Using Graph Databases for Automatic Composition of Web Services

by

Zhaojiang Zhang

Submitted in partial fulfilment of the requirements of
Bachelor of Computer Science and Engineering with
Honours in Software Engineering

Victoria University of Wellington
2016

Abstract

With the rapid development of computer network technology and applications, the demand for services from the internet has grown, as has users? expectation that the internet can provide the services they need. A Web service is a software component that takes input data and produces output data. Since a single Web service provides only limited functionality, in order to complete complex tasks, shared 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 provides best quality of service (QoS) with the increasing number of available services that can be used for service composition. It is challenging to find a service with optimal QoS efficiently. In this project, we propose an efficient automated QoS-Aware Web service composition approach using graph databases.

Contents

Chapter 1

Introduction

Service-Oriented Architecture (SOA [3]) is an architectural style for building software applications that uses services available in the network, such as the web. SOA is realised through a standards-based technology called Web services. It promotes loose coupling between Web services, so that they can be reused. A Web service is a self-contained unit of functionality, that takes input data and produces output data. A single Web service provides limited functionality. To provide value-added function, it is necessary to composite the Web services to provide powerful service functions. The result of this kind of selection is to achieve a set of output data needed by a user, which is based on a set of input data provided by the user.

Currently there are three main types of Web service Composition Approach Models. The first type is the traditional approach, such as Integer Linear Programming (ILP) [9] based Web service composition. It lacks scalability since the amount of Web services increases extraordinarily fast. The second type is Evolutionary computing (EC) approaches . For example, Genetic Algorithms (GA) [8], Genetic Programming(GP) [7, 4] and Particle Swarm Optimisation (PSO) [1]. These approaches are very slow when they checks the dependencies between the component services, which requires lots of resources involved. The third type is graph based approach model, such as GraphEval [2, 6]. This type of approach stores

the graph dependency in the memory temporarily rather than saves it in the local storage permanently.

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. The existing Web service composition approaches [9, 2, 8, 4, 1] do not save the Web service dependencies, which means that when running the composition algorithm for different tasks, the algorithm will only keep Web service dependencies in the memory. The problem with this is that when the task has changed it is necessary to regenerate the Web service dependencies. In practice, this behaviour dramatically increase the cost of running the system, the Web service dependencies should only be generated once. Moreover, the existing approaches is required to generate a graph after each composition established. It takes lots of time and slows down the whole progress. Therefore, a new graph database approach is expected to make the progress more effective and efficient. This project aims to make a graph database approach that generates Web service dependencies only once, easily updates the repository and produces graph immediately after database creation.

In order to achieve the five purposes of this project, we have divided the project into two phases. The first three purposes are implemented in the first period, and the second period will focus on the rest two goals. The first goal is to review the existing work that has been done and address the limitations on the existing approaches. The second goal is to use a graph database to model a service repository by considering the Web service dependency. The third goal is to generate non-QoS Web services composition. And also make sure that the services in the composition are all related to the task input and output, and all the internal services are not redundant. Fourthly, we are going to select the service with the best QoS

and generate global optimisation service compositions. Lastly, we will conduct a full evaluation by comparing the performance of our approach and that of the GraphEvol approach.

1.2 Structure of report

The report is organised in following way. Chapter 2 provides a background of the composition problem. Chapter 3 describes the work carried out as far as part of the project. Finally, in Chapter 4, describes the future work to be carried out and present our conclusions so far.

Chapter 2

Background and Literature review

2.1 Background

2.1.1 QoS Properties

QoS is one of the important considered factors when compositing services. It defines the non-functional requirements of a service, such as response time, cost, availability, etc. Good quality of service means good quality of Web service composition. Each of these QoS value determine whether a Web service is reliable, trustworthy, or efficient. Its significance stems from the fact that a Web service may be functionally capable of performing a given task, but might not reliable or efficient enough in performing the task up to the user's satisfaction. Web services are usually advertised with multiple QoS values, each value representing a quality aspect of the Web service called a QoS property. QoS properties are generally the most commonly used characteristics in measuring the quality of Web services and even composite services, this is because they indicate whether a service is capable of measuring up to userâĂŹs expectations.

2.1.2 Neo4j graph database

We employed a Neo4j [5] graph database for our project because it is suitable to represent connected and directed Web services and it allows for very fast retrieval, traversal and navigation of data. A Neo4j Graph Database stores all its data in Nodes and Relationships. Each Node and Relationship has their own properties. For each data set we only need to create a graph database once, and we can then use this database to answer different composition problems.

2.2 Literature review

Over the past few years, Web services have become widespread. A large number of complex applications require service composition. Web services automatic composition technology has become one of the main concerns and challenges to build robust applications, which is making use of distributed services with different functions. Many approaches have been developed for the Web services composition, but only several of them used the concept of graph theory. In this section, we will present a brief overview of some graph based techniques that deal with automatic Web services composition.

Alexandre et al. presented in [3] is an evolutionary computation technique that performs fully automated Web services composition using graph representations for solutions. There are two steps. Firstly, it initialises the population by employing the graph building algorithm which is based on the planning graph approach in composition literature [7]. The second step is to perform traditional mutation and crossover operations on the selected populations, generate a new set of the populations and evaluate the fitness of the new set of populations.

The drawback of [3] is that the graph mutation space contains a lot of invalid service compositions that either violates the constraints of service

dependency or deteriorate the fitness performance. Another disadvantage is that the system saves all the Web service dependencies in the memory. Whenever users use the system, they need to regenerate all the necessary dependencies. This will increase the cost of running the system. Furthermore, the cost to execute the graph building algorithm is very high. The graph building algorithm is employed in two different processes, the process of generating initial populations and the process of performing mutation and crossover operations.

Seyyed, V., H et al. [6] used a graph search algorithm to construct Web services compositions. The algorithm is based on input-output dependencies between Web services. In order to solve the composition problem, authors divided it into two steps. The first step is to look for Web services which potentially participate in the composition. The second step is to find the corresponding composition setup based on these Web services. The graph build algorithm constructs edge between direct related Web services based on input-output. The shortcoming of this approach is that semantic functions were not considered in the dependencies between input and output parameters. It is not guaranteed that the generated composite service presents the requested functionality accurately.

Both of the Web services composition approaches, Alexandre et al. [3] and Seyyed, V., H et al. [6], shared a common drawback. They didn't include quality of service in their consideration. It resulted in that the Web service might not be reliable in performing the task to satisfy users.

Compared with the temporarily Web service repository, this project is going to generate a graph database for each dataset, and the generated database will be saved into local drive. When different tasks loads into the system, the new developed approach only needs to look for the corresponding database rather than regenerating it. It also takes quality of service into consideration by adding weights onto each edge between Web services, which generates global optimisation service compositions.

Chapter 3

Work Done So Far

So far, this report has focused on the model service repository by considering dependency between the Web services and has generated Web service compositions using Neo4j graph databases. In the rest of this chapter, we will describe several new methods and show the results. We proposed five algorithms, which were used to create Web services graph database, reduce the database and generate initial population. The overall design of

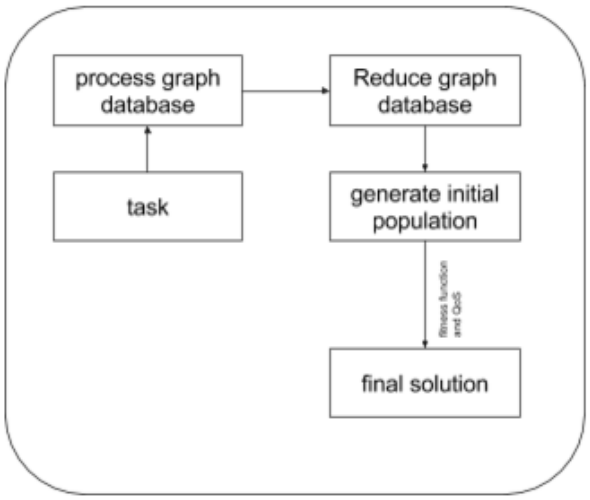


Figure 3.1: Overall system design.

our system is shown as Figure 3.1. The following sections will introduce the design in detail.

3.1 Proposed graph database

Web service composition modeling

Each Web service is an entity, which has different properties and is defined by its name, id, qos, inputs, outputs, inputServices, and outputServices. The first step is to model service repositories, so that the information of input, output and QoS, as well as the dependencies between services, are kept in graph database.

Inputs and Outputs

For the graph database, we need to label each Web service to make it visually identifiable in the graph database. The different inputs and outputs of each Web service should be easily identifiable by a concept defined within a taxonomy tree. In the taxonomy tree, any two concepts A, B can be related to each other in one of the four possible ways. The first way, A is a generalization of B. The second way, A is a specialization of B. The third way, A and B are not related to each other. Lastly, A equals B. For example, 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 presented in the taxonomy tree. The inputs and outputs of each Web service are clearly defined in terms of concepts belonging to a specialized taxonomy.

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

```
t  i ← 0;
while i < |taxonomyNodes| do
    tNode ← taxonomyNodes[i];
    tNode.parents ← findParentsNodes(tNode);
    tNode.children ← findChildrenNodes(tNode);
    i ← i + 1;
i ← 0;
while i < |serviceNodes| do
    j ← 0;
    outputs ← serviceNodes[i].outputs;
    while j < |outputs| do
        tNode ← findTaxonomyNode(outputs[j]);
        k ← 0;
        while j < |tNode.parents| do
            tNode.parents ← tNode.parents ∪ {serviceNodes[i]};
            k ← k + 1;
        j ← j + 1;
    j ← 0;
    inputs ← serviceNodes[i].inputs;
    while j < |inputs| do
        tNode ← findTaxonomyNode(inputs[j]);
        k ← 0;
        while j < |tNode.children| do
            tNode.children ← tNode.children ∪ {serviceNodes[i]};
            k ← k + 1;
        j ← j + 1;
    i ← i + 1;
```

InputServices and OutputServices

InputServices is the set of Web services connected to the input side of the Web service, and *outputServices* is the set of Web services which sends output from the Web service to another input. The reason that I added 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. Here is the algorithm for identifying all the input and output services for each Web service using a taxonomy tree. For example, Service *A* has inputs $I_A = \{\text{a}\}$ and outputs $O_A = \{\text{b}\}$, service *B* has Inputs $I_B = \{\text{e}\}$ and outputs $O_B = \{\text{b}\}$, and service *C* has Inputs $I_C = \{\text{b}\}$ and outputs $O_C = \{\text{f}\}$. Our algorithm 1 searches services inputs and outputs of *A*, *B* and *C*. Each value in inputs and outputs is a node in the taxonomy tree, we call it taxonomy node (*tNode*). Each *tNode* has input services (*IS*) and output services (*OS*) fields, then we add Web service into corresponded *IS* or *OS* field. Taxonomy is represented as a tree, and each *tNode* might 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 Web service into all *tNode*'s parent node's *IS* field. Same as when add Web service into *OS*, we also add Web service into all *tNode*'s children node's *OS* field. After we run the algorithm 1, we get $IS_a = \{\text{A}\}$, $IS_b = \{\text{C}\}$,$OS_b = \{\text{A, B}\}$, $IS_e = \{\text{B}\}$ and $OS_f = \{\text{C}\}$.

The algorithm 2 goes through each input value in the Web services (*WS*), finds corresponding tNode and adds all the services in *IS* field including all the value in the *IS* filed in all parent taxonomy nodes into *WS*. It also runs through each output value in the *WS*, finds corresponding *tNode* and adds all the services in *OS* field including all the value in the *OS* filed in all children taxonomy nodes into ws. After we run the algorithm 2, we get $IS_C = \{\text{A, B}\}$, $OS_A = \{\text{C}\}$, and $OS_B = \{\text{C}\}$.

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

```
i ← 0;
while i < |serviceNodes| do
    sNode ← serviceNodes[i];
    sNode.inputServices ← {};
    sNode.outputServices ← {};
    j ← 0;
    while j < |sNode.outputs| do
        tNode ← findTaxonomyNode(sNode.outputs[j]);
        k ← 0;
        while k < |tNode.outputServices| do
            sNode.outputServices ←
              sNode.outputServices ∪ {tNode.outputServices[k]};
            k ← k + 1;
        j ← j + 1;
    j ← 0;
    while j < |sNode.inputs| do
        tNode ← findTaxonomyNode(sNode.inputs[j]);
        k ← 0;
        while k < |tNode.inputServices| do
            sNode.inputServices ←
              sNode.inputServices ∪ {tNode.inputServices[k]};
            k ← k + 1;
        j ← j + 1;
    i ← i + 1;
```

Web service relationships

Relationship is an directed edge that connects the output of a Web service node to the input of another node by their dependencies. Like Web service

nodes, relationships also have properties. It has a from node, to node and direction. Relationship between Web service nodes is an essential part of a graph database. It allows to find related Web services. In our case, we use relationships to solve composition problems.

Algorithm 3 shows the process to create relationships between Web services. For each Web service, we have found and have added all of the possible Web services into the set of inputServices in Algorithm 2. For Algorithm 3, we go through each Web service node *N*, loop through the corresponding inputServices of *N*, and then add relationships between the Web services in the set of inputServices and node *N*.

Algorithm 3: create relationships between Web services.

```
i ← 0;
while i < |serviceNodes| do
    sNode ← serviceNodes[i];
    j ← 0;
    while j < |sNode.inputServices| do
        inputSNode ← sNode.inputServices[j];
        relation ← inputSNode.createRelationshipTo(sNode);
        relation.setProperty("From" : inputsNode);
        relation.setProperty("To" : sNode);
        relation.setProperty("Outputs" : inputsNode.outputs);
        relation.setProperty("Direction" : incoming);
        j ← j + 1;
    i ← i + 1;
```

3.2 Reducing graph databases

We create temporary graph databases in order to generate a reduced graph. A reduced graph only contains the Web services which relate to tasks at hand. This section describes how we achieved this goal. The concept is

simple. Firstly, we created a start node S and an end node *E*. The start node *S* contained the inputs value of the task and the end node *E* contained the outputs value of the task. Secondly, we created new relationships between *S* and all other related nodes, and between *E* and all other related nodes. Lastly, we went through each Web service node to check if it involved in a relationship between *S* and *E*. If there wasn't such a relationship, we removed this Web service node from the graph. Web service node was also removed if its input was not fulfilled. Algorithm 4 shows the process to reduce the original graph database by removing all the redundant nodes.

Algorithm 4: reduce graph database.

```
i ← 0;
relatedNodes ← {};
while i < |serviceNodes| do
    sNode ← serviceNodes[i];
    if hasRelationship(sNode, startNode) ∧
     hasRelationship(sNode, endNode) then
        relatedNodes ← relatedNodes ∪ {sNode};
    if !fulfillInputs(sNode) then
        removeRelatedNodes(sNode);
    i ← i + 1;
return relatedNodes;
```

3.3 Generating Web service compositions

Our reduced graph database algorithm allows us to get all the Web service nodes related to the task. This section explains how to use related Web service nodes to composite Web services in order to create an initial population. At this stage, QoS is not part of our consideration. It will be handled during the second part of the project.

Algorithm 5 is designed to create feasible compositions that starts from the end node and searches internal nodes which directly connected to the end node. Internal nodes are added into the composition list if they could fulfill the input of the end node. Then the algorithm recursively goes through each node in the composition list and repeats the above logic until the size of the composition list is stable. This algorithm is also able to restrict the number of Web services for each composition list.

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

$i \leftarrow 0$;
$relationships \leftarrow \{\}$;
$composition \leftarrow \{\}$;
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**
    $composition \leftarrow composition \cup \{fulfilledNodes\}$;
    $j \leftarrow 0$;
    while $i < |composition|$ **do**
        $RecursivelyCallThisMethodUntilTheSizeOfFulfillNodesEquals0$;
        $j \leftarrow i + 1$;

3.4 Preliminary Evaluation

Our evaluation was carried out to check correctness of the Web service compositions. QoS is not included in the first part of the project, as it will be considered in the second part. The dataset employed in this evaluation

was set 1 of WSC 2008. This dataset contains 158 Web services. Due to the page limitation, we only briefly discuss two compositions generated by our system. All examples have been manually checked and they indeed represent the Web service composition correctly. The following shows the task and the service compositions produced by our approach (shown in Figure 3.2).

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

Results:

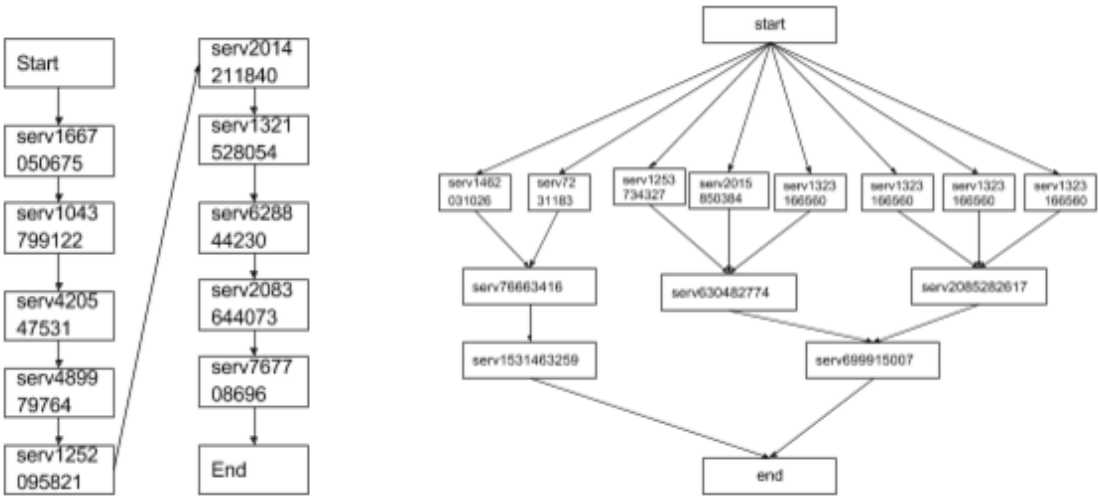


Figure 3.2: service compositions

The matching rules of Web services are validated by checking dependencies. For task 1 of WSC2008, the input of the Web service serv1043799122 is $I = \{\text{inst1227313922, inst1710062503, inst1154604639}\}$. It matches the output $O = \{\text{inst1129002422, inst918042856, inst1710062503, inst1156243145, inst1895009674, inst2038175551, inst1916720658}\}$ of the Web service serv

1667050675. It is because inst 1710062503 occurs in both Web services serv 1043799122 and serv 1667050675, while inst 1895009674 in *O* is a specialization of inst 1227313922 in *I*, and inst 2038175551 in *O* is a specialization of inst 1154604639 in *I* according to the taxonomy tree.

As you can see the amount of Web service involved in both compositions are different, there are 10 Web services included in left side of Figure 3.2 and 13 Web services included in the right side of the Figure 3.2. Since we only evaluate the correctness of the composition, we do not know whether the Web services composition is reliable or efficient enough to perform tasks. Therefor, from the second part of the project, we will add QoS as part of the Web service node and relationship property to create a weighted relationship between Web services. Then we will use a weighted relationship to generate QoS-Aware service composition. The full evaluation will compare the performance of our approach and that of the GraphEvol approach presented in [2].

Chapter 4

Future Plan and Conclusions

This report introduces the use of Neo4j graph database to model QoS-aware Web service composition. The proposed five algorithms can efficiently create the database, reduce the database, and generate initial population. The proposed algorithm can automatically construct a relatively good initial population. The evaluation results verified the correctness of the Web service composition.

Currently, our proposed method is only able to generate initial populations. For the second part of the project, we will use QoS to calculate global optimisation to find the better Web service compositions. We will also consider using GP to improve initial populations by using mutation and crossover operators.

The first task for future work is to design a fitness function to produce solutions with the smallest possible number of service nodes and with the shortest possible paths from the start node to the end node. The second task for future work is to build QoS-aware composition to calculate the global optimisation and improve the results. The third task for future work is to analyse and design the evaluation by comparing the performance of our system against the traditional GP approach and other GP approaches.

Chapter 5

References

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

[2] DA SILVA A. S., HUI MA, M. Z. Graphevol: A graph evolution technique for web service composition. Part II, LNCS 9262, pp. 134–142.

[3] LINTHICUM, D. Chapter 1: Service oriented architecture (soa). https://msdn.microsoft.com/en–us/library/bb833022.aspx.

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

[5] NEO4JTEAM. The neo4j developer manual v3.0. *https://neo4j.com/docs/developer-manual/current/introduction* (2016).

[6] SEYYED, V., H. F. M. A graph-based approach to web services composition. *In Proceedings of the 2005 Symposium on Applications and the Internet (SAINT'05)* (2005).

[7] WANG, A., M. 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.

[8] YILMAZ, A. ERDINC, K. P. Improved genetic algorithm based approach for qos aware web service composition. *Web Services (ICWS) IEEE International Conference on, pp. 463 - 470* (2014).

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

Appendix A

Appendix: Project Proposal Slides

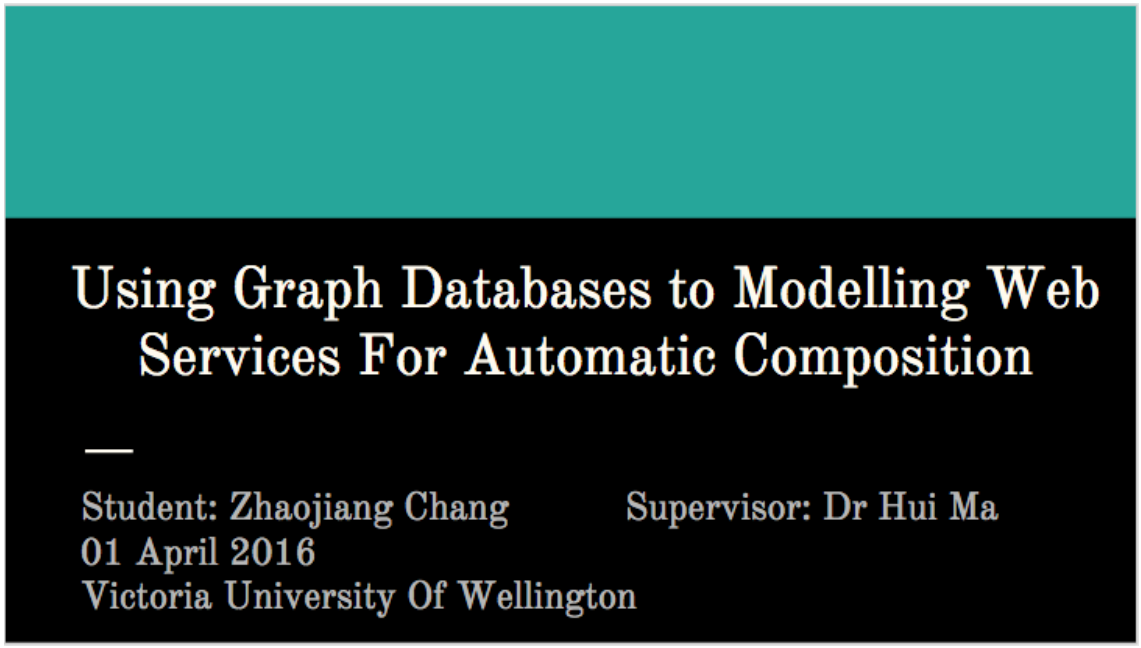


Figure A.1: slide 1

BACKGROUND

- Web Service
- Web Service Composition
- QoS Aware Service Composition
  - Quality of Service - response time, cost, availability,reputation and security etc.
  - Good Quality of Service == Good Quality of Web Service composition
- Graph Database
  - **nodes, edges** and **properties** to represent and store data.
  - **Advantages:** Easy to represent connected data, vary faster to retrieve/traversal/navigate large amounts connected data and no joins to retrieve data

Figure A.2: slide 2

PROBLEM

Current Web Service Composition Approach Models:

- Traditional approaches: e.g., ILP, lack of scalability
- Evolutionary Computation(EC) approaches: e.g., GA, GP, PSO, expensive to check data dependencies between component services
- Graph Based:
  - Some of the approaches do not considerate Quality of Service
  - Easy to check data dependencies - but using Petri-Nets expensive to check data dependencies

Figure A.3: slide 3

AIM

The aim of this project is to propose an automated QoS aware Web service composition approach by using Graph Databases

OBJECTIVES

Our project objectives are:

1. To survey existing approaches to solve web service composition problems by graph-based techniques
2. To propose and implement a graph data model for web service descriptions and compositions
3. To evaluate the proposed solution using benchmark data from web service contests

Figure A.4: slide 4

SOLUTION

1. Model service repositories using Graph databases (Tool: Neo4j) based on WSDL description of Web services
2. Propose a service composition approach using the Graph databases
3. Evaluate our proposed approach by comparing the performance of ours with other existing approaches, e.g.,EC and Graph based approaches

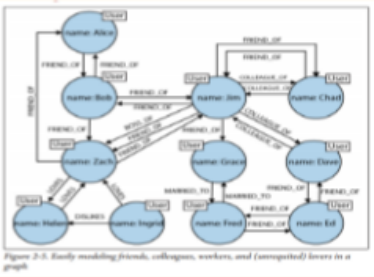
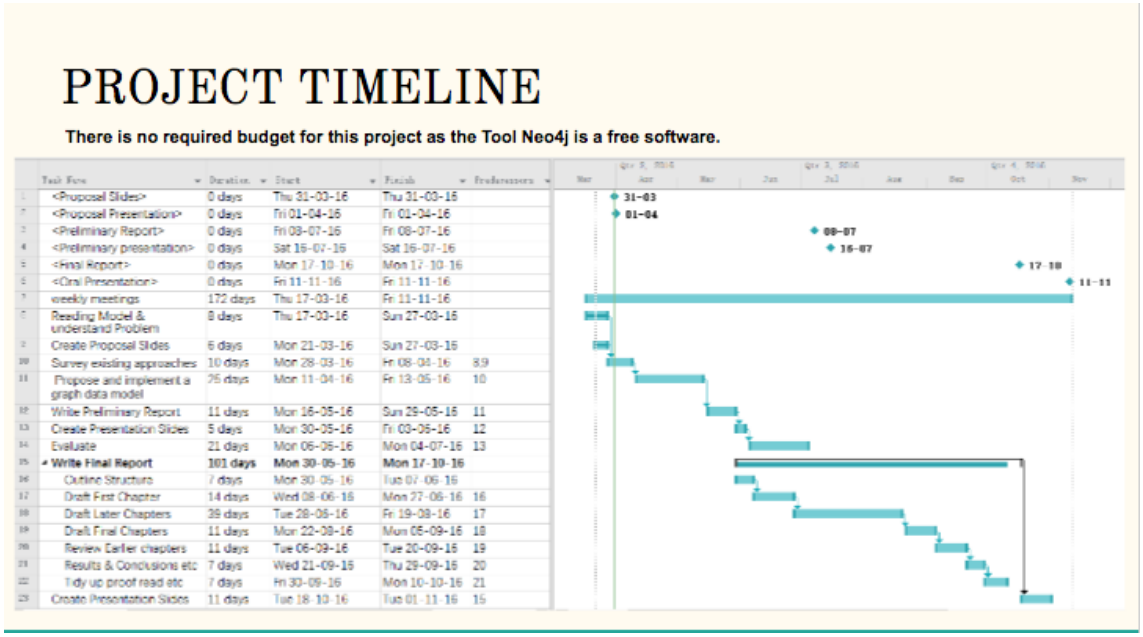


Figure A.5: slide 5



	Task Name	Duration	Start	Finish	Predecessors
1	<Proposal Slides>	0 days	Thu 31-03-16	Thu 31-03-16	
7	<Proposal Presentation>	0 days	Fri 01-04-16	Fri 01-04-16	
3	<Preliminary Report>	0 days	Fri 08-07-16	Fri 08-07-16	
4	<Preliminary presentation>	0 days	Sat 16-07-16	Sat 16-07-16	
5	<Final Report>	0 days	Mon 17-10-16	Mon 17-10-16	
6	<Oral Presentation>	0 days	Fri 11-11-16	Fri 11-11-16	
7	weekly meetings	172 days	Thu 17-03-16	Fri 11-11-16	
7	Reading Model & understand Problem	8 days	Thu 17-03-16	Sun 27-03-16	
9	Create Proposal Slides	6 days	Mon 21-03-16	Sun 27-03-16	
10	Survey existing approaches	10 days	Mon 28-03-16	Fri 08-04-16	8,9
11	Propose and implement a graph data model	75 days	Mon 11-04-16	Fri 13-05-16	10
12	Write Preliminary Report	11 days	Mon 16-05-16	Sun 29-05-16	11
13	Create Presentation Slides	5 days	Mon 30-05-16	Fri 03-06-16	12
14	Evaluate	21 days	Mon 06-06-16	Mon 04-07-16	13
15	**Write Final Report**	**101 days**	**Mon 30-05-16**	**Mon 17-10-16**	
16	Outline Structure	7 days	Mon 30-05-16	Tue 07-06-16	
17	Draft First Chapter	14 days	Wed 08-06-16	Mon 27-06-16	16
18	Draft Later Chapters	39 days	Tue 28-06-16	Fri 19-08-16	17
19	Draft Final Chapters	11 days	Mon 22-08-16	Mon 05-09-16	18
20	Review Earlier chapters	11 days	Tue 06-09-16	Tue 20-09-16	19
21	Results & Conclusions etc	7 days	Wed 21-09-16	Thu 29-09-16	20
22	Tidy up proof read etc	7 days	Fri 30-09-16	Mon 10-10-16	21
23	Create Presentation Slides	11 days	Tue 18-10-16	Tue 01-11-16	15

Figure A.6: slide 6