VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

School of Engineering and Computer Science
Te Kura Matai Pukaha, Purorohiko

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@ecs.vuw.ac.nz

Using Graph Databases for Automatic QoS-Aware Web Service Composition

Zhaojiang Zhang

Supervisor: Hui Ma

Submitted in partial fulfilment of the requirements for
Bachelor of (Software) Engineering.

Abstract

A Web service is a software component that takes input data and produces output data. With the rapid development of computer network technology, the demand for online services has grown, as has users' expectation that the quality of online services will meet their demands. To complete complex tasks, multiple Web services must be used, and it is necessary to combine, or compose these Web services to provide functions needed by users. The objective of service composition is to find a composite service that meets certain functional requirements and provides the best quality of service (*QoS*), in a challenging environment where the number of available services is rapidly increasing. Existing approaches to service composition suffer from the drawback of requiring excessive computation in in order to find a service composition that meets functional requirements. We propose an automated QoS-Aware Web service composition approach using graph databases to store information taken from service repositories. We will show that our proposed approach can efficiently find service composition solutions.

Acknowledgments

I would like to express my deepest appreciation to all those who supported me in completing this report. I wish to pass on special gratitude to my supervisor, Dr Hui Ma, who provided abundant help, stimulating suggestions, and much needed support, encouragement and guidance during the project and preparation of this report. Special thanks to Alexandre Sawczuk da Silva, who helped me greatly with suggestions and ideas during the project. I also wish to express my gratitude to my and my wife's families for their kind cooperation, encouragement and support which enabled me to complete this project.

Contents

1 Introduction **1**
- 1.1 Aims and Objectives . . . 1
- 1.2 Structure of Report . . . 2

2 Background and Literature Review **3**
- 2.1 Background . . . 3
  - 2.1.1 *QoS*-Aware Web Service Composition . . . 3
  - 2.1.2 *QoS* Properties . . . 4
  - 2.1.3 *QoS*-Aware Fitness Functions . . . 6
  - 2.1.4 Graph Database and Neo4j . . . 6
- 2.2 Literature Review . . . 7

3 The Graph Database-based Web Service Composition Approach **9**
- 3.1 Modelling of Web service Repository . . . 9
  - 3.1.1 Web Service . . . 9
  - 3.1.2 Taxonomy for Inputs and Outputs . . . 9
  - 3.1.3 InputServices and OutputServices . . . 10
  - 3.1.4 Web Service Dependencies . . . 10
- 3.2 Generating a Graph Database for a Service Repository . . . 10
  - 3.2.1 Preprocessing of the Service Repository . . . 11
  - 3.2.2 Creation of Web Service Nodes . . . 12
  - 3.2.3 Creation of Web Service Dependencies . . . 13
- 3.3 Generate a Graph Database for a Given Task . . . 16
- 3.4 Generating Web Service Compositions . . . 16
- 3.5 *QoS*-Aware Service Composition . . . 18
- 3.6 Summary . . . 20

4 Evaluation Design **21**
- 4.1 Datasets and Parameters . . . 21
- 4.2 Evaluation Results . . . 21
  - 4.2.1 Effectiveness of the Reducing Algorithm . . . 21
  - 4.2.2 Correctness of the Web Service Composition . . . 22
  - 4.2.3 Evaluation Results for QoS-Aware Service Composition . . . 23
  - 4.2.4 Summary . . . 25

5 Conclusion and Future work **27**

A Bibliography **29**

Chapter 1

Introduction

Service-Oriented Architecture (SOA) [23] is an architectural style for building software applications which uses services available in a network. SOA is realised through a standards-based technology called Web services, which allows coupling between Web services, so they can be reused. A Web service is a self-contained unit with limited functionality that takes input data and produces output data [14]. To provide value-added functions, it is necessary to compose Web services to provide powerful service functions. The result of such composition is to take a set of input data provided by the user and create a corresponding set of output data needed by a user.

Currently there are three main approaches to Web service Composition. The first group of approaches includes several those which employ traditional methods such as Integer Linear Programming (ILP) [34] to solve the problem of Web service composition. These approaches are lack of scalability and are no longer efficient since the number of Web services is increasing extraordinarily quickly. The second group includes various Evolutionary Computing (EC) approaches, such as Genetic Algorithms (GA) [3], Genetic Programming (GP) [12, 13, 24] and Particle Swarm Optimisation (PSO) [5, 11]. These approaches are slow, especially when checking the dependencies between the component services, a task which requires a substantial amount of resources. The third type is the graph based approach, such as [9, 29]. This approach temporarily stores graph dependencies in memory rather than permanently saving them on local storage.

1.1 Aims and Objectives

The aim of this project is to propose an efficient automated QoS-aware Web service composition approach using Graph Databases. Existing Web service composition approaches [13, 33] do not permanently store Web service dependencies, which means that when running the composition algorithm for different tasks, the algorithm will keep Web service dependencies in memory. The problem with this is that when a new task needs to be carried out, it becomes necessary to regenerate its Web service dependencies. This is because different tasks have different enquire (*input*) and result (*outputs*) data, upon which Web service dependencies are based. In practice, this behaviour dramatically increases the cost of running the system, since Web service dependencies which need only be generated once may in fact be generated several times. Moreover, the existing approach requires the generation of a corresponding workflow after each composition has been established. This takes a substantial amount of time and slows down the entire process.

Graph databases store data and relationships [16] between data as graphs. In this project we employ a graph database to store the dependencies between services in a service repos-

itory and our service composition approach is based on the use of these graph databases. This project aims to create a graph database-based approach that generates Web service solutions efficiently by reducing the composition costs involved in checking dependencies of services.

To achieve the aim of the project, the project is conducted with the following objectives:

1. To review existing works in literature on this topic.
2. To use a graph database to model and store services and their dependencies within a service repository.
3. To generate non-QoS-aware Web services compositions with no redundant services in the compositions.
4. To select service compositions with the best QoS.
5. To conduct a full evaluation of our approach by comparing the performance of our proposed approach with one of the existing approaches.

1.2 Structure of Report

The remainder of this report is organised as follows: A background and literature review is presented in Chapter 2. Chapter 3 gives a description of our Graph database-based approach. Chapter 4 covers our evaluation to compare the performance of our approach with an existing approach. Chapter 5 presents our conclusions regarding our approach, and proposes possible future developments regarding our approach.

Chapter 2

Background and Literature Review

2.1 Background

2.1.1 *QoS*-Aware Web Service Composition

Web services [1] are modular, distributed, self-contained, dynamic applications that are published on the Web in order to be available to users. They follow certain technical standards such as Simple Object Access Protocol (SOAP) [1] for accessing Web services, Web Service Description Language (WSDL) [1] for describing Web services, and Universal Description, Discovery and integration (UDDI) [2] for describing, publishing, and finding web services.

Web Services can be called by internet software and can be reused over and over by many applications. In essence, a web service is any piece of software that makes itself available over the internet [14].

Example 2.1.1. If I post pictures on Flickr about various steps needed to build my house, I may wish to dynamically display certain pictures. However, the relevant images are not stored on my local storage, but on Flickr. In this case I can make a dynamic request to ask Flickr to provide me with an image, since Flickr provides such a Web service. This is the example.

However, it is important to note that a single Web service may only provide limited functionality, which is not sufficient to respond to the user's request. In order to complete complex tasks, a range of Web services are required, and it is necessary to combine, or compose these Web services to provide new value-added and complex functionality [32]. When a user's request cannot be fulfilled by any available web service then Web Service Composition is required to fulfill the user's request.

Web service composition is the process of combining several Web services into a more complicated and powerful service. Since there is no uniform definition, different researchers have defined the Web service composition problem from different perspectives [10, 25, 30, 31]. In [31], from the point of view of business processes, Web service composition is regarded as the organic connection of services according to certain business rules, so that companies can cooperate with each other to reach certain established business goals. In [25], from the perspective of application integration, Web service composition is the process of seamlessly integrating heterogeneous information systems and software from different enterprises, eliminating information silos and forming interconnected software consortia. In [10], from the point of view of problem solving, Web service composition is considered to be a way to achieve user-specific objectives, and a combination service satisfying this goal is found in a given set of services. In [30], from the point of view of task planning, service

composition is a process of decomposing a large task into several subtasks, and breaking down these subtasks into even more smaller subtasks.

This project considers Web service composition to be the process of synthesizing several Web services when a single Web service can not meet a user's requirements, so as to form a large-scale composite service with internal process logic.

Nowadays, a large number of Web services on the Internet provide the same or overlapping functionality but present different non-functional characteristics. These non-functional characteristics are called Quality of Service (*QoS*) properties, such as response time, execution cost, availability, reliability etc. Each of those properties are described in detail in the next section. Because *QoS* properties have become the most commonly used set of characteristics for measuring the quality of Web services, meeting global *QoS* requirements while fulfilling various functional requirements is referred to as QoS-*aware Web service composition* [21, 32].

2.1.2 *QoS* Properties

Quality of service (*QoS*) is one of the important factors to consider when composing services. It defines the non-functional requirements of a service, such as response time, execution cost, availability, reliability etc. Good quality of service means achieving certain *QoS* goals which are termed *QoS* values or *QoS* properties. These *QoS* properties indicate whether a Web service is reliable, trustworthy or efficiency. Their significance stems from the fact that a Web service may be functionally capable of performing a given task, but might not be reliable or efficient enough to achieve users' satisfaction. Web services are usually rated using multiple *QoS* values, each value representing an aspect, or *QoS* property, of the Web service. *QoS* properties are the most commonly used characteristics for measuring the quality of Web services and even composite services, as they indicate whether a service is capable of meeting users' expectations.

To model the performance of service compositions we considered four *QoS* attributes: *availability, reliability, execution cost* and *response time*. We chose these because they are commonly used in this field [4, 12, 20, 33]. According to [22, 33, 35], the four above-mentioned *QoS* properties are defined as follows:

Execution Cost is the amount of money that a service requester has to pay for using the Web service. The global *execution cost* C_t of a composite set of services can be treated as the sum of the execution costs of all the operations invoked by the services s used.

$$C_t = \sum_{i=1}^{|s|} C_i \tag{2.1}$$

Where$|s|$is the number of services in the composition.

The *response time* of a single task is the time which elapses between sending a task request and receiving a response. Web service composition allows services to execute in parallel. Thus, when considering a step where two tasks execute in parallel, the path with the longest *response time* is chosen to calculate the time taken for that step.

$$T_i = \begin{cases} \sum_{i=1}^{|s|} T_i & \text{if sequential pattern} \\ \max_{i=1}^{|s|} T_i & \text{if paraller pattern} \end{cases} \tag{2.2}$$

Availability is defined as the ratio of (1) the time during which a service is ready for use and (2) the time during which the Web service exists. The global availability A_t of a

composite Web services can be computed as the product of the local availabilities of the Web services s used in the service composition.

$$A_t = \prod_{i=1}^{|s|} A_i \tag{2.3}$$

Reliability is the how reliable message delivery is to the Web service within the maximum permitted time frame. Global *reliability* R_t can be calculated as the product of the local reliabilities of the Web services s used in the service composition.

$$R_t = \prod_{i=1}^{|s|} R_i \tag{2.4}$$

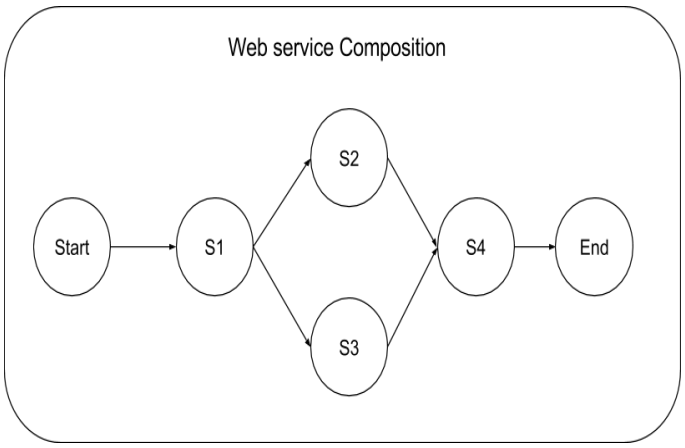

Figure 2.1: Example Web service composition

Example 2.1.2. For example, for the service composition depicted in Figure 2.1, we can calculate the *QoS* of the composition service using the above formulas as follows:

Execution cost

The total *execution cost* of the composite service can be calculated as:

$$C_t = Cost(S_1) + Cost(S_2) + Cost(S_3) + Cost(S_4)$$

Response time

In calculating *response time*, it must be noted that, as shown in Figure 2.1, services S_2 and S_3 can be executed in parallel. The *response time* depends on S_2 and S_3's local *QoS* duration property values. If S_2's *QoS* duration property value is greater than S_3's *QoS* duration property value, then the overall *response time* of the composite service can be computed as the sum of all three service nodes S_1, S_2 and S_4's local *QoS* duration property values.

$$T_t = Time(S_1) + Time(S_2) + Time(S_4)$$

Availability

The *availability* of the composite service, can be computed as:

$$A_t = Availability(S_1) \times Availability(S_3) \times Availability(S_2) \times Availability(S_4)$$

Reliability

The *reliability* of the composite service in Figure 2.1, can be computed as:

$$R_t = Reliability(S_1) \times Reliability(S_3) \times Reliability(S_2) \times Reliability(S_4)$$

2.1.3 *QoS*-Aware Fitness Functions

To evaluate the quality of a Web service composition we need a suitable fitness function. A fitness value can be used to reflect the global quality of service of a service composition. Following the common practice [11, 12, 33] used in multi-criteria optimization problems, we normalize the values of each *QoS* property and restrict them to the interval [0,1]. We choose the same objective function as the one used in [11, 33] for our approach. The fitness for candidate i is defined as follows:

$$Objective_i = (w_1 \times A_i) + (w_2 \times R_i) + (w_3 \times T_i) + (w_4 \times C_i) \tag{2.5}$$

where A_i, R_i, T_i and C_i denote the normalized availability, reliability, execution cost and execution time of candidate i, and weights w are real positive numbers where $w_1 + w_2 + w_3 + w_4 = 1$. The value of weights assigned by the user to the objective function represents the importance of the different *QoS* properties for the Web service composition.

To achieve normalization we identify the minimum and maximum values of all the composition solutions discovered so far. We then use those values to normalize the *QoS* properties according to the following formula [33]:

$$A_i = \begin{cases} \dfrac{A_i - A_{min}}{A_{max} - A_{min}} & \text{if } A_{max} - A_{min} \neq 0 \\ 1 & \text{if } A_{max} - A_{min} = 0 \end{cases} \tag{2.6}$$

$$R_i = \begin{cases} \dfrac{R_i - R_{min}}{R_{max} - R_{min}} & \text{if } R_{max} - R_{min} \neq 0 \\ 1 & \text{if } R_{max} - R_{min} = 0 \end{cases} \tag{2.7}$$

$$T_i = \begin{cases} \dfrac{T_{max} - T_i}{T_{max} - T_{min}} & \text{if } T_{max} - T_{min} \neq 0 \\ 1 & \text{if } T_{max} - T_{min} = 0 \end{cases} \tag{2.8}$$

$$C_i = \begin{cases} \dfrac{C_{max} - C_i}{C_{max} - C_{min}} & \text{if } C_{max} - C_{min} \neq 0 \\ 1 & \text{if } C_{max} - C_{min} = 0 \end{cases} \tag{2.9}$$

Normalized *QoS* property values close to *1* indicate better quality, while normalized values closer to *0* indicate poorer quality. Finally, we use the normalized *QoS* properties and weights to calculate the fitness of the individual candidates, where the largest fitness value will be chose as the best Web service composition out of that set of candidates.

2.1.4 Graph Database and Neo4j

A graph database is a type of *NoSQL* database, which uses graph theory to store, map and query relationships. Graph databases contains *nodes* and *edges* components [28]. They are widely used to model social networks and also used to solve real-world problems. *Transportation* [6], *protein-interaction* [7] and even *business networks* [26] can naturally be modeled as graphs. Paper [28] evaluates four current types of graph databases (*Neo4j* [16], *OrientDB* [17], *Titan* [18] and *DEX* [19]) from a performance point of view. The results show that the *Neo4j* graph database gets the best performance with *loading datasets*, *finding the shortest path* and *breadth-first exploration of nodes neighbourhood*. Good traversal performance reduces the time taken for Web service composition.

Because of its good performance, we employed a *Neo4j* [16] graph database for our project. We also chose it because it is suitable for representing connected and directed Web

services and it allows for very fast retrieval, traversal and navigation of data. Our project used a *Neo4j* graph database to store all services as *nodes* and dependencies between the services as *relationships* (edges). We gave all *nodes* and *relationships* their own individual properties. And for each data set we needed to create a graph database only once, after which we were able to use the database to solve various composition problems.

2.2 Literature Review

Over the past few years, Web services have become widely used. A large number of complex applications can be developed using compositions of existing services. Automatic Web service composition technology has become a major focus and challenge in building robust applications which make use of distributed services which provide various functions [27]. Many approaches have been developed for Web service composition among which graph-based approach shows its promise in generating feasible composition solutions efficiently, but only a few of them use the concept of graph theory. In this section, we will present a brief overview of some graph-based techniques that deal with automatic Web service composition.

Da Silva et al. presents in [13] an evolutionary computation technique that performs fully automated Web service composition using graph representations for solutions. There are two steps. The first step is to initialise the population by employing a graph building algorithm based on the planning graph approach described in composition literature [15]. The second step is to perform mutation and crossover operations on selected candidates, to generate a new set of candidates and evaluate the fitness of those candidates.

The drawback of the approach in [13] is that the graph mutation space contains many invalid service compositions that either violate the constraints of service dependencies or deteriorate the overall performance. Another disadvantage is that the system only keeps the Web service dependencies in memory. With every new service request, this approach needs to regenerate all the dependencies again which incurs expensive computation costs. Furthermore, the cost of executing the graph building algorithm is high, since the graph building algorithm is employed twice, in two different processes, namely in the process of generating initial populations and the process of performing mutation and crossover operations.

Seyyed et al. [29] uses a graph search algorithm to construct Web service compositions. This algorithm is based on input-output dependencies of Web services. In order to solve the service composition problem, the authors divide the process into two steps. The first step is to look for Web services which can potentially participate in the composition, and the second step is to find a corresponding composition. The graph building algorithm constructs edges between directly related Web services which have data dependencies between the inputs and outputs of these services. The main shortcoming of this approach is that semantic functions are not considered in the dependencies between input and output parameters. Thus there is no guarantee that a generated composite service will provide the precise functionality requested by the user.

The Web service composition approaches proposed by Da Silva et al. in [13] and Seyyed et al. in [29] also share a common drawback. They do not include quality of service in their approach. This means the resulting Web services may not perform tasks that meet a user's non-functional requirements.

Jing et al [21] developed a relational-database approach to constructing Web service compositions. This approach overcomes a major limitation of most automated service composition methods [13, 29], which was to store Web services and dependencies between them

only in memory. This approach stores Web services and dependencies between the Web services in a database instead of memory. The authors propose an algorithms that builds a path between services in the service repository. According to their method, each path has a PathID, and input and output values. After all paths have been generated, they are then stored in the database. To handle the task requests, a query must be sent to the database to find Web service composition paths which satisfy the request. The next step involves ranking and filtering the paths returned from the algorithms according to their *QoS* value. The top rank will be the best composition for a given task.

One of the major drawbacks of the approach in [21] is that it cannot efficiently handle a very large service repository. The proposed method is slow, as it preprocesses all the Web services in advance to generate paths between services which it stores in the database. However, if any service inputs or outputs have changed or any new Web services have been added, this will affect most of the paths stored in the database, since they are interconnected each other. In addition, many services offer identical or overlapping functionalities.

The result is that the number the paths in the database is enormous. And all the paths need to be processed in order to find the PathID and process the input and output of the each path in the related paths. So when creating web service compositions this greatly downgrades overall performance. The third drawback of the approach in [21] is that it only considers the response time property when dealing with *QoS*-aware Web service composition and does not consider execution cost, availability and reliability of the composition. This means the solution may not be the optimised solution.

As we have seen, existing graph-based [29] and GP based [13] approaches need to build a data dependency graph for each given task. However, database dependencies between services in a service repository remain stable for all service tasks. Therefore, once a service dependency graph is generated it should be stored, so it can be used for all service tasks. In this project, we propose to use a graph database to store the information related to services and dependencies of services of a service repository. Once a graph database has been created using information in the service repository, all the services and dependencies of the services are permanently stored on local storage, and a Graph database like Neo4j, which has built-in path-finding methods, is used. This is different to the relational-database approach in [21], where there is a need to generate all the paths between services and then store those paths in the database. With each new service task, our approach can utilise existing dependency information contained within graph database dependency graphs. Our approach also takes the quality of service (*QoS*) into consideration by including *QoS* properties with each edge between Web services.

In summary, most current non-database-type automated Web service composition methods [13, 29] are in-memory methods, which means Web service composition results, details of services, and dependencies between services are stored locally in simple text format. This means the reusability of the solution is very low, as most information is not stored, especially in the case of service dependencies and services are not reusable after memory is cleared. The relational-database-based approach [21] uses stored services with dependencies between services stored in the database as paths, but the performance of SQL queries on the large resulting database as well as database maintenance is very low.

Chapter 3

The Graph Database-based Web Service Composition Approach

This report focuses on the the use of graph databases to model and store a service repository, with dependencies between the Web services, and service compositions. Our graph based approach consists of four steps:

1. Creating graph database for a web service repository: We use all available services in the service repository to create a graph database, with Web services as *nodes* and dependencies between services as *edges*.

2. Generating a graph database for a given task: We select all the services related to the given task inputs and outputs, then use related services to generate a graph database.

3. Generating initial population of service composition solutions: We use the graph database we generated in step 2 and the algorithm we designed for this project to produce an initial population of service composition solutions.

4. Selecting a service composition solution from the population with the best overall *QoS*: We apply the fitness function to the initial population to find the solution with best overall *QoS*.

The overall design of our system is shown in Figure 3.1, following with some design details.

3.1 Modelling of Web service Repository

3.1.1 Web Service

In this project we model a Web service as an entity which has a *name*, and has a set of properties *ID, QoS, inputs, outputs, inputServices, and outputServices*. The first step is to model service repositories, which contains a set of services, between which there are data dependencies. Data dependency refers to the relationships between services. Each service takes inputs, produce outputs, and has *QoS*. In the following description of our approach we provide details of this model.

3.1.2 Taxonomy for Inputs and Outputs

In our approach we used a 'taxonomy tree' to represent the relationships between the inputs and outputs of services. In the particular taxonomy tree we used, any two concepts A and

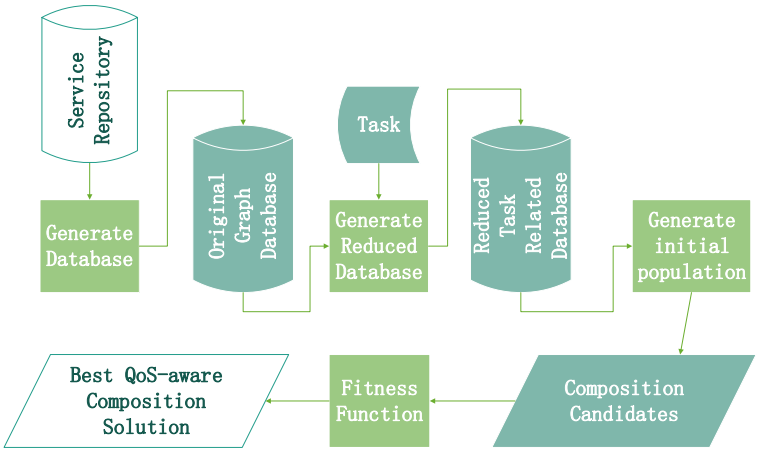

Figure 3.1: Overall system design

B can be related to each other in one of four possible ways. The first scenario is that *A* is a generalization of *B*. The second scenario is that *A* is a specialization of *B*. The third scenario is that *A* and *B* are not related to each other. And the last scenario is that *A* equals *B*.

Example 3.1.1. The Web service '*FindHotel*' uses a single input parameter (*cityName*) represented by the concept of '*CITY*', which belongs to the taxonomy tree. Therefore, '*FindHotel*' requires an instance of the concept '*CITY*'. The only output parameter (*Hotel*) is represented by an instance of the concept '*Hotel*' that also exists in the taxonomy tree. The *inputs* and *outputs* of each Web service are clearly defined in terms of concepts belonging to the specialized taxonomy tree.

3.1.3 InputServices and OutputServices

As shown in Figure 3.3, within a database, Web services are modelled as having a set of *InputServices*, which is the set of Web services connected to the input side of the Web service, and *outputServices* which is the set of Web services connected to the output side of the service. The reason that we add these two properties is to make it easier to retrieve the connected Web services and also to reduce the cost of creating relationships between Web services.

3.1.4 Web Service Dependencies

Each relationship is a directed edge that connects the output of a Web service node to the input of another node according to already established dependencies stored in the taxonomy tree. Like Web service nodes, relationships also have properties. They have a *from* node, a *to* node and direction. Relationships between Web service nodes are an essential part of a graph database. They allow us to find related Web services.

3.2 Generating a Graph Database for a Service Repository

There are three steps involved in generating our database using the components described in the previous section. Firstly, preprocessing takes place to find all the services' properties,

and create a Web service object for each service in the service repository. Secondly, creating the graph database Web service nodes using the web service objects we created in step 1. We accomplish this by assigning all the Web service objects' fields to graph database Web service node properties. Lastly, generating the graph database relationships based on the Web service nodes' input services and output services. The following sections will introduce the design in greater detail.

3.2.1 Preprocessing of the Service Repository

Before generating a graph database, we need to preprocess the service repository by creating a Web service object for each Web service from the service repository. We apply *Algorithm 1* to find all the input services and output services for each taxonomy node.

Algorithm 1: Populates the taxonomy tree by associating services with the nodes in the tree.

```
   Input  : taxonomyNodes, serviceNodes
   Output: taxonomyNodes
1: i ← 0;
2: while i < |taxonomyNodes| do
3:     tNode ← taxonomyNodes[i];
4:     tNode.parents ← findParentsNodes(tNode);
5:     tNode.children ← findChildrenNodes(tNode);
6:     i ← i + 1;
7: i ← 0;
8: while i < |serviceNodes| do
9:     j ← 0;
10:    outputs ← serviceNodes[i].outputs;
11:    while j < |outputs| do
12:        tNode ← findTaxonomyNode(outputs[j]);
13:        k ← 0;
14:        while j < |tNode.parents| do
15:            tNode.parents ← tNode.parents ∪ {serviceNodes[i]};
16:            k ← k + 1;
17:        j ← j + 1;
18:    j ← 0;
19:    inputs ← serviceNodes[i].inputs;
20:    while j < |inputs| do
21:        tNode ← findTaxonomyNode(inputs[j]);
22:        k ← 0;
23:        while j < |tNode.children| do
24:            tNode.children ← tNode.children ∪ {serviceNodes[i]};
25:            k ← k + 1;
26:        j ← j + 1;
27:    i ← i + 1;
```

Example 3.2.1. Figure 3.2 shows an example of a taxonomy tree. Service *1* has inputs $I_1 = \{A\}$ and outputs $O_1 = \{B\}$, service *2* has Inputs $I_2 = \{E\}$ and outputs $O_2 = \{B\}$, and service *3* has Inputs $I_3 = \{B\}$ and outputs $O_3 = \{F\}$. *Algorithm 1* searches services inputs and outputs of services *1*, *2* and *3*. Each value in inputs and outputs is a node in the taxonomy tree, which we call a taxonomy node (*tNode*). Each *tNode* has input services (*IS*)

and output services (*OS*) fields; then we add Web services into corresponding *IS* or *OS* field. Taxonomy is represented as a tree, and each *tNode* can have a parent node or child node. For example, 'CITY' *tNode* may have parent node called 'COUNTRY' and child node called 'HOTEL'. When we add Web service into *IS*, we also add a Web service into all the *tNode*'s parent node's *IS* field. The same applies to adding Web service into *OS*, in which case we also add Web service into all the *tNode*'s children node's *OS* field. After we run *algorithm 1*, we obtain $IS_A = \{service1\}$, $OS_A = \{service1, service2, service3\}$, $IS_B = \{service1, service3\}$, $OS_B = \{service1, service2\}$, $IS_E = \{service1, service3\}$, $OS_E = \{Null\}$, $IS_F = \{service1\}$ and $OS_F = \{service3\}$. This is shown in Figure 3.2.

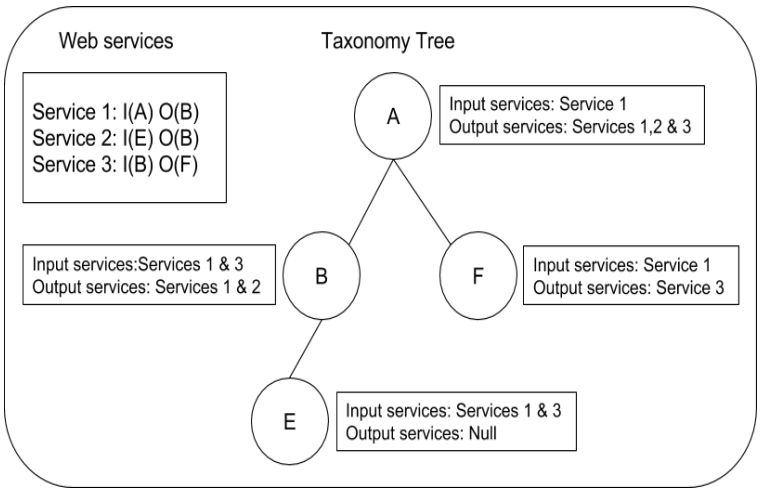

Figure 3.2: Taxonomy tree Example

We then apply *Algorithm 2* to assign input services and output services for each Web service object based on Web service object's *inputs* and *outputs*. It goes through each input value in each Web service (*WS*) in a service repository and finds a corresponding taxonomy node *tNode* and then adds all the input services into the *WS*'s *InputServices* field. It also adds all the input services found in all parent taxonomy nodes into *WS*'s *IS* field. Additionally, it checks each output value in each *WS* in a repository, and finds corresponding *tNode* , adding all the output services into the *WS*'s *OS* field, and all the output services found in all parent taxonomy nodes into *WS*'s *OS* field.

Example 3.2.2. After we run the *algorithm 2*, we get: Service *1*: $IS_1 = \{service1, service2, service3\}$ and $OS_1 = \{service1, service3\}$, Service *2*: $IS_2 = \{Null\}$ and $OS_2 = \{service1, service2\}$, Service *3*: $IS_3 = \{service1, service2\}$ and $OS_3 = \{service1\}$, shown in Figure 3.3.

3.2.2 Creation of Web Service Nodes

Each Web service is a node in the graph database, and each node contains list of properties. In this section we will introduce how to create graph database Web service nodes using Web service objects we produced during the service repository preprocessing. We use the graph database *createnode* method to generate a new node and we use the *setproperty* method to assign properties to the node. We also create an index for each Web service node for querying the Web service node by its service name when needed. After service nodes have been created, they are stored in the graph database permanently. Figure 3.4 is an example of a graph database - it shows the Web service nodes created, still without relationships between the Web services. The data set used in this example is WSC2008-01.

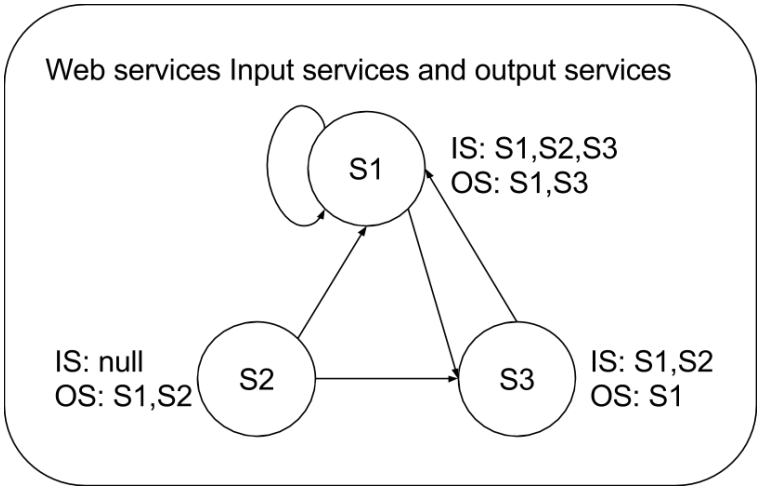

Figure 3.3: Input services and out services

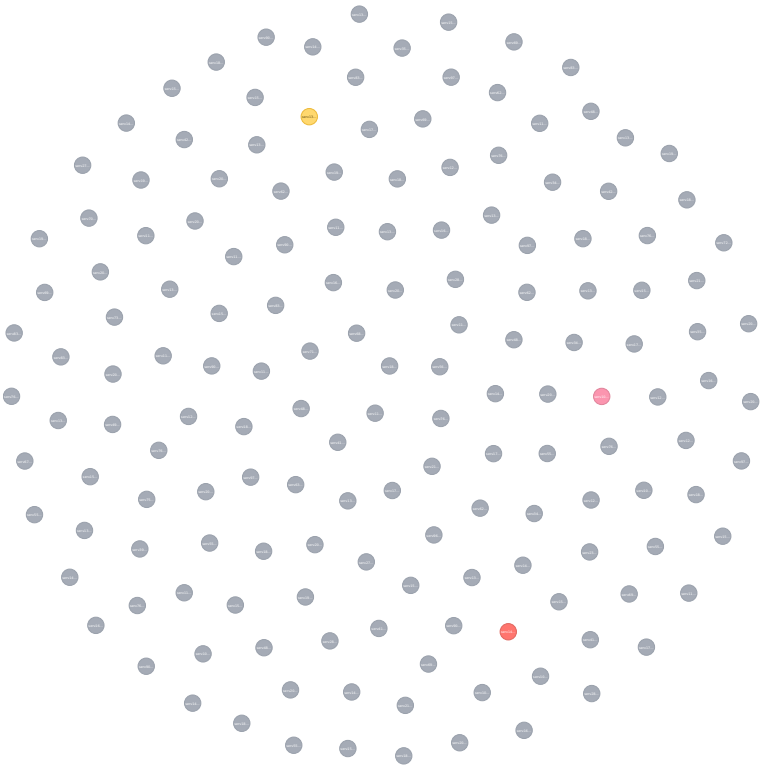

Figure 3.4: Graph database without relationships (WSC 2008-01)

3.2.3 Creation of Web Service Dependencies

To store Web service dependencies in a database, we need to traverse each and every Web service node in the graph database, find all the *input services* and *output services* and then use the *createRelationshipTo* method to create dependencies between the Web service nodes. *Algorithm 3* handles the creation of relationships between Web services. It goes through each Web service node (N), and loops through the corresponding *inputServices* of each N, and then adds relationships between the Web services according to the sets of *inputServices*. *Algorithm 3* handles the process of creating relationships between Web services.

Algorithm 2: Add correspondence input services and output services to each Web service.

Input : $serviceNodes$
Output: $serviceNodes$

$i \leftarrow 0$;
while $i < |serviceNodes|$ **do**
  $sNode \leftarrow serviceNodes[i]$;
  $sNode.inputServices \leftarrow \{\}$;
  $sNode.outputServices \leftarrow \{\}$;
  $j \leftarrow 0$;
  while $j < |sNode.outputs|$ **do**
    $tNode \leftarrow findTaxonomyNode(sNode.outputs[j])$;
    $k \leftarrow 0$;
    while $k < |tNode.outputServices|$ **do**
      $sNode.outputServices \leftarrow sNode.outputServices \cup \{tNode.outputServices[k]\}$;
      $k \leftarrow k + 1$;
    $j \leftarrow j + 1$;
  $j \leftarrow 0$;
  while $j < |sNode.inputs|$ **do**
    $tNode \leftarrow findTaxonomyNode(sNode.inputs[j])$;
    $k \leftarrow 0$;
    while $k < |tNode.inputServices|$ **do**
      $sNode.inputServices \leftarrow sNode.inputServices \cup \{tNode.inputServices[k]\}$;
      $k \leftarrow k + 1$;
    $j \leftarrow j + 1$;
  $i \leftarrow i + 1$;

Algorithm 3: Create relationships between Web services.

Input : $GraphDatabaseWithNoRelationships$
Output: $GraphDatabaseWithRelationships$

$i \leftarrow 0$;
while $i < |serviceNodes|$ **do**
  $sNode \leftarrow serviceNodes[i]$;
  $j \leftarrow 0$;
  while $j < |sNode.inputServices|$ **do**
    $inputSNode \leftarrow sNode.inputServices[j]$;
    $relation \leftarrow inputSNode.createRelationshipTo(sNode)$;
    $relation.setProperty(\text{"From"} : inputsNode)$;
    $relation.setProperty(\text{"To"} : sNode)$;
    $relation.setProperty(\text{"Outputs"} : inputsNode.outputs)$;
    $relation.setProperty(\text{"Direction"} : incoming)$;
    $j \leftarrow j + 1$;
  $i \leftarrow i + 1$;

Example 3.2.3. Figure 3.5 is an example of a graph database before reduction. The data set used in this example is WSC 2008 - 01, which contains is a total of 158 Web services. All the nodes represent the Web services and all the edges are the relationships between Web

services. Once you click a node or an edge from the graph database, the properties of that node or edge appear at the bottom of the page. Figure 3.6 is an example of the properties that appear when you click the node Web service serv7231183.

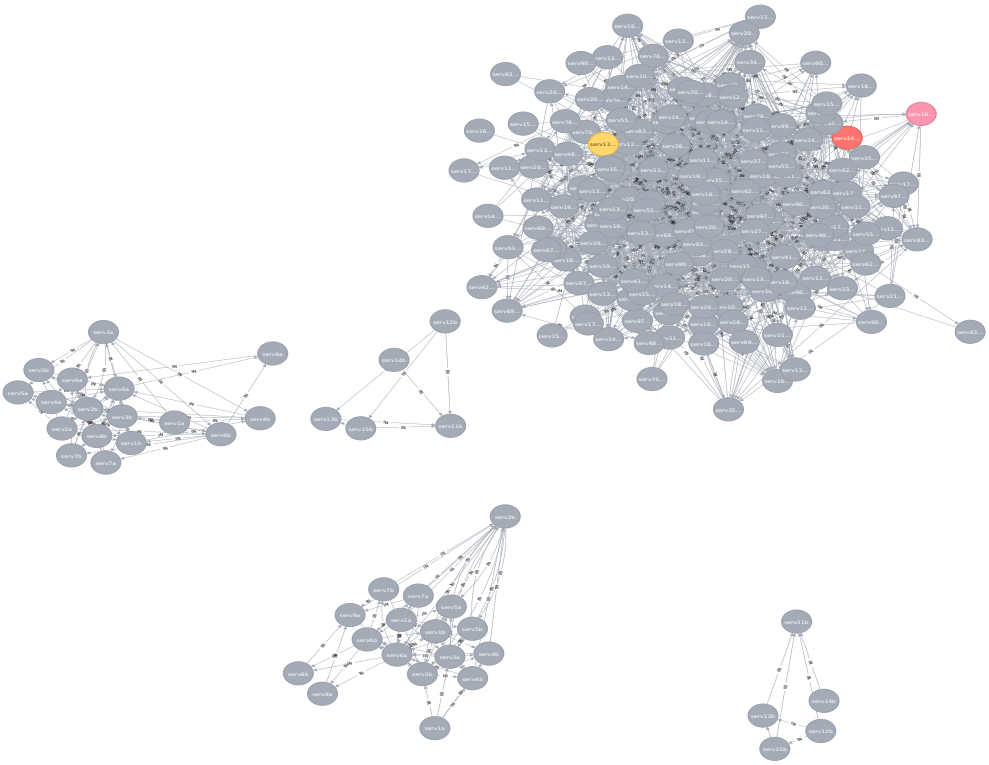

Figure 3.5: Graph database before reduction (WSC2008-01)

Figure 3.6: The properties of Web service serv7231183 (WSC2008-01)

3.3 Generate a Graph Database for a Given Task

For each given task which is described by $\{inputs, outputs\}$, only a subset of services in a repository is related to that task. Therefore we created a graph database for each given task. This section presents an algorithm that generates a reduced graph for a given task, as shown in *Algorithm 4*.

Algorithm 4: Reduce Graph Database.

Input : $GraphDatabase$
Output: $ReducedGraphDatabase$

```
1:  i ← 0;
2:  relatedNodes ← {};
3:  while i < |serviceNodes| do
4:      sNode ← serviceNodes[i];
5:      if hasRelationship(sNode, startNode) ∧ hasRelationship(sNode, endNode) then
6:          relatedNodes ← relatedNodes ∪ {sNode};
7:      if !fulfillInputs(sNode) then
8:          removeRelatedNodes(sNode);
9:      i ← i + 1;
10: return relatedNodes;
```

Our algorithm first creates a start node S and an end node E. The start node S contains the inputs of the task and the end node E contains the outputs of the task. Secondly, we insert these nodes into the database by creating new relationships between start node S and all the nodes that use S's input. Similarly, we create relationships between end node E and all the nodes that can output data to node E. Lastly, we find all services that from a path between S and E and remove all services that are not in those paths. *Algorithm 4* shows the process of reducing the original graph database by removing all superfluous nodes.

Example 3.3.1. The following example illustrates the effectiveness of our proposed algorithm. Applying our graph database reduction algorithm (*Algorithm 4*) to the graph database shown in Figure 3.3 leads to the new reduced graph database shown in Figure 3.7. This new reduced graph database contains only Web services which are related to both task *input* and task *output*. In this example, the procedure reduces the number of Web services in the graph database from 158 to 61.

3.4 Generating Web Service Compositions

Our reduced graph database algorithm allows us to retrieve all the Web service nodes related to a particular task. This section explains how we used related Web service nodes to generate Web service compositions in order to create an initial set of candidates.

Algorithm 5 is designed to create feasible compositions. It starts from the end node and searches internal nodes which are directly connected to the end node. Internal nodes are added into the composition list if they fulfill the input of the end node. Then the algorithm recursively goes through each node in the composition list and repeats the above steps until the size of the composition list is stable. This algorithm is also able to calculate the total response time for the creation of each composition, thus determining the total response time

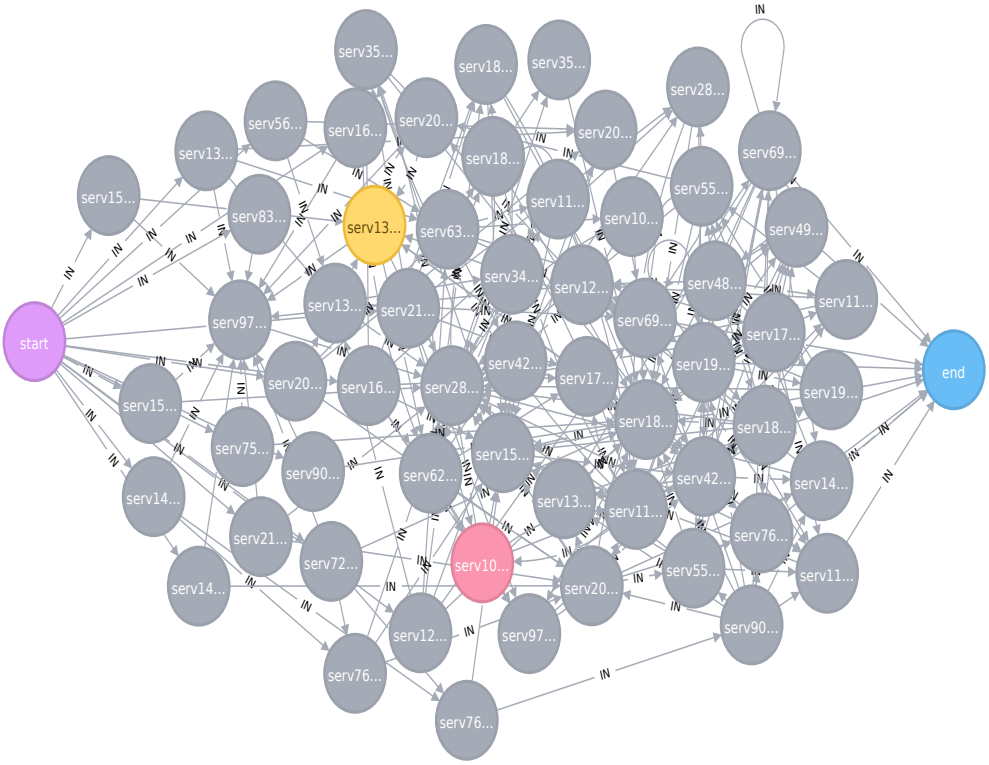

Figure 3.7: Reduced graph database (WSC2008-01)

from task *input* to task *output*. This total response time is then used to determine the quality of the solution, when we generate a *QoS*-Aware service composition. *Algorithm 5* calculates the response time during generation of the Web service composition by setting the duration property which reflects the time taken from End Node to Start Node. When adding node N to the composition list, the algorithm first checks the *druation* property of node N. If N's duration property value is less than the sum of N's previous node duration property value and N's execution time (N's *QoS* time value), then the algorithm sets N's duration property value to the sum of the N's previous node duration property value and N's execution time.

Algorithm 5: Web services composition algorithm (initial populations).

Input : $endNode, relatedNodes$
Output: $candidates$

$i \leftarrow 0$;
$relationships \leftarrow \{\}$;
$candidates \leftarrow \{\}$;
while $i < |getIncomingRelationships(endNode)|$ **do**
    $relationships \leftarrow relationships \cup \{tNode.getIncomingRelationships(endNode)[i]\}$;
    $i \leftarrow i + 1$;
$fulfilledNodes \leftarrow fulfilledNodes \cup \{getNodesFulfillCurrentNode(rels)\}$;
if $|fulfilledNodes| > 0$ **then**
    $candidates \leftarrow candidates \cup \{fulfilledNodes\}$;
    $j \leftarrow 0$;
    while $i < |candidates|$ **do**
        $RecursivelyCallThisMethodUntilTheSizeOfFulfillNodesEquals0$;
        $j \leftarrow i + 1$;

Two examples of Web service composition are shown in Figure 3.8 and Figure 3.9 to illustrate the application of our Web service composition algorithm (*Algorithm 5*) to the reduced graph database we generated in Figure 3.7. Both compositions show the relationship between task *input* and task *output*. For each node in the composition it is possible to click the Web service node *S* to find the inputs of *S* and also click the incoming edge of *S* to find the input values related to the Web service at the other end of the edge.

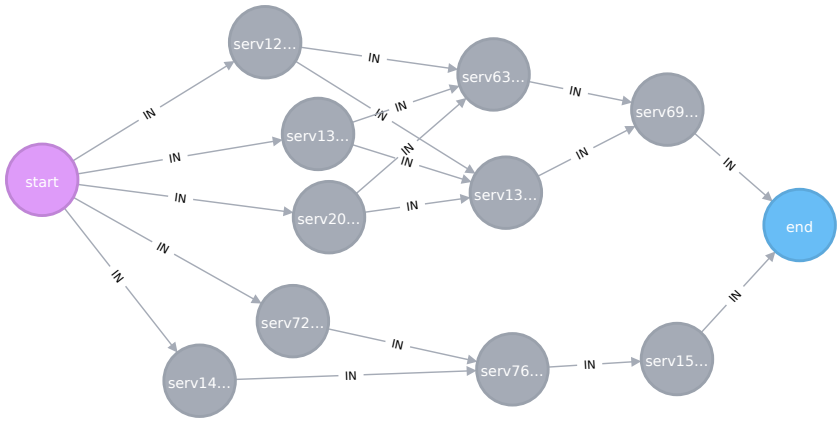

Figure 3.8: Web service composition 1 (WSC2008-01)

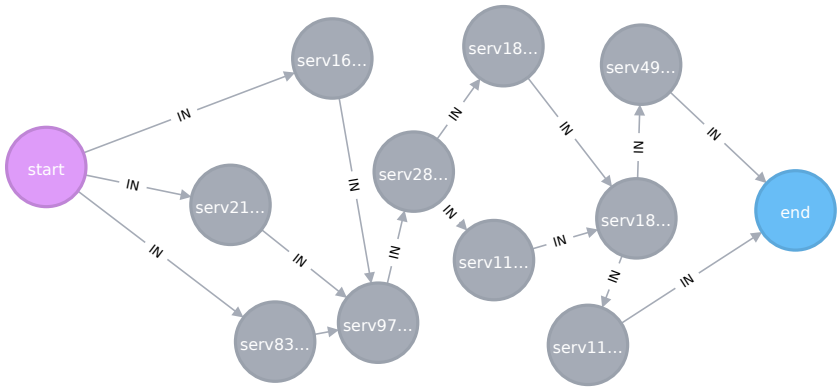

Figure 3.9: Web service composition 2 (WSC2008-01)

3.5 *QoS*-Aware Service Composition

In section 3.4, we proposed a Web service composition algorithm. The Web service composition algorithm we proposed uses a reduced number of Web services to search for service composition solutions using the smallest number of Web services possible.

The next step in our approach is to select a service composition for a set of solutions, which presents the best *QoS* for a given task. There are many service compositions which can fulfill a given task. As we can see in section 1, to meet a user's non-functional requirements, we must aim to find a service composition that proves to have the best *QoS*.

Algorithm 6 is designed to search for the solution with best *QoS*. Using the set of initial population generated from *Algorithm 5*, the algorithm goes through each and every service node invoked in the compositions. It then determines minimums and maximums of execution cost, response time, availability and reliability. Having found all minimums and maximums the next steps are to carry out normalization to restrict all values to the interval

[0,1], and then to apply a fitness function, using normalized *QoS* properties and user defined weights (see section 2.1.3), to each service composition. Lastly, the algorithm returns the service composition which has the best fitness value.

Algorithm 6: *QoS* aware Web services composition algorithm

Input : $InitialCompositions, weights$
Output: $bestComposition$

$i \leftarrow 0$;
$BestComposition \leftarrow \{\}$;
$BestFintess \leftarrow 0$;
$Map < composition, normalizedList > compWithNormalized \leftarrow \{\}$;
while $i < |InitialCompositions|$ **do**
    $j \leftarrow 0$;
    while $j < |InitialCompositions(i)|$ **do**
        $find(maxC, minC, maxT, minT, maxA, minA, maxR, minR)$;
        $j \leftarrow j + 1$;
    $i \leftarrow i + 1$;
$i \leftarrow 0$;
while $i < |InitialCompositions|$ **do**
    $j \leftarrow 0$;
    while $j < |InitialCompositions(i)|$ **do**
        $NormalizedList.add(normalize(Cost, Time, Availability, Availability))$;
        $j \leftarrow j + 1$;
    $compWithNormalized.getKey(composition) \leftarrow NormalizedList$;
    $i \leftarrow i + 1$;
$i \leftarrow 0$;
while $i < |compWithNormalized|$ **do**
    $fitnessValue = fitnessFunction(compWithNormalized(i), weights)$;
    if $fitnessValue > BestFitness$ **then**
        $BestFitness \leftarrow fitnessValue$;
        $BestComposition \leftarrow composition$;

The following example shows a comparison of two compositions and uses data from Table 3.1, which contains *QoS* properties and fitness values of web service compositions.

Example 3.5.1. This example shows a comparison of two Web service compositions. Composition 1 is shown in Figure 3.8 and composition 2 is shown in Figure 3.9. To determine which is the better of the two compositions, we normalized the *QoS* properties and used the fitness functions (section 2.1.3) of each service composition. Since we are only using two compositions in this example, the highest values on Availability and reliability, and the lowest values on Cost and Time will be equal or close to 1. We then set the weight for our fitness function to *0.25*. The result in Table 3.1 shows that composition 2 (originally shown in 3.9) has the best fitness.

3.6 Summary

In this chapter we have proposed how to model and store a service repository, and dependencies between the Web services, as a graph databases. We have also proposed several

QoS-aware service composition algorithms to carry out required tasks. We have shown that the QoS-aware service composition solutions generated by our approach are all functionally correct for all the test cases in the WSC2008.

Table 3.1: Comparison of two web service compositions

Compositions		Cost	Time	Availability	Reliability	Fitness
1 (Figure 3.8)	Non-normalized	48.11	23427.83	1.60043E-5	0.00935	0.25
	Normalized	0.0	0.0	0.0	1.0	
2 (Figure 3.9)	Non-normalized	44.18	12858.46	0.005	0.001408	0.7475
	Normalized	0.9899	1.0	1.0	0.0	

Chapter 4

Evaluation Design

In this chapter we present our performance evaluation of our graph database based approach, by comparing it with an existing approach, namely, the GraphEvol approach presented in [13]. In particular, we evaluated the effectiveness and efficiency of our approach by comparing the quality of composition solutions and the time taken to generate a Web service composition.

4.1 Datasets and Parameters

We evaluated the performance of our approach by testing it on a benchmark dataset, WSC2008 [8] which contains service collections of varying sizes. To provide a comparison with the GraphEvol approach, we ran each task independently 30 times, for each run recording the best Web service composition, fitness value and execution time. For all tests we set the weights for an objective function to the same value as was used with the GraphEvol method, namely, a fixed weight of 0.25. The other parameters for GraphEvol were: a population of 200 candidates, a mutation probability of 0.05, and a crossover probability of 0.5. Individuals were chosen for breeding using tournament selection with a tournament size of 2 [13].

The test to compare the performance of the graph database approach and GraphEvol approach is conducted on a Macbook Pro Mid 2012 with a 2.9GHz dual-core Intel Core i7 processor, 8GB of 1600MHz DDR3 memory, a 1536 Mb Intel HD Graphics 4000 graphics card, a 128GB solid-state drive and the OS X EI Capitan operating system.

4.2 Evaluation Results

In this section we present the results of our evaluation of our proposed algorithms.

4.2.1 Effectiveness of the Reducing Algorithm

To evaluate the effectiveness of our reducing graph database algorithm we applied it to all test cases contained of WSC2008 [8]. Table 4.1 shows the number of Web services for the original repositories and the number of Web service for the reduced graph database. For example, for WSC2008-02, there were 558 Web services in the original repository, but only 66 of them were related to the task input and task output. For task WSC2008-08, there were 8138 Web service in the original repository, but only 131 of them were related to the task input and output. Thus, the number of the Web services was greatly reduced when compared to the number in the original repositories, which greatly improved the execution time when generating the Web service composition.

Table 4.1: Comparison of the number of services in the original repositories and the reduced repositories

Dataset WSC2008	Original service repository	Reduced Service repository
01	158	61
02	558	66
03	604	105
04	1041	46
05	1091	102
06	2198	205
07	4113	195
08	8119	131

4.2.2 Correctness of the Web Service Composition

A composition correctness evaluation was carried out to check correctness of the Web service compositions. The correctness of the Web service composition was determined by checking that all the service node inputs are fulfilled by the output of their proceeding nodes in the graph database. We checked the correctness of all the service composition solutions produced by our algorithm. Our evaluation showed that all the solutions were correct. In this report, we show two composition solutions for Task 1 of WSC2008 with a repository of 158 services. The following examples show that for Task 1 the task and the service compositions produced by our approach (see Figure 4.1 and Figure 4.2).

Task 1: *WSC2008-01*
Task 1 inputs: *[inst1926141668, inst395151449, inst1557679659]*
Task 1 outputs: *[inst1913443608, inst664891780]*

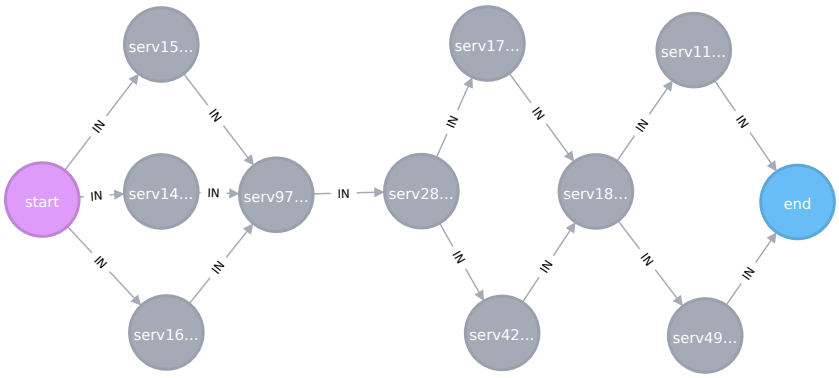

Figure 4.1: Web service composition 1 (Task1, WSC2008-01)

Example 4.2.1. In Figure 4.1, showing a example of Web service composition, the input of the Web service *serv283321609* is I = *inst722854357, inst347634243, inst1881697469, inst746203847* . This matches the output O = *inst722854357, inst2092246857, inst1326239605, inst1881697469, inst1437249127, inst1519789560* of Web service *serv976005395*. This exact match occurs as *inst722854357* and *inst1881697469* are found in both Web service *serv283321609* and Web service *serv976005395*, while, according to the taxonomy tree, *inst1519789560* and *inst1326239605* in O are specializations of *inst347634243* in I and *inst1437249127* in O is a specialization of *inst746203847* in I.

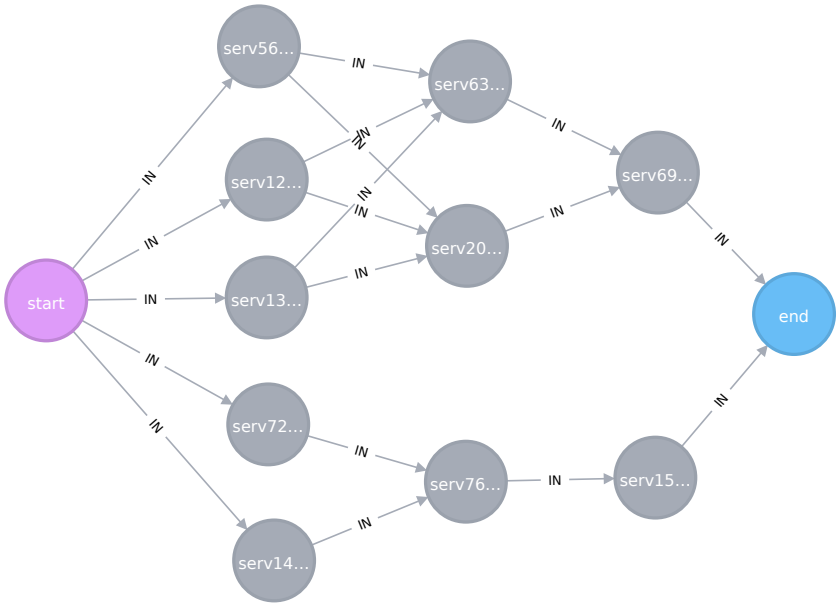

Figure 4.2: Web service composition 2 (Task1, WSC2008-01)

Example 4.2.2. In Figure 4.2, the input of the web service *serv699915007* is I = *inst102675811, inst1689375842, inst1716616603*. This matches the output O_1 = *inst927259823,inst608977925* of Web service *serv2085282617* and the output O_2 = *inst885068313,inst1420249694,inst1488043421* of Web service *serv630482774*. As you can see there is no exact match between the outputs of services *serv2085282617* and *serv630482774*, and inputs of service *serv699915007*, while, according to the taxonomy tree, *inst1488043421* in O_2 is a specialization of inst102675811 in I, *inst927259823* in O_1 is a specialization of *inst1689375842* in I and *inst885068313* in O_2 is a specialization of *inst1716616603* in I.

Thus, the number of Web services involved in both examples is the same. Since we are only evaluating the correctness of the composition, we do not know whether the Web services composition is capable of performing tasks reliably or efficiently (i.e accounting for non-functional attributes). The next section presents the performance of our QoS-aware approach by comparing it with the GraphEvol approach.

4.2.3 Evaluation Results for QoS-Aware Service Composition

Table 4.2 shows a full evaluation comparing the performance of our approach and that of the GraphEvol approach [13]. Both approaches were run on the same machine (mentioned in Section 4.1) to conduct significant analyses. We ran each task independently, for each run recording the best Web service composition, the number of services involved, the fitness value and the execution time taken. Then we calculated the mean, standard deviation for the number of services involved, fitness values and execution times separately for each task. For each run, our approach generated *50* candidates and the best solution from those candidates was chosen. For GraphEvol approach, the best candidate was generated from a population size of *200*.

Table 4.2 includes three columns for each approach. The *number* column records the number of services involved in the best Web service composition. The *time* column records the average time, over 30 independent runs, which was taken to generate the best Web service composition. And the *fitness* column records the average *QoS* fitness value for the best Web service composition, calculated from 30 independent runs.

Table 4.2: Average results of the tests for QoS-aware service composition

Dataset	Graph Database Approach			GraphEvol Approach		
2008	Number	Time (ms)	Fitness	Number	Time (ms)	Fitness
1	10 ± 0.00	2197.90 ± 329 ↓	0.521 ± 0.169 ↓	10.63 ± 2.17	4845.57 ± 315.42	0.645 ± 0.139
2	5 ± 0.00	5347.13 ± 880 ↑	0.48 ± 0.152 ↓	5.87 ± 2.12	3699.77 ± 364.57	0.906 ± 0.00
3	40 ± 0.00	10961.53 ± 790 ↓	0.387 ± 0.1 ↑	41.2 ± 0.79	17221.53 ± 764.85	0.176 ± 0.045
4	10 ± 0.00	3885.70 ± 399 ↓	0.431 ± 0.066 ↑	10.2 ± 0.48	6076.7 ± 281.58	0.305 ± 0.066
5	20 ± 0.00	4510.6 ± 468 ↓	0.403 ± 0.128↑	22.13 ± 2.59	10444.2 ± 572.59	0.164 ± 0.046
6	40 ± 0.00	258503.33 ± 42324 ↑	0.407 ± 0.089 ↑	40.2 ± 0.48	22183.53 ± 1639	0.228 ± 0.06
7	20 ± 0.00	17839.77 ± 763 ↓	0.457 ± 0.097 ↑	23.13 ± 7.33	20304.37 ± 1257	0.316 ± 0.039
8	30 ± 0.00	53003.7 ± 4465 ↑	0.468 ± 0.091 ↑	32.47 ± 3.86	18567.03 ± 2055	0.315 ± 0.028

The number of the Web services involved

The results in Table 4.3 compare the number of Web services required for a composition using two approaches. The result (number of services invoked) with the '*minimum number of Web services*' approach was equal to or very close to the result using the '*non-minimum number of Web services*' approach. However, the fitness values were much higher with the '*minimum number of Web services*' approach. A poor result occurred with Datasets *1* and *3*, and with Dataset *3* the '*non-minimum number of Web services*' approach took *1138667.65* milliseconds (*19* minutes) to generate a single service composition due to the huge search space involved, compared with *10916.53* milliseconds (*10.9* seconds) using the the '*minimum number of Web services*' method. This was because when we restrict our approach to a minimum number of Web services, the search space is also restricted. The result was that when we added web service nodes into the service composition, whenever the number of the web services in the composition was greater than the minimum number of the Web services, our approach led to continual re-starting of the composition generation step in order to find a new composition.

So we decided to use the '*minimum number of Web services*' method to generate Web service compositions in order to find the best candidates. The result was that the number of the services involved in our approach was less than or equal to the numbers of services when using the GraphEvol approach.

Execution time to generate best QoS-Aware Web service composition

Our approach successfully generated a Web service composition for each task in eight datasets from WSC2008. For five tasks our approach was significantly better than the GraphEvol approach in terms of execution time. However, the composition generation time for the remaining three tasks was slower than when using the GraphEvol approach. This was especially true for Dataset *6*, which took *258503.33* milliseconds using our approach compared with *22183.53* milliseconds using the GraphEvol approach.

The result we obtained when using our algorithm with Dataset *6* occurred because Dataset *6* required *40* Web service nodes for the composition and the reduced graph database contained *205* Web service nodes. So one of five nodes was involved in the composition leading to a large number of edges (relationships) between the Web service nodes, and the large number of service nodes and edges increased the search space, thus increasing the total execution time in generating a composition. But overall, our approach performed better than the GraphEvol approach and led to faster generation of web compositions.

Hypothesis test using two fitness value sets for each approach

We used a two-sample t-test to check if the fitness values and execution times for our approach were better than the fitness values and execution times for the GraphEvol approach at the *0.05* significance level. In Table 4.2, it is clear that the P-values from datasets *03* to *08*

Table 4.3: Graph Database: Average (30 independent runs) results

Dataset	Using minimum number of services			Using non-minimum number of services		
2008	Number	Time (ms)	Fitness	Number	Time (ms)	Fitness
1	10 ± 0.00	2197.90 ± 329 ↑	0.52± 0.166	10 ± 0.00	1047.06 ± 367	0.53 ± 0.174
2	5 ± 0.00	5347.13± 880 ↑	0.54± 0.103	5 ± 0.00	3699.77± 364.57	0.546± 0.179
4	10 ± 0.00	3885.70± 399 ↓	0.388± 0.08	10 ± 0.00	6076.7± 281.58	0.26± 0.028
3	40 ± 0.00	10961.53± 790 ↓	0.387± 0.1	40 ± 0.00	1138667.65 ± 83582.58	0.211± 0.068
5	20 ± 0.00	4510.6± 468 ↓	0.398± 0.121	20 ± 0.00	10444.2± 572.59	0.227± 0.065
6	40 ± 0.00	258503.33± 42324 ↑	0.354± 0.103	42.067± 1.29	22183.53± 1639	0.206± 0.086
7	20 ± 0.00	17839.77± 763 ↓	0.411± 0.076	20 ± 0.00	20304.37± 1257	0.317± 0.085
8	30 ± 0.00	53003.7± 4465 ↓	0.438± 0.012	33.67± 2.29	284831.37± 2125	0.358± 0.052

are smaller than the significance level *0.05* for the fitness values set and Datasets *1, 3, 4, 5* and *7* are less than the significance level *0.05* for the execution times set, as indicated by the upward and downward pointing arrows. This means there is considerable evidence that the sets of fitness values and execution times for GraphEvol approach are lower than for our approach. In other words, the best solutions in our approach are significantly superior to the best solutions produced by the GraphEvol approach for 6 out of 8 tasks. For execution time, our approach are faster than GraphEvol for 5 out of 8 tasks and only with Dataset *2, 6* and *8* were our execution time sets poorer than with the GraphEvol approach. In summary however, our approach successfully produced a better result overall, than the GraphEvol approach, in most test cases using the benchmark dataset.

4.2.4 Summary

This chapter we evaluate the performance of our proposed graph database-based approach with an existing GP-based approach by comparing with an existing GP-based approach known as GraphEvol, which is presented in [13]. From our evaluation results, we can conclude that our approach is efficient and effective in generating service composition solutions. It can generate near-optimal QoS-aware composition solutions in a shorter time than the GraphEvol approach.

a

Chapter 5

Conclusion and Future work

This project proposed graph database-based approach to QoS-aware service composition. To do that we have proposed to model and store of service repository as graph database. In this way, information concerning services and relationships between services can be stored and retrieved for any given tasks. We have also proposed several QoS-aware service composition algorithms based on graph databases.

We have evaluated this approach with a benchmark dataset. We have shown that the solutions generated by our approach are all functionally correct for all the test cases in the WSC2008 repository.

We then extended our approach to QoS-aware web service compositions by considering additional QoS requirements. In order to compare the performance of our approach with the GraphEvol approach we used the following four QoS properties: availability, reliability, execution cost and execution duration. These QoS properties were then combined into a single fitness value for each composition. Finding the best web service composition was then possible using the fitness values of individual candidate web compositions we generated.

The results of our evaluation of this approach show that our algorithm can compute optimised solutions, and provides good performance comparing with an existing approach. With our approach, Web service nodes and edges (relationships) are stored in the Graph database permanently, and do not need to be re-created for each task. On the other hand, the problem of GraphEvol approach that only stores this information in memory rather than in permanent storage, and must constantly re-create relationships. This requires additional resources and leads to a longer processing time. GraphEvol also presents the additional challenge of maintaining the web service repository and the associated taxonomy tree, because of the overly complicated format used for storing data structures.

In this report, our approach used a reduced graph database that shrinks the search space for given task, thus improving its performance. In the future, we would like to investigate the generation of a QoS-aware web service composition using the original database, which does not decrease the performance of the solution.

Our approach did not save the task-related services for reuse in the future. Our future work will focus on saving all the task related services in local storage. With each task, if it is similar to a task already saved in local storage, we would reuse those saved task-related services instead of generating a task related reduced graph database from our original graph database.

We modelled only the service repository when developing our approach, which requires loading taxonomy.xml file from local storage. This method means it is impossible to modify the taxonomy file because of the format of the file. Moreover, whenever the taxonomy file changes, Web service graph database also needs to be updated for consistency. Our future work on solving this shortcoming would focus on creating a graph database for taxonomy

repository and trying to link all the taxonomy nodes with services' inputs and outputs. Thus every time taxonomy nodes were modified, service nodes would be automatically updated since they were interconnected.

Appendix A

Bibliography

[1] XML Web Service. http://www.w3schools.com.

[2] UDDI Tutorial. https://www.tutorialspoint.com/uddi.

[3] A. ERDINC YILMAZ, KARAGOZ, P. *Web Services (ICWS)*, Improved Genetic Algorithm Based Approach for QoS Aware Web Service Composition. IEEE International Conference on, pp. 463 – 470 (2014).

[4] ALRIFAI, M., R. T. Combining global optimization with local selection for efficient QoS–aware service composition. 881–890.

[5] AMIRI, M.A., S. H. Effective web service composition using particle swarm optimization algorithm. *In: 6th Int. Symposium Telecommunications,*, pp. 1190–1194 (2012).

[6] ANTSFELD, T. W. L. Finding multi–criteria optimal paths in multi–modal public transportation networks using the transit algorithm. In Intelligent Transport Systems World Congress (2012).

[7] B. ANDREOPOULOS M. SCHROEDER L. ROYER, M. R. Unraveling protein networks with power graph analysis. PLoS Computational Biology (2008).

[8] BANSAL, A., B. M. K. S. B. S. W. T. J. M. WSC–08: continuing the Web services challenge. pp. 351–354.

[9] CHAKER B. M., FATHIA B., H. A., AND S., H. Towards A Graph Based Approach For Web Services Composition. 351.

[10] CHAKRABORTY D, PERICH F, J. A. Web service composition languages: old wine in new bottles? [A]. Proc. of the 29th Conference on Euromicro [C]. Washington, DC: IEEE Computer Society, 298 305 (2003).

[11] DA SILVA A. S., MA, H. Z. M. Effective web service composition using particle swarm optimization algorithm. pp. 3127–3134.

[12] DA SILVA A. S., MA, H. Z. M. A GP Approach to QoS–Aware Web Service Composition and Selection. 2014.

[13] DA SILVA A. S., HUI MA, M. Z. GraphEvol: A Graph Evolution Technique for Web Service Composition. Part II, LNCS 9262, pp. 134–142.

[14] GOTTSCHALK, K., G. S. K. H. S. J. Introduction to web services architecture. IBM Syst. J. 41(2), pp. 170âĂŞ177 (2002).

[15] GOTTSCHALK, K., G. S. K. H. S. J. Introduction to web services architecture. *IBM Syst. J. 41(2),*, pp. 170–177 (2002).

[16] GRAPHDATABASE. *Neo4j https://neo4j.com/docs/developer-manual/current/introduction*, The Neo4j Developer Manual v3.0 (2016).

[17] GRAPHDATABASE. OrientDB http://orientdb.com/orientdb–vs–neo4j/.

[18] GRAPHDATABASE. TITAN Distributed http://titan.thinkaurelius.com/.

[19] GRAPHDATABASE. DEx Graph Database https://dex.citd.tamu.edu/.

[20] JAEGER, M.C., M. G. QoS–based selection of services: The implementation of a genetic algorithm. pp. 1–12.

[21] JING. L, YUHONG. Y, D. L. Full Solution Indexing Using Database for QoS–aware Web Service Composition IEEE International Conference on Services Computing. pp. 99–106 (2014).

[22] LIANGZHAO Z, BOUALEM. B, A. H. N. M. D. J. K., AND HENRY, C. QoS–Aware Middleware for Web Services Composition. 311–327.

[23] LINTHICUM, D. Chapter 1: Service Oriented Architecture (SOA). https://msdn.microsoft.com/en–us/library/bb833022.aspx.

[24] MUCIENTES, M., L. M. C. M. A genetic programming-based algorithm for composing web services. *In: 9th Int. Conf. Intelligent Systems Design and Applications,*, pp. 379–384 (2009).

[25] PIRES P F, BENEVIDES M, M. M. A Reactive Service Composition Architecture for Pervasive Computing Environments [A]. Proc. of the IFIP TC6/WG6. 8 Working Conference on Personal Wireless Communications [C]. Washington, DC: IEEE Computer Society, 53 62 (2002).

[26] RITTER, D. From network mining to large scale business networks. WWW (Companion Volume), pages 989âĂŞ996, (2012).

[27] S., P. Service Composition Scenarios in the Internet of Things Paradigm. AICT–394, pp.53–60, (2013).

[28] SALIM J., V. V. An empirical comparison of graph databases. Doi: 10.1109/SocialCom.2013.106. pp. 708–715 (2013).

[29] SEYYED, V., H. F. M. *In Proceedings of the 2005 Symposium on Applications and the Internet (SAINT'05)*, A Graph–Based Approach to Web Services Composition (2005).

[30] SIRIN E, PARSIA B, W. D. HTN Planning for Web Service Composition Using SHOP2 [J]. Journal of Web Semantics. 377 396 (2004).

[31] VAN DER AALST W M P, DUMAS M, T. H. A. Building Reliable Web Services Compositions [J]. Lecture Notes in Computer Science, 2593:551 562 (2002).

[32] VIKAS A., GIRISH C., S. M. B. S. Understanding approaches for web service composition and execution. Doi 10.1145/1341771.1341773. (2008).

[33] WANG A., MA H., Z. M. Genetic programming with greedy search for web service composition. In: Decker, H., Lhotsk?a, L., Link, S., Basl, J., Tjoa, A.M. (eds.) DEXA 2013, Springer, Heidelberg. pp. 9–17 (2013).

[34] YOO J.J.W., KUMARA S., L. D. O. S. A web service composition framework using integer programming with non-functional objectives and constraints. algorithms 1, 7.

[35] ZHENG L., BENATALLAH B., D. M. K. J. S. Q. Quality driven web services composition. pp. 411–421 (2003).