The layer number L_a needs to be included in the situation because there are re-used actions in the planning graph. The situation when action a is enabled at action layer A_i is different from the situation when a is enabled at a different action layer A_j ($j \neq i$).

Definition 14 *A label t is a label of proposition p if $p \in effects(act(t))$.*

A label t of a proposition p associates p with a situation (described by t) when p is produced. A parent action a of p is fast retrieved from the parent label by function $act(t)$, *i.e.,* $a = act(t)$.

Definition 15 *The Partially Labelled Planning Graph (PLPG) is a Planning Graph $G = (V_A \cup V_P, E)$ where each proposition $p \in V_P$ at proposition layer P_i is associated with a set of labels T of p, i.e., $T = \{t | p \in effects(act(t)) \wedge 0 \leq layer(t) \leq i\}$.*

In the PLPG, the labels of proposition p at proposition layer P_i records all the situations when p is produced at the current proposition layer or any previous proposition layer. Labelling each proposition in the PLPG is for the aim of retrieving paths that can produce this proposition in the future. The retrieved paths are composed of actions and obtained through the backward search.


The composition query:
D_{in} = {A,B}
D_{out}= {D}

w_i	inputs	outputs	Q_1
w_1	A	C	120
w_2	A,B	C	80
w_3	C	D	50

Figure 4.2: An example of PLPG

Figure 4.2 shows a simple example of how to generate a PLPG from a composition query and a set of services. In Figure 4.2, "Q_1" denotes a QoS value (*e.g.,* response time in million seconds). "I " is a supposed action for any input in D_{in}. Proposition C at proposition layer P_1 has two labels $(w_1, 1, 120)$ and $(w_2, 1, 80)$. This is because C is generated by w_1 and w_2 at layer P_1.

4.2.2 Graph Conversion

In the second step, a Layered Weighted Graph (LWG) is converted from a PLPG. The purpose of a LWG is to simplify graph reachabilty for the ease of using Dijkstra's algorithm. We are able to use a simple path in a LWG to represent a plan for the planning graph problem. A plan in the planning graph represents a solution for the original service composition problem. A simple path means the path is a sequence of vertices rather than a sequence of sets of vertices. Simple graph reachability makes the simple path possible. Simple graph reachability is achieved in LWG by the proper design of vertices.

A vertex v in a LWG is a set of ordered pairs where each ordered pair (p, t) consists of a proposition p and a label t of p. An ordered pair (p, t) records a situation, described by t, to produce p. In a vertex v, an ordered pair (p, t) is designed for the fast retrieving of the parent action (contained in t) for each proposition p. Let T_v and P_v be the set of all labels and propositions contained in v respectively. Therefore, a vertex v describes a situation to produce a set of propositions P_v. In a vertex v, we require that no label can be replaced by any other in the set of labels T_v. Otherwise, more actions are enabled to produce the propositions contained in v which indicates more execution costs.

Definition 16 *A **vertex** v in a LWG is a set of ordered pairs, i.e.,*

$$v = \{(p, t) | p \in act(t) \text{ where } p \text{ is a proposition and } t \text{ is a label of } p\}$$

Any label t in v satisfies that:

$$\forall t \in T_v,\ \nexists t' \in T_v - \{t\}\text{: } (effects(act(t)) \cap P_v) \subseteq effects(act(t'))$$

where T_v (resp., P_v) is the set of all labels (resp., propositions) in v.

Since an action can produce more than one proposition, it is possible that the labels of several propositions in v are the same or several labels in v contain the same action. According to Definition 16, for a label t in v, we are only interested in the portion of propositions in P_v that actions in t can produce, *i.e.,* $effects(act(t)) \cap P_v$. If there is another label t' in v, action in t' can also provide this portion of propositions (*i.e.,* $(effects(act(t)) \cap P_v) \subseteq effects(act(t'))$). In such a case, t' can replace t in v. A LWG as defined by Definition 17 contains only one type of vertices.

Definition 17 *The **Layered Weighted Graph** (LWG) is a connected graph whose vertices are placed in hierarchically arranged layers $V_0, \ldots, V_n$. Each edge (v', v), which is associated with a weight $w(v', v)$, connects only vertices in successive layers. Each weight w is a set of labels in v, i.e., $w = \{t | t \in T_v \wedge v \in V_i : layer(t) = i\}$.*

A LWG is a layered graph. Vertices in a LWG are linked to vertices at adjacent layers. For a weight w assigned to edge (v', v) where v' and v are at layers V_{i-1} and V_i respectively, all the labels in w come from v. Also, each label in w contains an action that is enabled at action layer A_i in the corresponding PLPG. The weight w actually bridges the path from v' to v. $w \rightarrow (v', v)$ indicates that actions in w are enabled to support the path going from v' to v. We will propose an appropriate method to make this support reasonable. For example, v' provides all the propositions in the preconditions of actions in w and v contains all the propositions in the postcondition of actions in w.

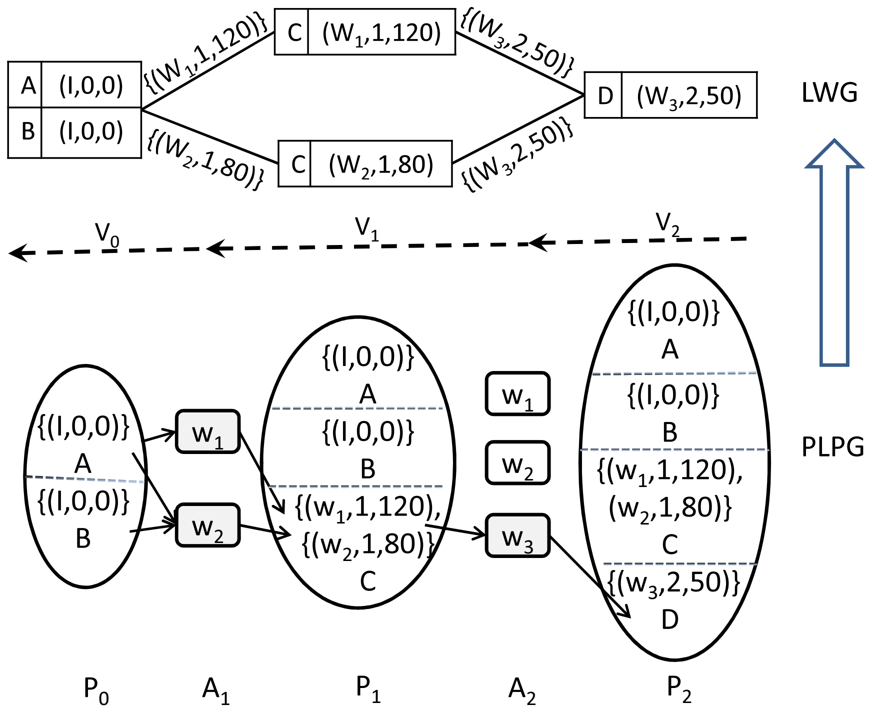


Figure 4.3: The LWG converted from the PLPG in Figure 4.2

Figure 4.3 presents a LWG that is converted from the PLPG in Figure 4.2. The dashed arrow shows the direction to gerenate the LWG. At beginning, we generate vertices at layer V_2 from proposition layer P_2 in the PLPG. Since D is the expected output parameter, each vertex at layer V_2 has only one ordered pair which takes D as the proposition. Starting from proposition layer P_2 which contains all the expected output parameters to construct the LWG quickly filters out the propositions and the actions that have no contribution to produce the expected output parameters.

4.2.3 Plan Generation

In the third step, we use Dijkstra's algorithm to calulate the *cost-to-come* value for each vertex in the LWG then we extract the plan with the best *cost-to-come* value through backtracking.

4.2.4 Main Algorithm

Algorithm 11 is the main algorithm. First, we build a partially labelled planning graph G to represent the problem space (line 1). Then, we check if G can achieve all goals (line 2-3). If all goals are achieved, G is converted into a layered weighted graph GC (line 4) and a plan with the best *cost-to-come* value is generated (line 5).

Algorithm 11: $QoSWSC(A, s_0, g)$

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

4.3 Planning Graph Labelling

Algorithm 12 builds the PLPG layer by layer. For simplicity, we use response time as a single quality criterion. The cost of action a is the response time of a. Originally, we assume all the given propositions are the effects of a dummy service I and the cost of I is 0 (line 1). Line 4 to line 23 generates action layer A_i and proposition layer P_i until a fixed point is detected. $Fixedpoint(G)$ is a function to check if a fixed point layer is reached. A fixed point in PLPG is a layer k such that for $\forall i (i \geq k)$, $A_i = A_k$ and $P_i = P_k$. Line 5 gets all the enabled actions for action layer A_i. The enabled actions are those whose inputs are in the previous proposition layer P_{i-1}. SA_i is a set of re-used or newly enabled actions contained in A_i (line 6). SP_i is a set of propositions that are the effects of SA_i (line 7). P_i contains all the propositions that are the effects of actions in A_i (line 8). Line 9 - 17 calculates the labels for each proposition in P_i. We use $T_i(p)$ to denote the labels of p at proposition layer P_i. If proposition p in P_i also belongs to $P_{i-1} - SP$ (line 10), this means p is inherited from P_{i-1} and p is not re-produced in P_i by some actions in A_i. Therefore, the labels of p in P_i is the same as the labels in P_{i-1} (line 11). If proposition p in P_i belongs to both P_{i-1} and SP (line 12), it means p is re-produced in P_i. The labels of p are its labels at P_{i-1} plus the labels composed of actions in SA that can produce p (line 13). If proposition p belongs to SP other than P_{i-1} (line 14), this means p is a newly generated proposition. The labels of p are composed of newly enabled actions that can produce p (line 15). Line 19 and 20 create arcs between actions and propositions.

Example 11 *A set of available services with their input/output parameters and response time in milliseconds are listed in Table 5.1. The composition query is* $(D_{in}, D_{out}) = (\{A, B\}, \{D, G\})$. *We construct a PLPG as in Figure 5.2.*

In Figure 5.2, we do not draw the no-op actions. This is because a no-op action inherits a true proposition from a previous proposition layer and has no cost. According to Algorithm 12, we calculate SA_i *at each action layer (the shaded actions in Figure 5.2). To make the figure readable, we only draw the arcs connecting to or from the shaded actions in the action layers. Please notice that the graph reaches the fixed point at layer* A_3. *There are four solutions:* $\{w_1; w_3, w_4\}$, $\{w_1, w_2; w_3; w_4\}$, $\{w_1, w_2; w_4, w_5; w_3\}$, *and* $\{w_2; w_4, w_5; w_3\}$.

Algorithm 12: *PLPGGeneration*(A, s_0, g)

```
1: $P_0 = \{(p, T_p^0) | p \in s_0, T_p^0 = \{(I, 0, 0)\}\}$;
2: $SP = s_0$;
3: $i = 1$;
4: repeat
5:    $A_i = \{a | pre(a) \subseteq P_{i-1}, a \in A\}$;
6:    $SA_i = \{a | a \in A_i \wedge pre(a) \cap SP \neq \emptyset\}$;
7:    $SP_i = \bigcup_{a \in SA} effects(a)$;
8:    $P_i = \{p | \exists a \in A_i : p \in effects(a)\}$;
9:    for each $p \in P_i$ do
10:      if $p \in P_{i-1} - SP$ then
11:         $T_i(p) = T_{i-1}(p)$;
12:      else if $p \in P_{i-1} \cap SP$ then
13:         $T_i(p) = T_{i-1}(p) \cup \{(a, i, C_a) | \exists a \in SA : p \in effects(a)\}$;
14:      else if $p \in SP - P_{i-1}$ then
15:         $T_i(p) = \{(a, i, C_a) | \exists a \in SA : p \in effects(a)\}$;
16:      end if
17:   end for
18:   for each $a \in A_i$ do
19:      link $a$ with precondition arcs to $pre(a)$ in $P_{i-1}$;
20:      link $a$ with to each of its $effects(a)$ in $P_i$;
21:   end for
22:   $i = i + 1$;
23: until $Fixedpoint(G)$
24: return $\langle P_0, A_1, ..., A_n, P_n \rangle$;
```

Table 4.1: A set of available services

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

Theorem 5 *Algorithm 12 PLPGGeneration has polynomial time complexity.*

Proof: The principle of *GraphLabel* is to expand a PLPG layer by layer until a fixed point layer k is reached. 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

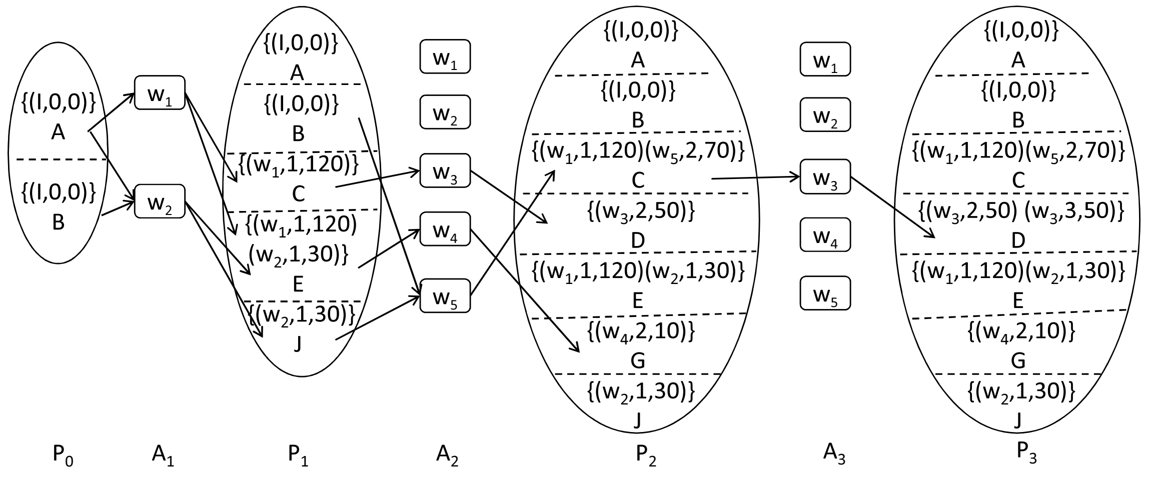


Figure 4.4: The partially labelled planning graph for Example 11.

proposition may be associated with different labels, 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 PLPG with k layers is $|s_0| + (m+2n)k$. The time to expand a PLPG to layer k is polynomial to the size of the planning problem. Therefore, *GraphLabel* has polynomial time complexity. $\square$.

4.4 Graph Conversion

Graph conversion converts a PLPG into a LWG. One the one hand, a LWG is a "layered" graph since the PLPG is composed of alternating proposition layers and actions layers. Similar to a PLPG, a LWG is built layer by layer. In terms of proposition P_i in the PLPG, layer V_i in the LWG is generated. Assume P_n and P_0 are the proposition layers in the PLPG that contains goal and initial inputs respectively. Different from a PLPG, the construction of a LWG starts from V_n to V_0 where V_n and V_0 correspond to P_n and P_0 respectively. Since the goal of a PLPG is also the objectives for a LWG to achieve, it is easy to decide vertices in V_n such that each vertex at V_n contains all goal propositions. We use V_n as the starting point for the construct of a LWG.

One the other hand, a LWG with simple graph reachability contains only one type of vertices. In a LWG, a vertice v, as defined in Definition 16, at V_i represents the connectivities between actions at A_i and propositions at P_i in the corresponding PLPG. For a proposition p in v, we can quickly find the parent action a of p when p is located at proposition layer P_i in the corresponding PLPG. In a LWG, vertices are linked to vertices at adjacent layers. Parent vertices of v at V_{i-1} are located at V_{i-1}. When generating the parent vertices of v, we treat the ordered pairs in a vertex v separately:

- Some labels in the ordered pairs in v contain actions that are not enable at A_i. These ordered pairs are kept in the parent vertices of v. Since actions in these labels will be enabled at any other layer A_j $(0 \leq j < i)$.
- Some labels in the ordered pairs in v contain actions that are enabled at A_i. To support these actions to be enabled, the parent vertieces of v must provide all the propositions in the preconditions of the enabled actions. We call the propositions in the preconditions are newly-added propositions in the parent vertices. We use a weight w to record these labels that contain enabled actions in v. The weight w is assigned to the edge (v', v) to bridge the path from v' to v.

According to proposition layer P_{i-1} in the corresponding PLPG, each proposition may be associated with several labels. For the newly added propositions in the parent vertices, it is possible to generate several sets of ordered pairs for these propositions. Therefore, a vertex v may have several parent vertices. Finally, the construct of a LWG stops at layer V_0.

Algorithm 13 describes how a PLPG is converted into a LWG. The inputs of Algorithm 13 are the PLPG G, the given inputs s_0, the goal g. Initially, each vertex layer is set to be an empty set (line 1-3). Since it is easy to figure out which propositions should be contained in a vertex, *e.g.,* the goal propositions g, we start

Algorithm 13: $GraphConversion(PLPG, s_0, g)$

```
Data: $PLPG = \langle P_0, A_1, P_1, ..., A_n, P_n\rangle$ is a partially labelled planning graph,
      $s_0$ is the initial state; $g$ is the goal;
1:  for $i = 1, \ldots, n$ do
2:     $V_{i-1} = \emptyset$;
3:  end for
4:  $V_n = \{v|v$ is a vertex $\wedge P_v = g \wedge T_v \subseteq T_n(p)\}$;
5:  $i = n$;
6:  repeat
7:     for each $v \in V_i$ do
8:        $w = \{t|t \in T_v : layer(t) = i\}$;
9:        if $w = \emptyset$ then
10:          $v'$ is a copy of $v$;
11:          $LayerUpdate(V_{i-1}, V_i, v', w, v)$;
12:       else
13:          $op' = \{(p,t)|(p,t) \in v \wedge layer(t) < i\}$;
14:          $PRE = \bigcup_{t \in w} pre(act(t))$;
15:          $T_{PRE} = \bigcup_{p \in PRE} T_{i-1}(p)$;
16:          $OP = \{op|op$ is a set of ordered pairs $\wedge P_{op} = PRE \wedge T_{op} \subseteq T_{PRE}\}$;
17:          for $op \in OP$ do
18:             $ST = \{t|t \in T_{op} \wedge layer(t) = i-1\}$;
19:             $SP = \bigcup_{t \in ST} effects(act(t))$;
20:             $CT = \{t|t \in w : pre(act(t)) \cap SP \neq \emptyset\}$;
21:             if $CT = T_{(\bullet,v)}$ then
22:                $v' = op' + op$;
23:                if $v$ is a vertex then
24:                   $LayerUpdate(V_{i-1}, V_i, v', w, v)$;
25:                end if
26:             end if
27:          end for
28:       end if
29:    end for
30:    $i = i - 1$;
31: until $i = 1$
32: $v' = \{(p,(I,0,0))|p \in s_0\})$;
33: $w = \{t|t \in T_v : layer(t) = 1\}$;
34: for each $v \in V_1$ do
35:    $LayerUpdate(V_0, V_1, v', w, v)$;
36: end for
37: return $\langle V_0, \ldots, V_n\rangle$;
```

from P_n to build the LWG. We generate V_n in the LWG according to P_n in the PLPG (line 4). As for any vertex v in V_n, each ordered pair in v takes a proposition p in g as the first entry and a label of p as the second entry. Therefore, we can quickly find a set of actions to produce g in terms of v. Once we have vertex layer V_i (initially, line 5 shows $i = n$), we build the the vertex layer V_{i-1} (line 6-25). For each vertex v in V_i, all parent vertice of v linked to v with the same weight w. The weight w are composed of the labels in v that are enabled at layer i (line 8).

If w is empty, no actions in v can be enabled such that no propositions in v are produced at layer i. In such cases, the parent vertex v' is a copy of v (line 10). This is because that actions in v are enabled at a layer k ($k < i$). $LayerUpdate(V_{i-1}, V_i, v', w, v)$ (line 11) is a function that adds the newly generated parent vertex v' into vertex layer V_{i-1}, links v with its parent vertex v', and associates the edge (v', v) with a weight w.

If w is not empty (line 12), some propositions in v are produced since their parent actions are enabled at layer i. In the LWG, $(v', v) \rightarrow w$ means v' can provide all preconditions of actions in w (also in v) such that these actions are enabled in v. Let op' be a set of ordered pairs in v where the label of each ordered pair (p, t) in op' contains an action that can not be enabled at layer i (line 13). For a vertex v at layer V_i, we consider the parent vertex v' of v from two respectives: op' and $v - op'$ (line 12-22):

- For op', op' is kept in the parent vertex v'. This is because the labels in op' are not enabled at layer i but at some previous layer k ($k < i$).

- For $v - op'$, the label t for each ordered pair (p, t) in $v - op'$ must contain an action that is enabled at layer V_i. We calculate a situation that can enable actions in $v - op'$ (*i.e.,* actions in w). Also, this situation is added into the parent vertex v' to support enabled actions in v through w.

There are many situations to enable actions in $v - op'$. Let PRE be the set of propositions that are the preconditions of actions in w (line 14). T_{PRE} is the label sets of PRE at proposition layer P_{i-1} (line 15). In terms of PRE and T_{PRE}, a family of the ordered pair sets OP is generated to describe all the situations to produce all propositions in PRE (line 16). For each ordered pair set $op \in OP$, we check if op is the ordered pair set to enable all the actions in w. ST is the set of labels in op whose actions are enabled at layer V_{i-1} (line 18). SP is the effects of actions in ST (line 19). In terms of op, CT is a set of labels in w whose actions can be actually enabled at layer V_i (line 20). If CT is equal to w, it means op is the right ordered pair that every action in w can be enabled at layer V_i (line 21). In this case, we assume that the parent vertex v' is the union of op and op' (line 22). If v' is also a vertex (line 23), no actions in v' can be replaced. Then, v' is a parent vertex. $LayerUpdate(V_{i-1}, V_i, v', w, v)$ is invoked to update vertex layer V_{i-1}, add the edge (v', v), and associates (v', v) with weight w (line 24). Otherwise, v' is not a parent vertex of v. For an initial proposition, we suppose its parent action is I and the cost of I is 0. Therefore, the parent label for each initial proposition is $(I, 0, 0)$. V_0 has only one vertex v' where v' takes initial inputs s_0 as the key values and each key is mapped to the parent label $(I, 0, 0)$. v' is linked to each vertex v in V_1. $LayerUpdate(V_0, V_1, v', w, v)$ is invoked to update vertex layer V_0 , add the edge (v', v), and associates (v', v) with weight w (line 35).

In a LWG, each vertex is able to be reached by another one vertex. It is easy to use Dijkstra's algorithm on the LWG. A path from the single source vertex to a vertex in V_n corresponds to a plan. Actions in the same vertex are enabled in parallel. Actions in different vertices are enabled in the order of when vertices can be reached. All the parent vertices of a vertex v is generated from v.

Proposition 5 *Let v be a vertex at vertex layer V_i. V' is a set of parent vertices of v (i.e., $V' = \{v' | v' \in V_{i-1} : v'$ is a parent of $v\}$). For $\forall v' \in V'$, there is only one weight w such that $(v', v) \rightarrow w$.*

Proposition 5 specifies that any edge that links a parent with v' is associated with the same weight. The reason is that all labels in w come from labels in v and each of these labels contains an action that is enabled at layer A_i in the corresponding PLPG.

Proposition 6 *Let v be a vertex at layer V_i. If v' is a parent vertex of v (i.e., $(v', v) \to w$), we have $P_{v'} = P_v - \bigcup_{t \in w} effects(act(t)) + \bigcup_{t \in w} pre(act(t))$.*

Proposition 6 states the relation among the propositions in the parent vertex v', the propositions in v, and the propositions in the preconditions of actions in the weight w that is assigned to the edge (v', v). Assume v and v' are at layer V_i and V_{i-1} respectively, all actions in w are enabled at layer A_i in the corresponding PLPG. Proposition 6 shows that propositions in the parent vertex v', denoted as $P_{v'}$, removes the propositions produced by actions in w, denoted as $\bigcup_{t \in w} effects(act(t))$. Also, Proposition 6 guarantees that $P_{v'}$ must contains the preconditions of actions in w, denoted as $\bigcup_{t \in w} pre(act(t))$, to support actions in w to be enabled at layer A_i.

Theorem 6 *Algorithm 13 GraphConversion has polynomial time complexity.*

Proof: *GraphConversion* converts a PLPG with k layers, n propositions, and $|A|$ actions, into a LWG. The conversion process starts from the top layer k. A vertex in LWG contains a set of propositions and a set of actions called parent actions that produce propositions in the vertex. The maximal number propositions in one vertex is n, and each proposition is maximally produced by $|A|$ services. The maximal number of combination of parent actions that can produce n propositions in one vertex is $|A|^n$. A vertices layer in LWG contains maximal $|A|^n \times n$ vertices. Since a LWG has k layers, the maximal number of vertics in a LWG is $|A|^n \times n \times k$. Therefore, the time complexity of *GraphConversion* is $|A|^n \times n \times k$.

Example 12 *According to Algorithm 13, the PLPG in Figure 5.2 is converted into the LWG as shown in Figure 4.5. Due to the space limit, we only present the propositions and the actions in the PLPG that are involved in construct of the LWG. The dashed arrow in the middle of Figure 4.5 separates the LWG (on the upper side) with the PLPG (on the lower side). The dashed arrow also denotes the direction to generate the layers of LWG (i.e., from V_3 to V_0). Firstly, two vertices are generated at vertex layer V_3 in terms of propositions D and G at P_3 in the PLPG. Then, V_2 is generated in terms of V_3 and P_2. For example, we generate the parent vertices for vertex $v = \{(D, (w_3, 3, 50)), (G, (w_4, 2, 10))\}$ at V_3. Since only w_3 in v is enabled at A_3 in the PLPG, the weight $w = \{(w_3, 3, 50)\}$ is the label set mapping to the edges linking v's parent vertices with v. Hence, $op' = \{(G, (w_4, 2, 10))\}$ is the mapping set which will be kept in parent vertices of v. w_3 produces D and has the input C. We have $PRE = \{C\}$. The label set of C at layer P_2 is $\{(w_1, 1, 120), (w_5, 2, 70)\}$, then $OP = \{(C, (w_1, 1, 120)), (C, (w_5, 2, 70))\}$. In terms of $op = \{(C, (w_1, 1, 120))\}$, ST is the empty set since no label whose action can be enabled at V_2. Then, SP is also an empty set. In such case, CT is an empty set since no service in the weight w can be enabled. Hence, $op = \{(C, (w_1, 1, 120))\}$ is discarded. Similarly, we calculate ST, SP, and CT for $op = \{(C, (w_5, 2, 70))\}$. We have the result that CT is equal to $w = \{(w_3, 3, 50)\}$. Hence, $op = \{(C, (w_5, 2, 70))\}$ is the right ordered pairs. $v' = op + op' = \{(C, (w_5, 2, 70)), (G, (w_4, 2, 10))\}$ is a vertex and recorded as a parent of v. We continue the calculation until the vertex layer V_1. V_0 contains only one vertex $v = \{(A, (I, 0, 0)), (B, (I, 0, 0))\}$ who takes initial inputs as keys. Finally, we connect v with each vertex in V_1 to complete graph conversion.*

4.5 Plan Generation

We perform plan generation on a LWG. Plan generation contains two steps: forward search and backtracking.

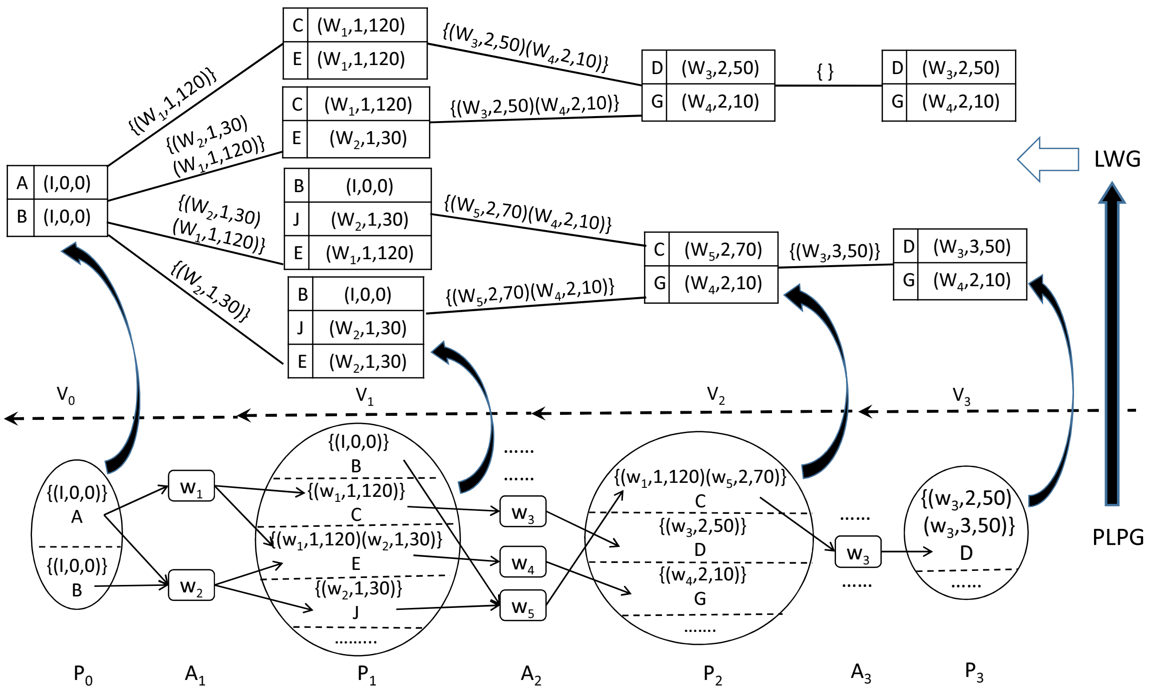


Figure 4.5: The Layered Weighted Graph (LWG) for Example 12.

During forward search, we use the Dijkstra's algorithm to calculate the *cost-to-come* value for each vertex v. The principle behind the calculation is based on the actions in the weights assigned to the edges. If a vertex v' can reach v, it indicates that v' contains the preconditions to enable all the actions in the weight assigned to edge (v', v) such that v contains the propositions produced by these actions in the weight. Also, the calculation needs to look at the service models to identify that a proposition belongs to the precondition or the post-condition of an action. The forward search starts from layer V_0 to layer V_n because each vertex V_0 contains the initial state and each vertex V_n contains the goal. Since each vertex contains a set of propositions, the *cost-to-come* value for each vertex v is considered as the optimal QoS value to obtain all the propositions in v. Let $r(v)$ be a function to get the *cost-to-come* value for each vertex v. Since different quality criterion has different calculation for QoS value, the format of the *cost-to-come* value for v is different in terms of quality criteria. Accordingly, the method for choosing the *cost-to-come* value for v varies. For simplicity, we use response time as a single quality criterion in Algorithm 14. We will discuss other quality criterion in Section 4.6. Let C_p^v be the QoS value for obtaining proposition p in v. If the response time is the quality criterion, the *cost-to-come* value $r(v)$ is a set of QoS values ($i.e.,\ r(v) = \{C_p^v | p \in P_v\}$). If there is an execution path leading to a vertex v, the QoS value for obtaining the propositions in v through this path is calculated by accumulating the QoS values of actions assigned to the edges in this path. If there are several paths leading to a vertex v, the optimal path is the path that produces all the propositions in v with the minimum response time. Since the response time to obtain all the propositions in v is actually decided by the proposition obtained with the maximum response time ($i.e.,$ $\max\{C_p^v\}$), the optimal path makes $\max\{C_p^v\}$ have the minimum value among all the paths. Hence, $r(v)$ is calcuated by accumulating the QoS values of actions assigned to the weights on the optimal path.

Backtracking starts from the vertex v at V_n with the optimal *cost-to-come* value. This is because any vertex V_n contains the goal. As for quality criterion of response time, a vertex v at V_n has the optimal *cost-to-come* value if the maximum QoS value of a proposition in v is minimum as compared to any other vertex at V_n. We extract an execution path from the LWG through backtracking. The execution path starting from a vertex at V_0 to a vertex at V_n corresponds to an optimal plan for the planning problem. The plan is retrieved by extracting the actions in the weights assigned to the edges in the path. Actions are enabled in the order of the weights to be extracted, while actions in one weight are enabled in parallel.

Algorithm 14 describes how to find a best plan from the LWG. First, we calculate the *cost-to-come* value for each vertex at layer V_0. V_0 contains only one vertex v that includes all the initial inputs. We assume the cost to obtain each proposition in v is zero (line 1-3), since no action needs to be enabled to produce any propsosition in v. Next, the *cost-to-come* values for vertices are calculated layer by layer and the calcuation goes from V_1 to V_n (line 14-18). For any vertex v at proposition layer V_i,

Algorithm 14: *PlanGeneration*(LWG, A, g)

```
Data: $LWG = \langle V_0, \ldots, V_n \rangle$ is a layered weighted graph; $A$ is a set of actions;
      $g$ is the goal;
 1: for each $v \in V_0$ do
 2:     $r(v) = \{0, \ldots, 0\}$;
 3: end for
 4: $i = 1$;
 5: repeat
 6:     for each $v \in V_i$ do
 7:         $V' = \{v' | (v', v) \in E_i\}$;
 8:         if $w = \emptyset$ then
 9:             The only parent $v'$ is recorded as the best parent of $v$;
10:             $r(v) = r(v')$;
11:         else
12:             $v' = \min^{-1}_{v' \in V'} \{\max\{C^v_p | C^v_p \in CalQoS(v', v, w) : t^v_p \in w\}\}$;
13:             $v'$ is recorded as the best parent of $v$;
14:             $r(v) = CalQoS(v', v, w)$;
15:         end if
16:     end for
17:     $i = i + 1$;
18: until $i = n$;
19: $v = \min^{-1}_{v \in V_n} \{\max_{C^v_p \in r(v)} \{C^v_p | p \in P_v\}\}$;
20: for $i = n, \ldots, 1$ do
21:     $\pi_i = \bigcup_{t \in w} act(t)$;
22:     $v'$ in $V_{i-1}$ is the best parent of $v$;
23:     $v = v'$;
24: end for
25: return $\pi$;
```

we calculate the *cost-to-come* value $r(v)$ in terms of all of its parent vertices (line 6-16). V' contains all the parent vertices of v (line 7). If weight $w = \emptyset$, no action needs to be enabled to reach v from any parent v' (line 8). Then v has only one parent v' who has the same content as v. v' is recorded as the best parent of v and $r(v) = r(v')$ (line 9-10). Otherwise, some action in weight w has to be enabled to reach v from any parent v' (line 11). For vertex v reachable from its parent v', the QoS value for obtaining the propositions in v from v' is calculated by function $CalQoS(v', v, w)$. $CalQoS(v', v, w)$ uses Equation 4.1 calculates the QoS value for obtaining each proposition in v from v' through w.

$$C_p^v = \begin{cases} \max_{p' \in pre(act(t_p^v))}\{C_{p'}^{v'}\} + cost(t_p^v), & \text{if } t_p^v \in w \\ C_p^{v'}, & \text{otherwise} \end{cases} \tag{4.1}$$

where t_p^v denotes the label mapped to p in vertex v.

Equation 4.1 treats each proposition from case to case.

- If $t_p^v \in w$ for proposition p, it means the action in t_p^v is enabled in v such that p is produced. The cost of p, *i.e.,* C_p^v, is the maximal costs of all of the preconditions of action $act(t_p^v)$ plus $cost(t_p^v)$.
- If $t_p^v \notin w$ for proposition p, the action in t_p^v is not enabled in v such that p is not produced. The cost of p in v is the same as the cost of p in v', *i.e.,* $C_p^v = C_p^{v'}$.

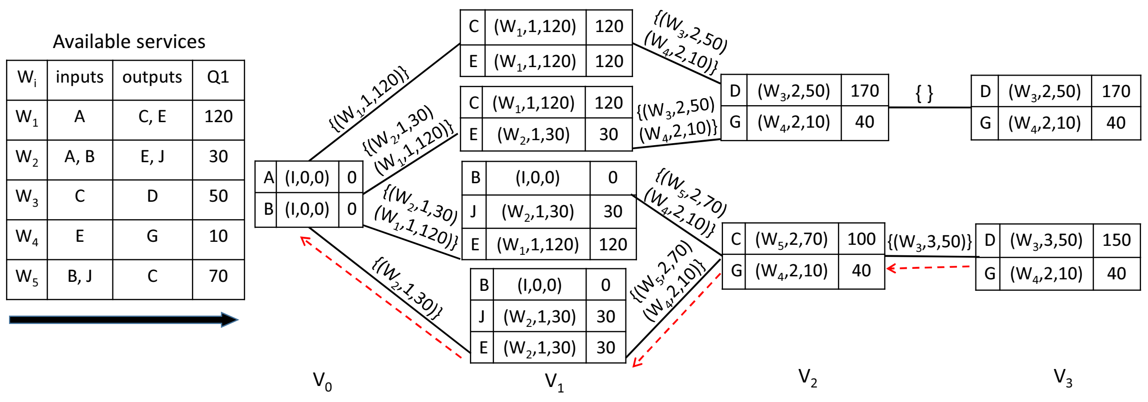
The cost to reach v from v' is actually decided by the maximum cost of p produced in v and the action in parent label of p is enabled by the propositions in v'. This is what $\max\{C_p^v | C_p^v \in CalQoS(v', v, w) : t_p^v \in w\}$ means at line 12. If there are several parent vertices to reach v, we choose the parent v' which causes the minimum cost to reach v (line 12). v' is recorded as the best parent (line 13) and $CalQoS(v', v, w)$ as the *cost-to-come* value of v (line 14).

Backtracking starts from the selected vertex in V_n. The extraction of an optimal plan consists of retrieving the path that can reach this vertex with minimum cost (line 19-24).

Theorem 7 *Algorithm 14 PlanGeneration has linear time complexity.*

Proof: *PlanGeneration* contains two steps: using Dijkstra's algorithm to traverse the graph from the initial layer 0 to the top layer k, and extract a plan through backtracking. For a LWG with $|E|$ edges and V vertices, Dijkstra's algorithm traverse each vertex through the edges from the initial vertex layer by layer. Sine the LWG is a layer graph and each vertex has only one parent vertex, Dijkstra's algorithm scans each edge once when it reaches the goal vertex. In total, Dijkstra's algorithm scans $|E|$ edges to reach the destination vertex. For the extraction phase, *PlanGeneration* extracts one path at each layer. Since there are k layers in the LWG, it takes k operations to extract a plan. Therefore, the time complexity of *PlanGeneration* is $|E| + k$. □.

Example 13 *Following Example 12, the dotted arrow in Figure 5.3 shows the trace of backtracking for plan generation. The solid arrow below the table of available services shows the direction to apply the Dijkstra's algorithm on the LWG.*


Available services

W_i	inputs	outputs	Q1
W_1	A	C, E	120
W_2	A, B	E, J	30
W_3	C	D	50
W_4	E	G	10
W_5	B, J	C	70

Figure 4.6: Plan Generation for Example 12.

*We use Algorithm 14 to calculate the value of each vertex in Figure 5.3. For a vertex v in Figure 5.3, the QoS values of propositions are shown in the rightmost column in v. For example, the value of vertex $v = \{(G, (w_3, 2, 50)), (D, (w_4, 2, 10))\}$ at V_2 is $\{C_D^v, C_G^v\} = \{170, 40\}$. v has two parents v' and v''. $w = \{(w_3, 2, 50), (w_4, 2, 10)\}$ is mapped to any edge at E_2 that links a parent with v. This means both w_3 and w_2 are enabled when a path goes from a parent to v through an edge. We use $CalQoS(v', v, w)$ to calculate the **cost-to-come** value from a parent v' to v. If we choose v' at V_1 with the QoS value of $(C_C^{v'}, C_E^{v'}) = \{120, 120\}$ to reach v, we get $C_D^v = \max\{C_C^{v'}\} + cost(w_3, 2, 50) = 120 + 50 = 170$ and $C_G^v = \max\{C_E^{v'}\} +$*

$cost(w_4, 2, 10) = 120+10 = 130$. *Hence,* $CalQoS(v', v, w) = \{C^v_D, C^v_G\} = \{170, 130\}$. *If we choose* v'' *at* V_2 *with the QoS value of* $(C^{v''}_C, C^{v''}_E) = \{120, 30\}$ *to reach* v*, we get* $CalQoS(v'', v, w) = \{C^v_D, C^v_G\} = \{170, 40\}$. *Since the parent* v'' *reach* v *with the minimum response time,* v'' *is recorded as the best parent and* $CalQoS(v'', v, w)$ *as the best value of* v*. Similarly, we calculate the values for other vertices and record the best parents accordingly. The best solution* $\{w_2; w_4, w_5; w_3\}$ *is obtained by retrieving the path through the dotted arrow in Figure 5.3. The response time of the best solution is* 150 *milliseconds.*

4.6 Other Single QoS Criteria

If throughput is considered as a single QoS criteria in Algorithm 14, we use Equation 3 and Equation 4 to calculate *cost-to-come* value for vertices. We also change the calculation in the corresponding places in Algorithm 14. Line 2 sets the *cost-to-come* value for a proposition in V_0 as the maximum throughtput among all actions (*i.e.,* $\max_{a\in A} C_a$). To calculate the *cost-to-come* value of proposition p, we replace Equation 4.1 with Equation 4.2.

$$C^v_p = \begin{cases} \min\{\min_{p'\in pre(act(t^v_p))}\{C^{v'}_{p'}\}, cost(t^v_p)\}, & \text{if } t^v_p \in w \\ C^{v'}_p, & \text{otherwise} \end{cases} \tag{4.2}$$

Line 12 should be $v' = \max^{-1}_{v'\in V'}\{\min\{C^v_p | C^v_p \in CalQoS(v', v, w) : t^v_p \in w\}\}$. The min function in line 2 means the *cost-to-come* value of v that can be reached from v' is decided by the minimum throughput of all propsitions in v. If there are several parent vertices, we choose the parent v' that can reach v with the maximum *cost-to-come*. This is what function $\max_{v'\in V'}$ means in line 12. Line 19 should be $v = max^{-1}_{v\in V_n}\{\min_{C^v_p\in r(v)}\{C^v_p | p \in P_v\}\}$, since the vertex with the maximum *cost-to-come* at layer V_n must be contained in the best plan.

For other quality criteria, we count the QoS values in terms of all the actions included in the plan. If an action presents twice in a plan (*i.e.,* a re-used action), this action is only considered once for the calculation of the QoS values of vertices on the plan. Let A_v be a set of actions that are enabled on the path to produce propositions in v. We add the calcluation of A_v for each vertex v in Algorithm 14. For the only vertex v in V_0, we add $A_v = \{\}$ after line 2. For vertex v at other layer V_i $(i > 0)$, we add the calculation of A_v somewhere between line 6 to line 18. If $w = \emptyset$ (line 8), we add $A_v = A_{v'}$ after line 10. Otherwise, we add $A_v = A_{v'} \cup \{act(t) | t \in w\}$ after line 14. Table 4.3 lists other changes that have to be made accordingly in Algorithm 14.

4.7 Experimental Results

We use WSC09 data set [7], as explained in Section 3.10. WSC09 has five data sets denoted as D1-D5 in Table 4.2. Dataset 1 has 500 services and 5,000 concepts. Dataset 5 has 15,000 services and 100,000 concepts. The other data sets have 4,000-8,000 services and 40,000-60,000 concepts.

We compare our method *QoSWSC* with method *QoSGraphPlan* or method *QoSGraphPlanExt* presented in Chapter 3 in terms of the quality of the solutions. *QoSGraphPlan* is used when the single quality criterion is response time or throughput. *QoSGraphPlanExt* as an extension of *QoSGraphPlan* is used to calculate the best QoS value for the single criterion of execution price, reputation, successfull rate, or availability. In Table 4.2, we show whether the correct QoS values can be calculated (the checkmarks). We can see that method *QoSWSC* can find the correct QoS values for all the five data sets. In Chapter 3, We find *QoSGraphPlanExt* can not find the solution with the best QoS value for these criteria. This is because *QoSGraphPlanExt* uses a heuristic search while searching the best solution for these criteria. The heuristic method may find the local optimal rather than the

Table 4.3: Changes made in Algorithm 14 for different single criteria (r,r' are real values, $k, k' \geq 0$ are integers)

QoS criteria	format of $r(v)$	line 2	line 12	line 19
exe. price	r	$r(v) = 0$	$v' = \max^{-1}_{v' \in V'} \{r(v') + \sum_{\substack{t \in w: \\ act(t) \notin A_{v'}}} cost(t)\}$	$v = \max^{-1}_{v \in V_n} r(v)$
reputation	(r, k)	$r(v) = (0, 0)$	$v' = \max^{-1}_{v' \in V} \frac{r'+C_w}{k'+\lvert\lvert w-A_{v'}\rvert\rvert}$ where $C_w = \sum_{\substack{t \in w: \\ act(t) \notin A_{v'}}} cost(t)$	$v = \max^{-1}_{v \in V_n} \{\frac{r}{k}\}$
succ. exe. rate / availability	r	$r(v) = 1$	$v' = \max^{-1}_{v' \in V'} \{r(v') \times \prod_{\substack{t \in w: \\ act(t) \notin A_{v'}}} cost(t)\}$	$v = \max^{-1}_{v \in V_n} r(v)$

global optimal solution.

Table 4.2: Results with the WSC09 Data Sets:
method *QoSWSC*/method *QoSGraphPlan* (*QoSGraphPlanExt*)

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

4.8 Summary

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

Chapter 5

Redundant Service Removal in QoS-aware Service Composition

5.1 Introduction

In this chapter, we study the redundancy removal problem to further optimize the QoS optimal solutions obtained by QoS-aware service composition algorithms, where QoS-aware service composition achieves functional goals and QoS optimization at the same time.

The QoS-aware service composition problems can be classified into single QoS criterion problems or multiple QoS criteria problems. To solve multiple QoS criteria problems, people use a preemptive model where multiple single criterion problems are solved in a sequence of priorities or a non-preemptive model where an aggregated score is calculated as a single criterion for optimization. Sometimes, execution cost is not in the list to optimize. In this case, many composition algorithms can get the solutions with optimal QoS values. However, the solutions obtained can possibly contain redundant services, the removal of which does not worsen the QoS value of the solution. For example, in the literatures using Web Service Challenge (WSC) open data sets [7], *e.g.,* [32], [74], and *QoSGraphPlan* approach proposed in Chapter 3, we find that all the algorithms generate redundant services with almost every data set. The removable services can be over 30% of the services in a solution in some cases. Surprisingly, this very common problem is ignored by all the WSC participants and is not considered in the evaluation of the WSC results. In reality, even if the execution cost is not under consideration to reduce the number of services included in the solution without worsening the QoS performance is a reasonable requirement. This is our motivation to do this research.

In this chapter, for simplicity, we consider response time or throughput as the first optimal criterion in this paper. Execution cost can then be considered as the second criterion in the preemptive model. We model the redundancy removal problem as an integer programming problem. Though solvable using a standard solver, we present an algorithm to solve the problem in this specific context and it proves to have better performance than a standard integer programming solver. We also present the results of our data experiments. Experimental results show that our method can find the cost-optimized solution by removing redundant services and the optimal response time (or throughput) of the solution is guaranteed.

5.2 Motivation

Our previous work in Chapter 3 combined a planning algorithm called the Graph-Plan [8] with Dijkstra's algorithm to solve the single QoS criterion service composition problem. When a second QoS criteria is under consideration, we find the removal of some services from the solution will further optimize the QoS value of the second QoS criteria while keeping the satisfaction of functional goals and optimization of the first QoS criterion untouched. We will explain in detail what triggered us with the following example.

Example 14 *A set of available services are presented in Table 5.1. A composition query is* $(\mathcal{D}_{in}, \mathcal{D}_{out}) = (\{A, B, C\}, \{K, L, J\})$*, and its objective is to optimize*

Table 5.1: Services in the solution shown in Figure 5.2

Service	inputs	outputs	response time	cost
w_1	A	D, E	40	20
w_2	B	F	20	30
w_3	C	G, I	120	30
w_4	D	N, H	30	20
w_5	E, F	H, I	70	30
w_6	G	M, J	100	50
w_7	G	J	120	20
w_8	H	K	50	30
w_9	I	L	20	30
w_{10}	J	Q	30	30

response time. Figure 5.1 shows the Tagged Planning Graph (TPG) for this service composition problem. For clarity, we do not draw the duplicated services if they have appeared in the previous action layers. In Figure 5.1, each service is labelled with its response time, and each proposition is labeled with the optimal response time to obtain it. The computation of the optimal response time of each proposition is similar to the classic Dijkstra's algorithm, except now the optimal value depends on several inputs. The solution with optimal response time $\{(w_1||w_2||w_3);(w_4||w_5||w_6);(w_8||w_9)\}$ *is generated by retrieving the recorded best parents for each goal. The TPG of the solution is shown in Figure 5.2. The TPG in Figure 5.2 actually removes the services and propositions that do not exist in the solution from the TPG in Figure 5.1. The response time of the solution is* 220 *because this is the maximum response time to obtain the individual goals. The execution cost of the solution is 240 by adding up all the costs of the services in the solution. The bold arrows in Figure 5.1 and 5.2 are the optimal paths to get* $\mathcal{D}_{out}$. *no-op is a dummy activity to keep the service layers and the parameter layers interleaving. no-op takes 0 time and 0 cost to execute.*

The service composition problem is an AI planning problem without negative effects. It is known that it is a polynomial time problem [31]. It is also known that it takes polynomial time to construct a Planning Graph [8]. The *QoSGrpahPlan*

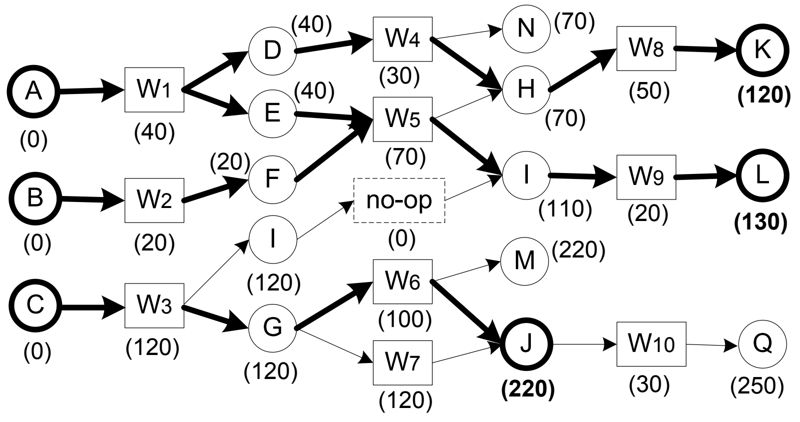


Figure 5.1: The tagged planning graph for **Example 14**

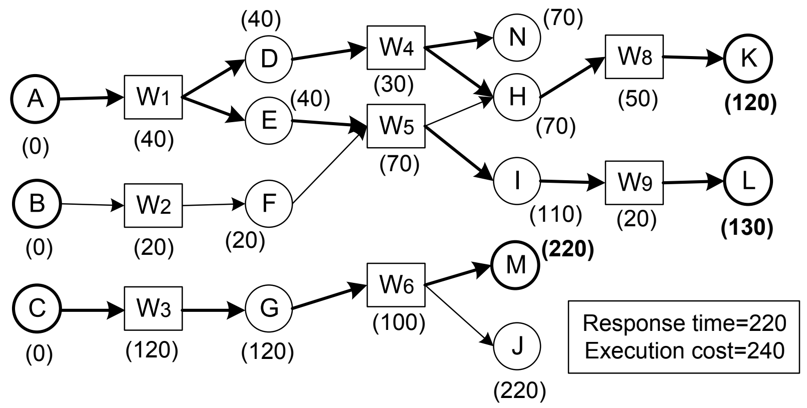


Figure 5.2: The TPG for the solution for **Example 14**

algorithm can solve a single QoS criterion (*i.e.*, throughput or response time), service composition problem with throughput or response time in $O(|w|^2)$ time complexity[1], where w is the set of services.

Example 15 *Let us check the solution in Fig. 5.2. When the total execution cost for the solution is taken into consideration, we find it is possible to reduce the total cost without worsening the optimal response time by removing some services. Please check the following two cases:*

Case 1*: w_4 is removed. The final solution becomes* $\{(w_1||w_2||w_3); (w_5||w_6); (w_8|| w_9)\}$*. The total cost of the solution is reduced to* 220 *while the response time of the*

[1]Other constraints apply for the other single criteria.

solution remains 220.

Case 2: w_5 *and* w_2 *are removed. The final solution becomes* $\{(w_1||w_3);(w_4||w_6);(w_8||w_9)\}$. *The total cost of the solution is reduced to* 180 *while response time of the solution remains* 220.

In Section 3.5 of Chapter 3, we detected the existence of redundant services in an optimal solution. The optimal solution is valid because it satisfies the functional goals, and has an optimal QoS value. However, it is necessary to remove the redundant services, because the removal can reduce the execution cost. After checking the papers using the same data sets (e.g. [32] and [74]) we found this is a common problem. Surprisingly, people seem satisfied after obtaining a solution with the optimal QoS value without further studying this problem. We consider that the fundamental reason for this problem is that execution cost is not one of the criteria to optimize. In reality, even if execution cost is not under consideration, to reduce the number of services included in the solution without worsening the QoS performance is a reasonable requirement.

Intuitively, the redundant services are due to the reproduction of the outputs by multiple services. For example, output H is produced by both w_4 and w_5, which brings the possibility to remove one of them. However, due to the complexity of how the outputs of w_4 and w_5 are used, it is difficult to make a decision without checking the whole solution carefully.

In this chapter, we propose a method to use the execution cost as a second criterion to further optimize the solution obtained from optimizing the first criterion [2]. Our method can be extended to solve more general multiple criteria optimization problems with a preemptive model.

[2] Limit to throughput and response time in this thesis

5.3 Analysis of Redundant Service Removal

We extend the definition of a Direct Acyclic Graph (DAG) $G = (V, E)$ to represent a solution to a QoS-aware composition query.

Definition 18 *An Extended Direct Acyclic Graph (EDAG) $EG = (V, E)$ is a direct acyclic graph with alternating levels of vertices V_p and vertices V_s, where $V = V_p \cup V_s$ is the vertex set and $E = (V_p \times V_s) \cup (V_s \times V_p)$ is the edge set.*

Similar to a planning graph, a EDAG is a layered graph with alternating levels of proposition vertices and service vertices. However, an EDAG contains two dummy service vertices which do not exist in a planning graph. The two dummy service vertices are added into an EDAG as a single source vertex and a single destination vertex. We assume that the dummy service corresponding to the single source vertex provides the initial inputs parameters, while the precondition of the dummy service corresponding to the single destination vertex consists of the expected output parameters.

In an EDAG, we label the levels starting from level 0 where P_i is a level of vertices V_p and S_i is a level of vertices V_s. The precondition of each service at layer S_i $(i > 0)$ is provided by services at layer S_{i-1}.

We map a solution to an EDAG in the following way:

- The vertices V_p are the parameter vertices.
- The vertices V_s are the service vertices.
- The edges $V_p \times V_s$ connect the input parameters with the services.
- The edges $V_s \times V_p$ connect the services with their output parameters.
- The initial input of the solution is considered to be a dummy service with $\emptyset$ as its inputs and $\mathcal{D}_{in}$ as its outputs.

- The expected output of the solution is considered to be a dummy service with $\mathcal{D}_{out}$ as its inputs and $\emptyset$ as its outputs.

Example 16 *Fig. 5.3 shows the EDAG for the solution in Example 11. D_I and D_G are the two dummy services.*

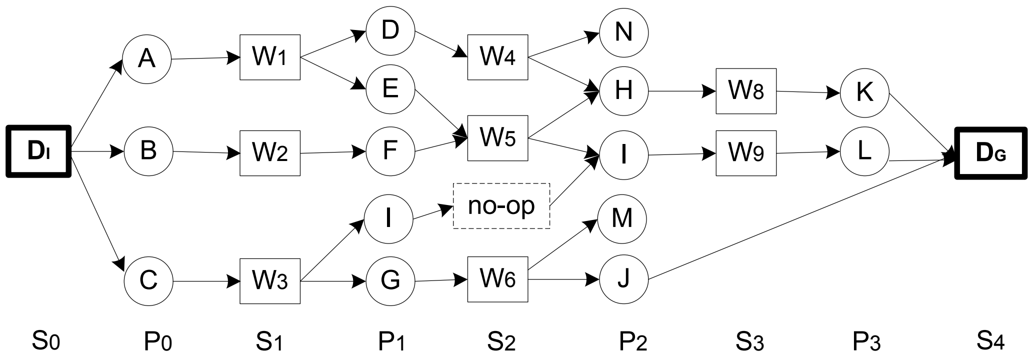


Figure 5.3: An EDAG with labelled levels for Example 11

Definition 19 *A parameter is a key parameter, if it is used as an input parameter of a service in the EDAG .*

One condition of a redundant service is as follows:

Definition 20 *A service is* ***redundant*** *if all its key outputs are reproduced by some other services.*

Example 17 *w_4 has one key output H, and H is reproduced by w_5. Therefore w_4 is a redundant service. w_5 has two key outputs H and I, both of which are reproduced by some other services. Therefore, w_5 is a redundant service.*

Redundant services can be possibly removed without worsening the optimal QoS value. The removal can cause some other services to become useless and can be also removed. Please check the following example.

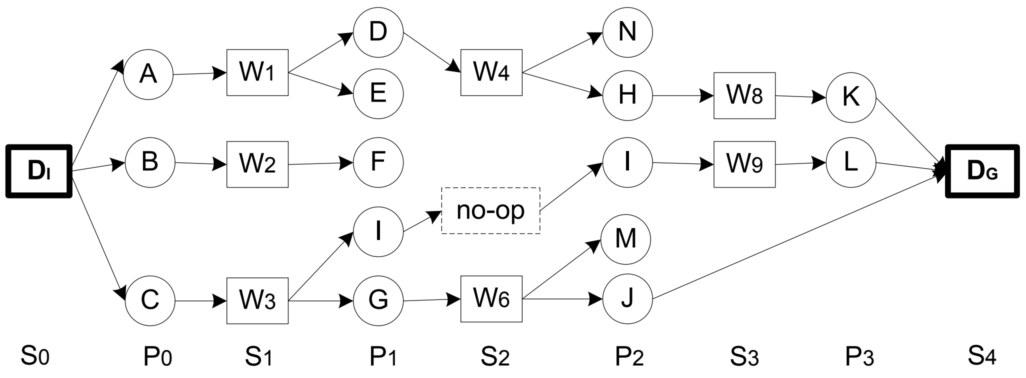


Figure 5.4: The EDAG after w_5 is removed from Fig. 5.3

Example 18 *After removing service* w_5 *from Fig. 5.3, the new EDAG is shown in Fig. 5.4. Obviously,* w_4 *is not a redundant service anymore and* w_2 *can be further removed because its only output is not a key output.*

Therefore, we have the following proposition:

Definition 21 *A service is useless and removable if none of its outputs is a key parameter.*

Removing services from a solution may worsen the optimal QoS value for a solution. Please check the following example.

Example 19 ***Change the response time of*** w_6 ***in Example 11 to*** 25. *Table 5.2 shows response time before and after redundancy removal. Response time of the solution increases after service* w_4 *is removed. Hence,* w_4 *cannot be removed from the solution. After removing service* w_2 *and service* w_5*, response time of the solution remains optimal. Since removing services implies reducing cost, the solution after removing* w_2 *and* w_5 *becomes the optimal solution.*

5.4 Model Redundant Service Removal Problem

Assume a solution *sol* with optimal response time contains n services $\{w_1, \ldots, w_n\}$. The total number of concepts $m = |\mathcal{C}|$, where $\mathcal{C}$ is the concept set. We model the re-

Table 5.2: Response time before and after redundancy removal for Example 19

Solution	**Goal**	**Execution path**	**Response time**	
Before removing service	K	$\{w_1; w_4; w_8\}$	120	145 (solution)
	L	$\{w_1 \| \| w_2; w_5; w_9\}$	130	
	J	$\{w_3; w_6\}$	145	
After removing w_4	K	$\{w_1 \| \| w_2; w_5; w_8\}$	160	160 (solution)
	L	$\{w_1 \| \| w_2; w_5; w_9\}$	130	
	J	$\{w_3; w_6\}$	145	
After removing w_2, w_5	K	$\{w_1; w_4; w_8\}$	120	145 (solution)
	L	$\{w_3; w_9\}$	140	
	J	$\{w_3; w_6\}$	145	

dundant service removal problem as an integer programming problem $(X, D, C, f(sol))$, where X is the variable set, D is the domain set, C is the constraint set and $f(sol)$ is the objective function on sol.

Definition of variables and domains of the model

Variable set $X = \{X_0, X_1, \ldots, X_n, X_{n+1}\}$ corresponding to the set of services in sol, where

- X_i $(1 \leq i \leq n)$ is regarded as a real services w_i $(1 \leq i \leq n)$ in sol
- X_0 is a dummy service corresponding to $\mathcal{D}_{in}$
- X_{n+1} is a dummy service corresponding to $\mathcal{D}_{out}$
- Each variable X_i has a tuple $(X_{i.v}, L_i, I_i, O_i, X_{i.r}, X_{i.c})$, where:
  - $X_{i.v}$ is the variable to identify whether service w_i is included in the final solution. When $X_{i.v} = 1$, service w_i remains in sol; when $X_{i.v} = 0$, service w_i is removed from sol.
  - L_i is the variable corresponding to the level where service w_i is located
  - I_i is the $1 \times m$ array representing the input attributes of w_i. $I_{ij} = 1$, when the j^{th} attribute is an input of w_i, otherwise 0.

- O_i is the $1 \times m$ array representing the output attributes of w_i. $O_{ij} = 1$, when the j^{th} attribute is an output of w_i, otherwise 0.
- $X_{i.r}$ is the response time of service w_i.
- $X_{i.c}$ is the execution cost of service w_i.

Objective function

The objective function f is the minimum total cost of the solution.

$$f = \min\{\sum_{i=1}^{n} X_{i.v} \cdot X_{i.c}\} \tag{5.1}$$

Constraints

C is the constraint set that contains all the constraints *sol* needs to satisfy after redundancy removal.

- Initial inputs constraint: X_0 should always be included in *sol* because this dummy service provides the initial inputs of the composition query.

$$X_{0.v} = 1 \tag{5.2}$$

- Goal constraint: X_{n+1} should always be included in *sol* because this dummy service's inputs are the goals and should always be satisfied.

$$X_{n+1.v} = 1 \tag{5.3}$$

- Service invokable constraint: Each service except the dummy service X_0 needs to be invokable in *sol*.

$$X_{k.v} \cdot I_{kj} \leq \sum_{L_i < L_k} X_{i.v} \cdot O_{ij} \tag{5.4}$$

where $k = 1, \ldots, n+1$, $j = 1, \ldots, m$, and $i = 0, \ldots, n$.

- Constraint on the key output parameter: Each real service in the final *sol* has to at least produce one key parameter.

$$X_{k.v} \cdot \sum_{j=1}^{m} \left[O_{kj} \cdot \sum_{L_i > L_k} X_{i.v} \cdot I_{ij} \right] \geq X_{k.v} \tag{5.5}$$

where $k = 1, \ldots, n$ and $i = 2, \ldots, n+1$.

This constraint needs a little explanation. When $X_{k.v} = 0$, service w_k is removed from *sol*. We have $X_{k.v} \cdot \sum_{j=1}^{m} \left[O_{kj} \cdot \sum_{L_i > L_k} X_{i.v} \cdot I_{ij} \right] = 0$ because none of the outputs of w_k is produced.

When $X_{k.v} = 1$, service w_k is not removed from *sol*. We check every output of w_k. For the j^{th} attribute of w_k,

$$O_{kj} \cdot \textstyle\sum_{L_i > L_k} X_{i.v} \cdot I_{ij} \tag{5.6}$$
$$\begin{cases} \geq 1 & \text{if } O_{kj} \text{ at least matches an input of} \\ & \text{services at an upper level } L_i \ (L_i > L_k) \\ = 0 & \text{otherwise} \end{cases}$$

Therefore, if w_k at the level L_k is able to at least produce one parameter which matches an input of services at an upper level L_i $(L_i > L_k)$, we get the following equation

$$X_{k.v} \cdot \sum_{j=1}^{m} \left[O_{kj} \cdot \sum_{L_i > L_k} X_{i.v} \cdot I_{ij} \right] \geq 1$$

- Constraint on response time: The optimal response time of the solution needs to be guaranteed after redundancy removal.

$$\sum_{h=1}^{L_{max}} \max_h \{ X_{i.v} \cdot X_{i.r} | L_i = h \} = Q_r \tag{5.7}$$

where Q_r is the optimal response time of the solution, h denotes a level in the solution and $L_{max} = \max\{L_i | 1 \le i \le n\}$.

Based on the constraints and objective function defined, the redundant service removal model can be formulated:

$$f(sol) = \min\{\sum_{i=1}^{n} X_{i.v} \cdot X_{i.c}\}$$

subject to the constraints (5.2) through (5.7).

When throughput is the first QoS criterion, the variable $X_{i.r}$ is changed into $X_{i.p}$ which corresponds to the throughput of service w_i. Equation 5.7 is replaced with the constraint on the throughput:

$$\min_{h=1}^{L_{max}} \min_{h}\{X_{i.v} \cdot X_{i.p} | L_i = h\} = Q_p \tag{5.8}$$

where Q_p is the optimal throughput of the solution.

5.5 Redundant Service Removal in QoS-aware Service Composition

According to the model in the previous subsection, we can use a standard integer programming solver to find a solution. Under the worst case, the time complexity of the standard solver would be the same as that of the exhaustive search. If the size of the service set in *sol* is n, the complexity is 2^n. In the context of redundancy removal, we will present an algorithm with less time complexity in this subsection.

The intuition behind our algorithm is that we probe the removal of all the combinations of redundant services. For each combination of redundant services, we remove redundant services in the combination and useless services afterwards. We will pick a solution which keeps the same response time (or throughput) and has minimal execution cost. Since the redundant services are normally just part of the total services, our algorithm has less time complexity.

Suppose we are removing redundant services from a solution *sol* with optimal response time, where *sol* contains n real services and $\mathcal{D}_{in}$ (resp. $\mathcal{D}_{out}$) corresponds to service w_0 (resp. service w_{n+1}) located at the lowest (resp. highest) level of *sol*. Algorithm 15 *RedundancyRemoval* is the main algorithm. Initially, we have the optimal response time Q_{opt} (line 1) and the cost-minimized solution *optSol* is supposed to be the original solution *sol* (line 2). Algorithm 16 *FindReduntS* (line 3) searches for all redundant services in *sol*. If there are redundant services, Algorithm 17 *ReduntSolver* (line 5) is called to remove redundant services and useless services in order to find a cost-minimized solution *optSol* whose response time is still optimal.

Algorithm 15: *RedundancyRemoval*(*sol*)

```
1: $Q_{opt}$ is the optimal response time of $sol$;
2: $optSol \leftarrow sol$;
3: $reduntS \leftarrow FindReduntS(sol)$;
4: if $reduntS \neq \emptyset$ then
5:     $optSol \leftarrow ReduntSolver(sol, reduntS, Q_{opt})$;
6: end if
7: return $optSol$;
```

Algorithm 16 *FindReduntS* searches for all redundant services *reduntS* satisfying Proposition 20. L_i denotes the level where service w_i is located in solution *sol*. If all key outputs of service w_i are contained in the union of outputs of other services at level L_j ($L_j \leq L_i$)(line 3), w_i is a redundant service (line 4).

Algorithm 17 *ReduntSolver* removes redundant services and finds a cost-minimized solution with the optimal response time. For each possible combination of redundant services (line 3), a new solution *newSol* is generated by removing redundant services in the combination from *sol* (line 5). Algorithm *RemoveUselessServices* (line 6) searches for useless services in current solution *newSol* and removes these useless

Algorithm 16: $FindReduntS(sol)$

```
1: $reduntS \leftarrow \emptyset$;
2: for $i = 1$ to $n$ do
3:     if $|w_{i.out} \cap (\cup_{L_j > L_i} w_{j.in}) - \cup_{w_j \neq w_i \wedge L_j \leq L_i} w_{j.out}| = 0$ then
4:         $reduntS \leftarrow reduntS \cup \{w_i\}$
5:     end if
6: end for
7: return $reduntS$;
```

Algorithm 17: $ReduntSolver(sol, reduntS, Q_{opt})$

```
1: $optSol \leftarrow sol$;
2: $minCost$ is the total cost of $sol$;
3: for $cmbReduntS \in 2^{|reduntS|} - \{\emptyset\}$ do
4:     $newSol \leftarrow sol$;
5:     $newSol \leftarrow newSol - cmbReduntS$;
6:     $RemoveUselessServices(newSol)$;
7:     $isInvokable \leftarrow CheckInvokability(newSol)$;
8:     if Response time of $newSol = Q_{opt}$ then
9:         $isOpt \leftarrow true$;
10:    else
11:        $isOpt \leftarrow false$;
12:    end if
13:    if $isInvokable = true$ and $isOpt = true$ then
14:        if The total cost of $newSol < minCost$ then
15:            $minCost \leftarrow$ The total cost of newSol;
16:            $optSol \leftarrow newSol$;
17:        end if
18:    end if
19: end for
20: return $optSol$;
```

services from $newSol$. If each service in $newSol$ is invokable, this means $newSol$ can achieve the goals. Thus, $newSol$ is also a solution (line 7). Line 8 checks whether the response time of $newSol$ is optimal. If functional invokable solution $newSol$ with the optimal response time (line 13) has lower execution cost (line 14), $newSol$ becomes the current cost-minimized solution $optSol$ (line 16).

Algorithm 18: $RemoveUselessServices(newSol)$

```
1: $maxL \leftarrow \max\{L_i | w_i \in newSol - \{w_0, w_{n+1}\}\}$;
2: for $l = maxL, \ldots, 1$ do
3:     $setS \leftarrow \{w_i | w_i$ locates at level $l$ of $newSol\}$;
4:     $uselessS \leftarrow \emptyset$;
5:     for $w_i \in setS$ do
6:         if $|w_{i.out} \cap (\cup_{L_j > L_i} w_{j.in})| = \emptyset$ then
7:             $uselessS \leftarrow uselessS \cup \{w_i\}$;
8:         end if
9:     end for
10:    $newSol \leftarrow newSol - uselessS$;
11: end for
12: return
```

Algorithm 19: $CheckInvokability(newSol)$

```
1: for $i = 1$ to $n + 1$ do
2:     if $|w_{i.in} - \cup_{L_j < L_i} w_{j.out}| > 0$ then
3:         return $false$;
4:     end if
5: end for
6: return $true$;
```

Algorithm 18 *RemoveUselessServices* removes useless services from current solution *newSol*. We search for useless services from level $maxL$ that is the highest level where real services are located in *newSol* (line 1). If none of the outputs of service w_i is an input of services at an upper level L_j ($L_j > L_i$) or a goal (line 6), w_i is a useless service (line 7). Line 2 - line 11 removes useless services from *newSol* level by level.

Algorithm 19 *CheckInvokability* checks whether each service is able to be invoked and the solution *newSol* can achieve the goals (line 2).

Theorem 8 *The composite service is correct after redundancy removal.*

Proof 1 *In the redundant service removal model, initial inputs constraint (Equation 5.2), goal constraint (Equation 5.3) and service invokable constraint (Equation 5.4) guarantees the composite service generates the expected outputs for the given* *initial inputs such that the goal is satisfied. Algorithm 19 CheckInvokability implements this functionality of the composite service.*

The complexity of the *RedundancyRemoval* is as Theorem 9. The proof is omitted.

Theorem 9 *The complexity of RedundancyRemoval is* 2^k*, where* k *is the number of redundant services.*

In the worst case, the maximal number of redundant services is the total number of services in a solution *sol*. However, it is impossible that all services in a solution *sol* are redundant. Hence, the number of redundant services k is considerably less than the number of the services in *sol*. The proposed redundant removal algorithm is normally faster than a standard integer programming solver.

Example 20 *We apply Algorithm 15 to Example 11. The objective is to minimize the cost of the solution on the condition that optimal response time is guaranteed.*

Initially, the optimal response time Q_{opt} *of the solution is* 220 *and* $minCost = 240$.

Redundant service set $reduntS = \{w_4, w_5\}$*, because all key parameters generated by* w_4 *and* w_5 *are reproduced. And* $2^{|reduntS|} - \{\emptyset\} = \{\{w_4\}, \{w_5\}, \{w_4, w_5\}\}$ *contains all the possible combinations of redundant services in reduntS.*

Table 5.3 shows the probes to remove each of the combinations. For example, if $\{w_4\}$ *is removed (see row 1), no useless services are generated (row 2). The services in newSol are shown in row 3. Each service in newSol is invokable and newSol can achieve the goals (row 4). The new solution has the same response time 220 (row 5). Execution cost is reduced to 220 (row 6). When* $\{w_5\}$ *is removed,* w_2 *becomes useless. After removal, we can see the new solution has the same response time 220. The execution cost is reduced to 180. We pick this solution* $\{(w_1||w_3); (w_4||w_6); (w_8||w_9)\}$ *as the new optimal solution.*

Table 5.3: Redundancy removal for Example 11

$cmbRedutS$	$\{w_4\}$	$\{w_5\}$	$\{w_4, w_5\}$
$uselessS$	$\emptyset$	$\{w_2\}$	$\{w_2\}$
Services in $newSol$	$\{w_1, w_2, w_3, w_5, w_6, w_8, w_9\}$	$\{w_1, w_3, w_4, w_6, w_8, w_9\}$	$\{w_1, w_3, w_6, w_8, w_9\}$
$checkInvokable$	true	true	false
$optQ(newSol)$	220	220	-
$cost(newSol)$	220	180	-

5.6 Experimental Results

A service in the data sets is provided with response time and throughput. Its execution cost is not available. To deal with this situation, we use the value of throughput as the value of execution cost for a service if removing redundant services from a solution optimized with response time. Conversely, when removing redundant services from a solution with optimal throughput, we take the value of response time as the value of execution cost for a service.

After analyzing the composition results posted at [7], especially the composition results of the first place winner [32] and the second place winner [74], we find redundant services exist in almost all the results for all the data sets. Though all the presented algorithms can find a solution with the correct optimal response time (or throughput), service redundancy is a quite common problem in their solutions. We use the results of the second place winner [74] as the original solutions to test our method. This is because these results contain more redundant services than any other solutions posted at [7].

We compare our method with the redundant service removal method presented in Section 3.5. The redundant service removal method presented in Section 3.5 randomly selects a redundant service to remove and further removes any useless services, as long as the removal does not worsen the QoS value or renders the solution invalid.

Table 5.4: Redundancy removal results for the solutions with optimal response time: our method/random removal method in Section 3.5

		D1	**D2**	**D3**	**D4**	**D5**
Original solution produced by [74]	**Resp. Time**	500	1690	760	1470	4070
	Exec. Cost	126000	322000	80000	562000	432000
	#Services	13	27	11	52	41
Our method	**Keep Resp. Time**	✓	✓	✓	✓	✓
	Exec. Cost	73000	248000	75000	453000	342000
	#Services	8	21	10	42	33
	Redunt. #Services	4	8	1	13	4
Random removal method in Section 3.5	**Keep Resp. Time**	✓	✓	✓	✓	✓
	Exec. Cost	112000	301000	75000	556000	414000
	#Services	5	20	10	40	32

Table 5.5: Redundancy removal results for the solutions with optimal throughput: our method /random removal method in Section 3.5

		D1	**D2**	**D3**	**D4**	**D5**
Original solution produced by [74]	**Throughput**	15000	6000	4000	2000	4000
	Exec. Cost	1810	6300	7490	13600	9720
	#Services	7	24	31	53	41
Our method	**Keep Throughput**	✓	✓	✓	✓	✓
	Exec. Cost	1200	5190	4840	10960	7500
	#Services	5	20	21	40	30
	Redunt. #Services	2	4	8	12	5
Random removal method in Section 3.5	**Keep Throughput**	✓	✓	✓	✓	✓
	Exec. Cost	1730	5350	7290	12880	9330
	#Services	5	20	15	42	32

This method cannot guarantee to find a cost-optimized solution after redundancy removal.

Table 5.4 shows the results over the solutions with optimal response time. The check marks on the row "Keep Resp. Time" mean that the redundancy removal maintains the optimal value. The row "Redunt. # Services" shows the number of redundant services when starting our algorithm. "Redunt. # Services" determines the time complexity. "# Services" shows the number of services in the solutions. Comparing the "# Services" before and after the removal, we can see the removed services count from 9% to 38% of the total services in the original solutions. "Exec. Cost" shows the execution cost of the solution. Noticeably, our method can give better execution cost than the redundant service removal method in Section 3.5. Similarly, Table 5.5 shows the results over the solutions with optimal throughput. Please note that less numbers of services may have the higher execution cost because the execution cost of the services are different. Thus, a solution with less services may cost may than a solution with more services.

5.7 Summary

In this chapter, we study the redundant service removal problem in QoS-aware service composition, where QoS-aware service composition achieves functional goals and QoS optimization at the same time. When execution cost is not used to optimize the solutions, it is possible that the solutions may contain redundant services. Even if execution cost is not the criterion to optimize, it is a better choice to remove unnecessary services from the solution, which reduces the execution cost of the solution. Therefore, we recommend using our method as a last step to further optimize the solution obtained by other methods. We try to solve this problem in the context of redundancy removal, though we can model this problem as yet-another IP problem and use a standard IP solver to solve it. In fact, the proposed redundant removal algorithm is normally faster than a standard integer programming solver.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we study QoS-aware service composition problem that can achieve functional goals and QoS optimization simultaneously. We also study redundant service removal problem which is a derived problem of QoS-aware service composition.

As for the QoS-aware service composition problem, we are motivated to combine a planning algorithm, called GraphPlan, with a systematic search algorithm like Dijkstra's algorithm to achieve both functional goals and QoS optimization at the same time. We propose two methods to combine the GraphPlan with Dijkstra's algorithm.

In the first method, we extend Dijkstra's algorithm from working on a single source graph to working on the extended planning graph whose nodes have multiple sources. The advantage of this method is that this method gets optimal plan with the best QoS value for the single criteria of throughput or response time in polynomial time. However, this method does not provide a uniform graph structure (*i.e.,* an extended planning graph with single or multiple tags) to generate an optimal plan for all kinds of quality criteria.

In the second method, we improve the idea of combining the Graphplan with

Dijkstra's algorithm by providing a uniform graph structure to generate a QoS optimal solution for all kinds of quality criteria. A Layered Weighted Graph (LWG) is converted from an extension of a planning graph. A LWG provides a uniform structure for the ease of using Dijkstra's algorithm to find an optimal plan for all kinds of quality criteria. By using multi-objective shortest path algorithms, this method can be easily extended to solve QoS optimization on multiple QoS criterion for service composition problems.

In the second strategy, we employ three steps to combine the GraphPlan algorithm with Dijkstra's algorithm. First, a Partially Labeled Planning Graph (PLPG) is generated to represent the problem space. The PLPG extends the classic planning graph in the way that each proposition is associated with a label. Next, a Layered Weighted Graph (LWG) is converted from the PLPG. The purpose of the PLPG is to simplify graph reachability for the ease of using Dijkstra's algorithm. In the last step, we use Dijkstra's algorithm on the LWG to obtain a solution with best QoS value. The contribution is our approach can find the global optimal solution for all kinds of QoS criteria. One advantage of our approach is it reduces the possibilities of combinatorial explosion to a large degree when exploring the graph for a best plan. The other advantage is our approach can be easily extended by using multi-objective shortest path algorithms to solve QoS optimization on multiple QoS criteria for service composition problem.

In this thesis, we also study redundant service removal to further optimize the QoS optimal solutions. The removal of redundant services does not worsen the QoS value of the optimal solution. Fewer number of services indicates lower execution costs to invoke these services. A redundant service removal problem is modeled as an optimization problem such that the optimal solution without redundancy is found.

6.2 Future Work

We plan to use multi-objective path traversal algorithms to solve QoS-aware service composition problem based on multiple QoS criterion, since we have proposed to build a Layered Weighted Graph (LWG) for the ease of use Dijkstra's algorithm. The graph reachability in the LWG is the same as it is in an normal graph which means one vertex in the graph can be reachable from any other vertex through one edge rather several vertices through several edges. We plan to start from a comparatively simple case such as to optimize two type of QoS criteria. We can use bi-objective path traversal algorithms. Then, we can use multi-objective path traversal algorithms to study multiple QoS optimization cases for QoS-aware service composition.

Bibliography

[1] Aiello, M., Platzer, C., Rosenberg, F., Tran, H., Vasko, M., Dustdar, S.: Web service indexing for efficient retrieval and composition. In: Proc. of Enterprise Computing, E-Commerce, and E-Services. pp. 63–63 (2006)

[2] Alonso, G.: Web Services: Concepts, Architectures and Applications. Springer Verlag (2004)

[3] Amin, J., Sundararajan, E., Othman, Z.: Cloud computing service composition: A systematic literature review. Expert Systems with Applications 41(8), 3809–3824 (2014)

[4] Armbrust, M., Fox, A., et al., R.G.: A view of cloud computing. Communications of the ACM 53(4), 50–58 (2010)

[5] Bastia, A., Parhi, M., Pattanayak, B.K., Patra, M.R.: Service composition using efficient multi-agents in cloud computing environment. Intelligent Computing, Communication and Devices pp. 357–370 (2015)

[6] Berbner, R., Spahn, M., Repp, N., Heckmann, O., Steinmetz, R.: Heuristics for qos-aware web service composition. In: Proc. of ICWS. pp. 72–82 (2006)

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

[8] Blum, A.L., Furst, M.L.: Fast planning through planning graph analysis. Artificial Intelligence Journal 90(1-2), 225–279 (1997)

[9] Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proc. of ICDE. pp. 421–430 (2001)

[10] Bouguettaya, A., Yu, Q., Liu, X., Malik, Z.: Service-centric framework for a digital government application. IEEE T. Services Computing 4(1), 3–16 (2011)

[11] Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: An approach for qos-aware service composition based on genetic algorithms. In: Proc. of GECCO. pp. 1069–1075 (2005)

[12] Casati, F., Ilnicki, S., Jin, L., Krishnamoorthy, V., Shan, M.C.: Adaptive and dynamic service composition in eflow. Advanced Information Systems Engineering pp. 13–31 (2000)

[13] Casati, F., Sayal, M., Shan, M.C.: Developing e-services for composing e-services. Advanced Information Systems Engineering (2001)

[14] Chan, K., Bishop, J., Baresi, L.: Tech. rep., Dept Computer Science, University of Pretoria (2007)

[15] Committee, O.W.S.B.P.E.L.W.T.: Web services business process execution language version 2.0 - oasis standard (2007), `http://docs.oasis-open.org/wsbpel/2.0/varprop`

[16] Cui, L., Kumara, S., Lee, D.: Scenario analysis of web service composition based on multi-criteria mathematical programming. Service Science 3(4), 280–303 (2011)

[17] Duan, Y., C., N.N., Bo, H., Donghong, L., Feng, F.W., Wencai, D., Junxing, L.: A survey on the categories of service value/quality/satisfactory factors. Computer and Information Science pp. 141–152 (2015)

[18] E., P., Y., R., M., E.P., A, S.: Web service composition methods: A survey. In: Proc. of the International MultiConference of Engineers and Computer Scientists (March 14-16 2012)

[19] Erl, T.: Service-Oriented Architecture (SOA): Concepts, Technology, and Design (2005)

[20] Erl, T.: SOA: Principles of Service Design, vol. 1. Upper Saddle River: Prentice Hall (2008)

[21] Erl, T., Gee, C., et al., J.K.: Next Generation SOA. Prentice Hall (2014)

[22] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved optimization problems. Journal of the ACM (JACM) 34(3), 596–615 (1987)

[23] G., S.T., Vrakas, D., Vlahavas, I.: A survey of service composition in ambient intelligence environments. Artificial Intelligence Review 40(3), 247–270 (2013)

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

[25] Greiner, R., Smith, B., Wilkerson, R.: A correction to the algorithm in reiterâĂŹs theory of diagnosis. Artificial Intelligence 41(1), 79–88 (1989)

[26] Gutierrez-Garcia, J.O., Sim, K.M.: Agent-based service composition in cloud computing. Grid and distributed computing, control and automation pp. 1–10 (2010)

[27] Gutierrez-Garcia, J.O., Sim, K.M.: Agent-based cloud service composition. Applied intelligence 38(3), 436–464 (2013)

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

[29] Hassine, A.B., Matsubara, S., Ishida, T.: A constraint-based approach to horizontal web service composition. In: Proc. of ISWC. pp. 130–143 (2006)

[30] Hewett, R., Kijsanayothin, P., Nguyen, B.: Scalable optimized composition of web services with complexity analysis. In: Proc. of IEEE International Conference on Web Services (ICWS). pp. 389–396 (2009)

[31] Hoffmann, J., Nebel, B.: The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research 14, 253–302 (2001)

[32] Huang, Z., Jiang, W., Hu, S., Liu, Z.: Effective pruning algorithm for qos-aware service composition. In: Proc. of IEEE Conference on Commerce and Enterprise Computing (CEC'09). pp. 519–522 (2009)

[33] Immonen, A., Pakkala, D.: A survey of methods and approaches for reliable dynamic service compositions. Service Oriented Computing and Applications 8(2), 129–158 (2014)

[34] Jiang, W., Zhang, C., Huang, Z., Chen, M., Hu, S., Liu, Z.: Qsynth: A tool for qos-aware automatic service composition. In: Proc. of ICWS. pp. 42–49 (2010)

[35] Jula, A., Sundararajan, E., Othman, Z.: Cloud computing service composition: A systematic literature review. Expert Systems with Applications 41(8), 3809–3824 (2014)

[36] Keller, A., H., L.: The wsla framework: Specifying and monitoring service level agreements for web services. Journal of Network and Systems Management 11(1), 57–81 (2003)

[37] Kil, H.: Efficient web service composition: from signature-level to behavioral description-level. Tech. rep., The Pennsylvania State University, USA (2010), phD thesis

[38] Kil, H., Nam, W.: Anytime algorithm for qos web service composition. In: Proc. of WWW (Companion Volume). pp. 71–72 (2011)

[39] Kornyshova, E., Salinesi, C.: Mcdm techniques selection approaches: State of the art. Computational Intelligence in Multicriteria Decision Making pp. 22–29 (2007)

[40] Kwon, J., Kim, H., Lee, D., Lee, S.: Redundant-free web services composition based on a two-phase algorithm. In: Proc. of IEEE International Conference on Web Services. pp. 361–368 (2008)

[41] L, H.C., Yoon, K.: Multiple Criteria Decision Making. Lecture Notes in Economics and Mathematical Systems, Springer-Verlag (1981)

[42] LaValle, S.M.: Planning Algorithms. Cambridge (2006)

[43] Li, Z., Fang, H., Xia, L.: Increasing mapping based hidden markov model for dynamic process monitoring and diagnosis. Expert Systems with Applications 41(2), 744–751 (2014)

[44] Massam, B.H.: Multi-criteria decision making (mcdm) techniques in planning. Progress in planning 30, 1–84 (1988)

[45] McDermott, D.V.: Estimated-regression planning for interactions with web services. In: Proc. of the 6th international conference on AI planning and scheduling (2002)

[46] McIlraith, S.A., Son, T.C., Zeng, H.: Semantic web services. IEEE intelligent systems 16(2), 46–53 (2001)

[47] Medjahed, B., Bouguettaya, A., Elmagarmid, A.K.: Composing web services on the semantic web. The VLDB journal 12(4), 333–351 (2003)

[48] Narayanan, S., McIlraith, S.A.: Simulation, verification and automated composition of web services. In: Proc. of the 11th international conference on World Wide Web (May 2002)

[49] Newcomer, E., Lomow, G.: Understanding SOA with Web Services (Independent Technology Guides). Addison-Wesley Professional (2004)

[50] Nguyen, X.T., Kowalczyk, R., Phan, M.T.: Modelling and solving qos composition problem using fuzzy discsp. In: Proc. of ICWS. pp. 55–62 (2006)

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

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

[53] Papazoglou, M.P.: Web Services: Principles and Technology. Pearson Education (2008)

[54] Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: a Research Roadmap. International Journal of Cooperative Information Systems 17(2), 223–255 (2008)

[55] Peer, J.: Web Service Composition as AI Planning – a Survey. Tech. rep., University of St.Gallen (2005)

[56] Pistore, M., Traverso, P., Bertoli, P.: Automated Composition of Web Services by Planning in Asynchronous Domains. In: Proc. of ICAPS. pp. 2–11 (2005)

[57] Poizat, P., Yan, Y.: Adaptive composition of conversational services through graph planning encoding. In: Proc. of IsoLA. pp. 35–50 (2010)

[58] Ramasamy, R.K., Chua, F.F., Haw, S.C.: Web service composition using windows workflow for cloud-based mobile application. Advanced Computer and Communication Engineering Technology pp. 975–985 (2015)

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

[60] Schuster, H., Georgakopoulos, D., Cichocki, A., Baker, D.: Modeling and composing service-based and reference process-based multi-enterprise processes. Advanced Information Systems Engineering pp. 247–263 (2000)

[61] Triantaphyllou, E.: Multi-Criteria Decision Making: A Comparative Study. Springer (2000)

[62] Velte, T., Velte, A., Elsenpeter, R.: Cloud Computing, a Practical Approach. McGraw-Hill, Inc. (2009)

[63] W3C: Owl-s: Semantic markup for web services (2004), `http://www.w3.org/Submission/OWL-S/`

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

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

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

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

[68] Wang, J., Korambath, P., Altintas, I., Davis, J., Crawl, D.: Workflow as a service in the cloud: Architecture and scheduling algorithms. Procedia Computer Science 29, 546–556 (2014)

[69] Wikipedia: Beam search. `http://en.wikipedia.org/wiki/Beam_search`, retrieved 2011-09-02

[70] Wikipedia: Dijkstra's algorithm. `http://en.wikipedia.org/wiki/Dijkstra'27_algorithm`, retrieved 2011-06-30

[71] Wu, D., Sirin, E., Hendler, J., Nau, D., Parsia, B.: Automatic web services composition using shop2. Maryland univ college park dept of computer science (2006)

[72] Wu, Q., Zhang, M., Zheng, R., Lou, Y., Wei, W.: A qos-satisfied prediction model for cloud-service composition based on a hidden markov model. Mathematical Problems in Engineering (2013)

[73] Yan, Y., Xu, B., Gu, Z.: Automatic service composition using and/or graph. In: Proc. of CEC/EEE. pp. 335–338 (2008)

[74] Yan, Y., Xu, B., Gu, Z., Luo, S.: A qos-driven approach for semantic service composition. In: Proc. of IEEE Conference on Commerce and Enterprise Computing (CEC'09). pp. 523–526 (2009)

[75] Yang, S., Ma, S.P., Kuo, J.Y., FanJiang, Y.Y.: A survey on automated service composition methods and related techniques. In: Proc. of IEEE Ninth International Conference in Services Computing (SCC) (June 24-29 2012)

[76] Yu, Q., Bouguettaya, A.: Computing service skyline from uncertain qos. IEEE T. Services Computing 3(1), 16–29 (2010)

[77] Yu, Q., Bouguettaya, A.: Efficient service skyline computation for composite service selection. Knowledge and Data Engineering, IEEE Transactions 25(4), 776–789 (2013)

[78] Yu, T., Zhang, Y., Lin, K.J.: Efficient algorithms for web services selection with end-to-end qos constraints. ACM Transactions on the Web (TWEB) 1(1) (2007)

[79] Zeng, L., Benatallah, B., Dumas, M., Kalagnanam, J., Sheng, Q.Z.: Quality driven web services composition. In: Proc. of WWW. pp. 411–421 (2003)

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

[81] Zheng, X., Yan, Y.: An Efficient Web Service Composition Algorithm Based on Planning Graph. In: Proc. of ICWS. pp. 691–699 (2008)

List of Publications

The following publications were made during the course of the thesis study:

[1] **Chen, M.**, and Yan, Y.: QoS-aware Service Composition over Graphplan through Graph Reachability. In: Proceedings of 11th IEEE International Conference on Services Computing (SCC), June, 2014, USA.

[2] **Chen, M.**, and Yan, Y.: Redundant Service Removal in QoS-aware Service Composition. In: Proceedings of 19th IEEE International Conference on Web Services (ICWS), June, 2012, USA.

[3] Yan, Y., **Chen, M.**, and Yang, Y.: Anytime QoS Optimization over the Plan-Graph for Web Service Composition. ACM SAC, March, 2012, Italy.

[4] **Chen, M.**, Poizat P. and Yan, Y.: Adaptive Composition and QoS Optimization of Conversational Services through Graph Planning Encoding. Web Services Handbook 2012, Springer.

[5] Yan, Y., **Chen, M.**: Anytime QoS-Aware Service Composition over the Plan-Graph. Journal of Service Oriented Computing and Applications, June Springer, 2013. http://dx.doi.org/10.1007/s11761-013-0134-6.