# Genetic Algorithm Based QoS-Aware Service Compositions in Cloud Computing

Zhen Ye[1,2], Xiaofang Zhou[1,3], and Athman Bouguettaya[2]

[1] School of Information Technology and Electrical Engineering
The University of Queensland, Australia
[2] CSIRO ICT Centre, Australia
[3] School of Information, Renmin University of China, China
Key Lab of Data Engineering and Knowledge Engineering,
Ministry of Education, China
zhenye@itee.uq.edu.au, zxf@uq.edu.au, Athman.Bouguettaya@csiro.au

**Abstract.** Services in cloud computing can be categorized into two groups: *Application services* and *Utility Computing Services*. Compositions in the application level are similar to the Web service compositions in SOC (Service-Oriented Computing). Compositions in the utility level are similar to the task matching and scheduling in grid computing. Contributions of this paper include: 1) An extensible QoS model is proposed to calculate the QoS values of services in cloud computing. 2) A genetic-algorithm-based approach is proposed to compose services in cloud computing. 3) A comparison is presented between the proposed approach and other algorithms, i.e., exhaustive search algorithms and random selection algorithms.

## 1 Introduction

Cloud computing is emerging as the new paradigm for the next-generation distributed computing. It has attracted a lot of attention in both academia and industry. An increasing number of organizations have started to migrate their IT infrastructure to the cloud. Several companies such as Amazon, Microsoft and IBM are already offering cloud solutions in the market. The vision of cloud computing is that computing will not be conducted on local computers in the future, but on distributed facilities operated by third-party computing utilities [1]. Cloud computing aims at uniformly exposing hardware and software as a service that can be rented at will [2]. In this new framework, organizations no longer require the large capital outlays in hardware to deploy their services. They need not be concerned about overprovisioning for a service whose popularity does not meet their predictions, thus wasting costly resources, or underprovisioning for one that becomes popular, thus missing potential customers and revenue.

Services in cloud computing can be categorized into application services and utility computing services [2]. Almost all the software/applications that are available through the Internet are application services, e.g., flight booking services, hotel booking services. Utility computing services are software or virtualized

hardware that support application services, e.g., virtual machines, CPU services, and storage services. Service compositions in cloud computing therefore include compositions of application services and utility computing services. Compositions in the application level are similar to the Web service compositions in SOC. Compositions in the utility level are similar to the task matching and scheduling in grid computing. A composite application service fulfills several tasks (i.e. *abstract services*). Each task is implemented by several substitute application services (i.e. *concrete services*). The choice among these substitute services is based on their non-functional properties, which are also referred to as Quality of Service (QoS). QoS values of these substitute application services are further dependent on the choices of utility computing services. In a word, once a concrete application service is selected for each abstract service, the following decisions have to be made: *matching*, i.e. assigning concrete application services to utility computing services, and *scheduling*, i.e. ordering execution sequence of application services.

Several approaches and systems are proposed to solve Web service composition problems in SOC. Most of them [3] [4] only consider the compositions in the application level. Composition approaches in cloud computing need to consider compositions both in the application level and utility computing level. Besides, most existing composition approaches in SOC [3] [4] use integer programming to find the global optimized solution. Although this is useful for small-scale compositions, it incurs a significant performance penalty if applied to large-scale composition problems such as compositions in cloud computing [6]. Contrasts to these existing approaches, Genetic Algorithms (GAs) are heuristic approaches to iteratively find near-optimal solutions in large search spaces. There is ample evidence regarding the applicability of GAs for large-scale optimization problems [5] [6]. Whereas, no GA based approach is available to compose services in cloud computing.

In this paper, a genetic-algorithm-based service composition approach is proposed for cloud computing. In particular, a coherent way to calculate the QoS values of services in cloud computing is presented. At last, comparisons between the proposed approach and other approaches show the effectiveness and efficiency of the proposed approach. The rest of the paper is structured as follows: Section 2 illustrates the background and preliminaries of service composition in cloud computing. Section 3 elaborates the details of the proposed approach. Section 4 evaluates the approach and shows the experiment results. Section 5 presents the related work to the proposed approach. Section 6 concludes this paper and highlights some future work.

## 2 Preliminaries

This section presents preliminary knowledge about cloud computing, service compositions in cloud computing. Genetic algorithms are also introduced at the end of this section. Services in a cloud, refers to both the applications delivered as services over the Internet and the hardware and system software in the data centers that provide those services [2]. Cloud computing provides easy access to *Application Services* (i.e. SaaS) and *Utility Computing Services* (UCS) (Fig. 1).
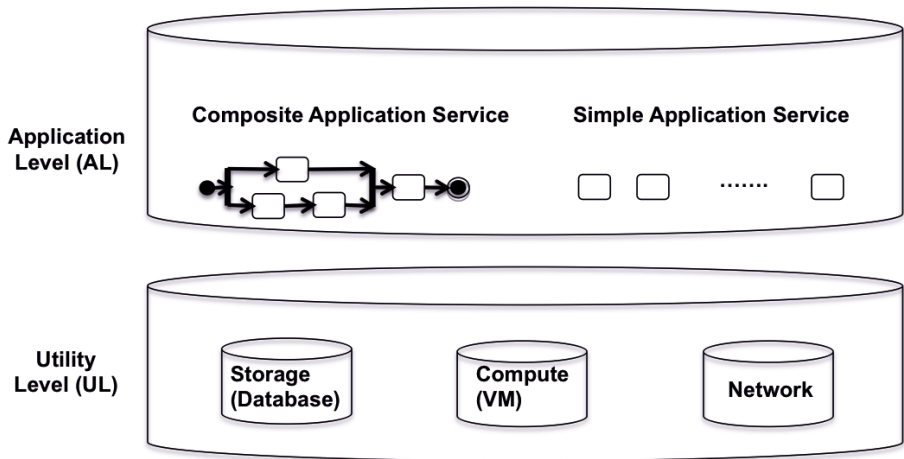
**Fig. 1.** Cloud System

- Application Services are the most visible services to the end users. Examples of application services include: Salesforce's CRM applications, Google Apps etc. Application services that contain other component application services are *Composite Application Services. Simple Application Services* do not contain other component application services. *Application Users* can be end users or other application services. *Application Providers* are providers of application services.
- Utility Computing Services. Some vendors use terms such as PaaS (Platform as a Service) or IaaS (Infrastructure as a Service) to describe their products. In this paper, PaaS and IaaS are considered together as UCSs. PaaS are platforms that are used to develop, test, deploy and monitor application services. For example, Google has Google App Engine works as the platform to develop, deploy and maintain Google Apps. Microsoft Azure and Force.com are also examples of PaaS. IaaS services provide fundamental computing resources, which can be used to construct new platform services or application services. UCSs can be categorized into computation services, i.e., *Virtual Machines* (VMs); storage services, i.e., *Databases*; and network services. *UCS Users* are these application providers or other utility computing services etc. *UCS Vendors* are these companies or organizations that make their computing resources available to the public.

## 2.1 Service Compositions in Cloud Computing

A composite service is specified as a collection of abstract application services according to a combination of control-flow and data-flow. Control-flow graphs are represented using UML activity diagrams. Each node in the graph is an abstract application service. There are four control-flow patterns. For example, Fig. 2 shows a composite service consists of four patterns of control-flows. $S_1$ and
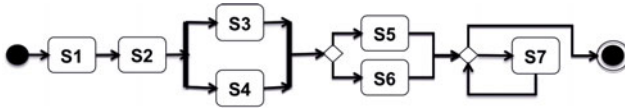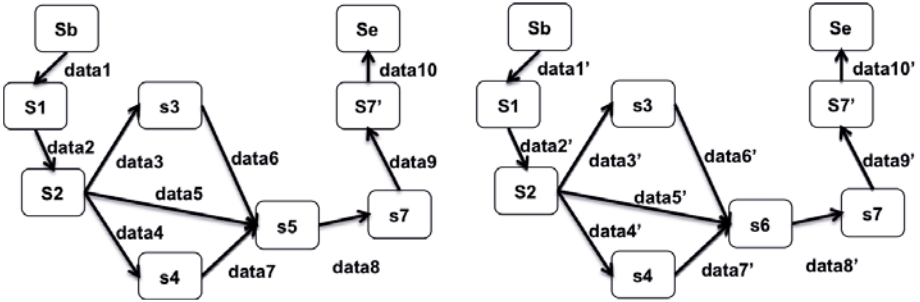
**Fig. 2.** Control Flows



**Fig. 3.** Data Flow Graphs

$S_2$ run in a sequence pattern. $S_3$ runs in parallel with $S_4$ (parallel pattern). After that, either $S_5$ or $S_6$ is selected to run (conditional pattern). Finally, $S_7$ cycles for a certain times (loop pattern). There are several data-flow graphs for the same control-flow graph, if the control-flow graph contains conditional patterns. Fig. 3 shows the two data-flow graphs corresponding to the control-flow shown in Fig. 2. Directed acyclic graphs (DAGs) are used to represent data-flow graphs. The start node of an edge is denoted as *source service*, the node where the edge ends is denoted as *destination service*. Source services must be executed before the destination services. The destination service can only be executed after all its source services are finished. Node $S_b$ represents the start point of the composite service. $S_e$ represents the end point. The data items transferred between these abstract application services form a set $D = \{data_i, 1 \leq i \leq d\}$.

A set of $k_n$ concrete application services $\{s_{n1}, s_{n2}, \ldots, s_{nk_n}\}$ is available to execute the abstract service $S_n$. A concrete application service can be executed on several virtual machines, databases and network services. After mapping each abstract service to a concrete application service, VM UCSs and Database UCSs need to be selected for each application service. Network UCSs need to be selected for each data transfer in the data-flow graph. Assume each VM can only execute one application service at a time. A late application service can only execute on the VM after the former application services finish their executions. To sum up, any solution to a composition problem in cloud computing includes: 1) Map the abstract application services to concrete application services and corresponding UCSs (VM, database and network services). 2) Schedule the execution order of the application services. This execution order is a topological sort [7] of the data-flow graph, i.e. a total ordering of the nodes in the DAG that obeys the precedence constraints.

| QoS Attributes | Time $Q^1$ | Price $Q^2$ | Availability $Q^3$ | Reputation $Q^4$ |
|---|---|---|---|---|
| Aggregation Function | $\sum_{i=1}^{m} Q_i^1$ | $\sum_{i=1}^{m} Q_i^2$ | $\prod_{i=1}^{m} Q_i^3$ | $\prod_{i=1}^{m} Q_i^4$ |

**Fig. 4.** Aggregation Functions for each QoS Attribute

## 2.2 QoS Model

QoS attributes contains (1) ascending QoS attributes, i.e. a higher value is better; (2) descending QoS attributes, i.e. a smaller value is better; (3) equal QoS attributes, i.e. no ordering but only equality, e.g. security protocol should be X.509. Four QoS attributes are considered in this work: response time, price, availability and reputation. Among them, time and price belong to the descending attributes while availability and reputation belong to the ascending attributes. Vector $\boldsymbol{Q} = Q^1, Q^2, Q^3, Q^4$ denotes all the available QoS attributes. $Q^i, 1 \leq i \leq 4$ represents time, price, availability and reputation.

QoS values of an application service consist of three parts: execution, network and storage QoSs. Existing QoS models in SOC [3] only consider the execution QoSs. *Execution QoS* refers to the QoS value for executing an application service in a specified VM. Same application service has different execution QoS in different VMs. *Network QoS* refers to the QoS for transferring data from one application service to another using a specified network UCS. Data transfers are determined by the source services and the destination services. Each data will be transferred as soon as the source service produces them. Hence, network QoS values are only calculated at the destination services. *Storage QoS* refers to the QoS for storing certain amount of data for a certain time using specified database service. Assume no data will be stored during the execution of an application service. Therefore, the only data needs to be stored are the input data. For example, a destination service has two input data. One input data arrives early, the other arrives later. The earlier arrived data need to be stored when waiting the second input data to arrive. The QoS value for a service therefore equals to the sum of execution QoS, network QoS and storage QoS. Fig. 4 shows the aggregation functions for calculating the overall QoS for composite services. $m$ is the number of component services in the composite service. QoS values are normalized using Simple Additive Weighting (SAW), which is also used in [3]. The best QoS values are normalized to 0, the worst QoS values are normalized to 1. Thus, higher normalized values indicate worse quality.

QoS constraints (denoted as $QC$) for composite services have two types: *Global Constraints* and *Local Constraints*. Global Constraints are the QoS constraints for the overall composite service, while Local Constraints apply to component services within the composition. A global constraint (GC) for a given QoS attribute $Q^l$ is denoted as $GC^l$. Local constraints are denoted as $LC^l$. Constraints on different QoS attributes are transformed into inequality constraints [8]. $QC^1$ (time) and $QC^2$ (price) can be transformed by subtract the threshold to the constraints, e.g. $QC^1 \leq 1\ minute$ is transformed to $QC^1 \Leftarrow QC^1 - 1 \leq 0$;

$QC^2 \leq 5\ USdollars$ is transformed to $QC^2 \Leftarrow QC^2 - 5 \leq 0$. $QC^3$ (availability) and $QC^4$ (reputation) can be transformed by subtracting the QoS value from the threshold, e.g. $QC^3 \geq 0.9$ is transformed to $QC^3 \Leftarrow 0.9 - QC^3 \leq 0$. Constraints on equal QoS attributes can be transformed using this function: $QC \Leftarrow |QC| - \epsilon \leq 0$, where $\epsilon$ is the tolerance allowed range (a very small value).

### 2.3   Genetic Algorithms

*Genetic Algorithms* (GAs) are heuristic approaches to iteratively find near-optimal solutions in large search spaces. Any possible solution to the optimization problem is encoded as a *Chromosome* (normally a string). A set of chromosomes is referred to as a *Population*. The first step of a GA is to derive an initial population. A random set of chromosomes is often used as the initial population. This initial population is the first generation from which the evolution starts. The second step is *selection*. Each chromosome is eliminated or duplicated (one or more times) based on its relative quality. The population size is typically kept constant. The next step is *Crossover*. Some pairs of chromosomes are selected from the current population and some of their corresponding components are exchanged to form two valid chromosome. After crossover, each chromosome in the population may be *mutated* with some probability. The mutation process transforms a chromosome into another valid one. The new population is then *evaluated*. Each chromosome is associated with a *fitness value*, which is a value obtained from the objective function (details will be discussed in section 3). The objective of the evaluation is to find a chromosome that has the optimal fitness value. If the stopping criterion is not met, the new population goes through another cycle (iteration) of selection, crossover, mutation, and evaluation. These cycles continue until the stopping criterion is met.

## 3   QoS-Aware Service Composition in Cloud Computing

Assume there are $m$ VM UCSs ($vm_1, vm_2, \ldots, vm_m$), $p$ database UCSs ($db_1$, $db_2$, $\ldots$, $db_p$) and $q$ network UCSs ($net_1, net_2, \ldots, net_q$) in different cloud systems. Each composition solution (chromosome) consists of two parts, the matching string ($ms$) and the scheduling string ($ss$). $ms$ is a vector of length $n$, such that $ms(i) = s_j vm_x db_y net_z$, where $1 \leq i \leq n$, $1 \leq j \leq k_n$, $1 \leq x \leq m$, $1 \leq y \leq p$ and $1 \leq z \leq q$. A matching string means that abstract service $S_i$ is assigned to concrete service $s_{ij}$ which is lodged on virtual machine $vm_x$ and has database service $db_y$, network service $net_z$. The scheduling string is a topological sort of the data-flow graph. $ss(k) = i$, where $1 \leq i, k \leq n$; i.e. service $S_i$ is the $k$th running service in the scheduling string. Thus, a chromosome represents the mapping from each abstract service to concrete service and UCSs, together with the execution order of the application services. Fig. 5 shows a solution to the composite problem that has the control-flow shown in Fig. 2, and the data-flow shown in Fig. 3 (left DAG). In this solution, $ms$ represents the mapping string, e.g., abstract service $S_1$ is mapped to application service $S_{11}$, $S_{11}$ is further deployed on virtual machine $vm_1$ and database $db_1$. The network service for
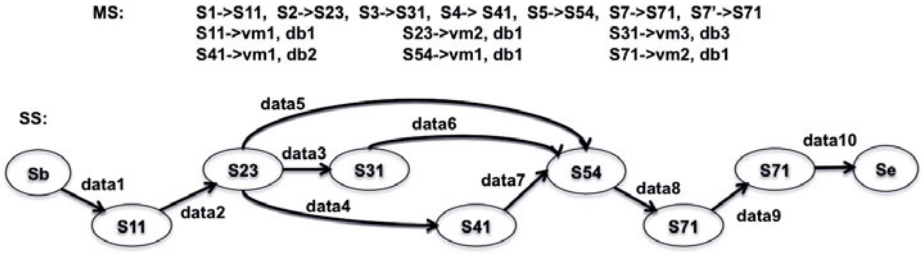
MS:  S1->S11, S2->S23, S3->S31, S4-> S41, S5->S54, S7->S71, S7'->S71
     S11->vm1, db1        S23->vm2, db1         S31->vm3, db3
     S41->vm1, db2        S54->vm1, db1         S71->vm2, db1



**Fig. 5.** Composition Solution

a transferred data is determined when the source service and the destination service are mapped to the corresponding virtual machine and database services. *ss* represents the scheduling string of the solution, e.g., the execution order of this solution in Fig. 5 is $S_{11}, S_{23}, S_{31}, S_{41}, S_{54}, S_{71}, S_{71}$.

### 3.1 Genetic Algorithm Based Approach

In the first step, a predefined number of chromosomes are generated to form the initial generation. The chromosomes in a generation are first ordered by their fitness values (explained later) from the best to worst. These having the same fitness value are ranked arbitrarily among themselves. Then a rank-based roulette wheel selection schema is used to implement the selection step [9]. There is a higher probability that one or more copies of the better solution will be included in the next generation, since a better solution has a larger sector angle than that of a worse solution. In this way, the chromosomes formed the next generation are determined. Notice that the population size of each generation is always $P$.

The crossover operator for a matching string randomly chooses some pairs of the matching strings. For each pair, it randomly generates a cut-off point to divide both matching strings into two parts. Then the bottom parts are exchanged. The crossover operator for a scheduling string randomly chooses some pairs of the scheduling strings. For each pair, it randomly generates a cut-off point, which divides the scheduling strings into top and bottom parts. The abstract application services in each bottom part are reordered. The new ordering of the services in one bottom part is the relative positions of these services in the other original scheduling string in the pair. This guarantees that the newly generated scheduling strings are valid schedules. Fig. 6(a) demonstrates the crossover operator for a scheduling string.

The mutation operator for a matching string randomly selects an abstract service and randomly replaces the corresponding concrete service and other utility computing services. The mutation operator for a scheduling string randomly chooses some scheduling strings. It then randomly selects a target service. The *valid range* of this target service is the set of the positions in the scheduling string at which the target service can be placed without violating any data
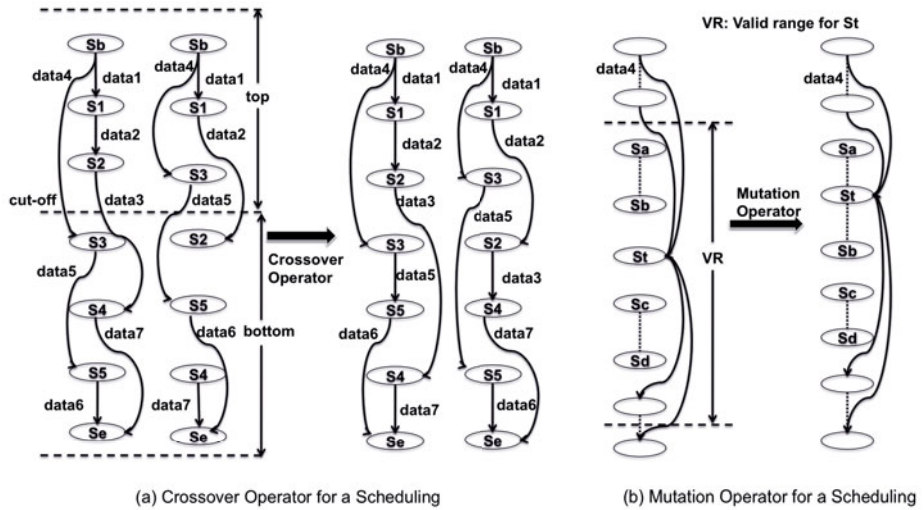
**Fig. 6.** Crossover and Mutation Operators

dependency constraints. The valid range is after all source services of the target service and before any destination service of the target service. The mutation operator can move this target service randomly to another position in the scheduling string within its valid range. Fig. 6(b) demonstrates the mutation operator for a scheduling string. $s_v$ is between $s_b$ and $s_c$ before the mutation, it is between $s_a$ and $s_b$ after the mutation operator.

After crossover and mutation operators, GA will evaluate the chromosomes using *fitness function*. The fitness function needs to maximize some QoS attributes (i.e. ascending attributes), minimize some other attributes (i.e. descending attributes) and satisfy other QoS attributes (i.e. equal QoS attributes). In addition, the fitness function must penalize solutions that do not meet the QoS constraints and drive the evolution towards satisfaction. The distance from constraint satisfaction for a solution $c$ is defined as:

$$D(c) = \Sigma_{i=1}^{l} QC^i(c) \times e_i \times weight^i, e_i = \begin{cases} 0 & QC^i(c) \leq 0 \\ 1 & QC^i(c) > 0 \end{cases} \quad (1)$$

where $weight^i$ indicates the weight of the QoS constraint. Notice that this distance function for constraints include both local and global constraints specified. The fitness function for a chromosome $c$ is then defined as follows:

$$F(c) = \Sigma_{i=1}^{4} w^i * Q^i(c) + weight_p * D(c) \quad (2)$$

$w^i$ are the weights for each QoS attribute. $weight_p$ is the *penalty factor*. Several features are highlighted when calculating the fitness function based on the match string and the scheduling string:

(a) Example 1                  (b) Example 2 for Data Forwarding
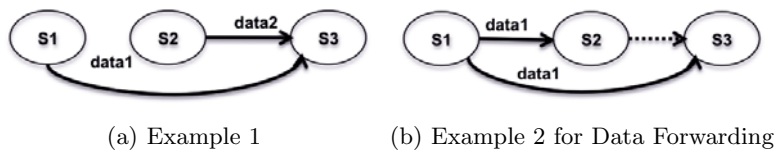
**Fig. 7.** Example of Scheduling String

1. Services are executed exactly in the order specified by the scheduling string. For example, Fig. 7(a) shows a scheduling string for a composition. Assume there are two different match strings for this $ss$. a) $ms_1$: Let $S_1$ and $S_2$ be assigned to the same VM $vm_1$, and $S_3$ be assigned to another VM $vm_2$. In this chromosome, because $S_1$ is to be executed before $S_2$, $data_1$ is available before $data_2$. Thus, $data_1$ will be transferred to $S_3$ before $data_2$. And $data_1$ will be stored in $S_3$'s database service till $data_2$ has been transferred to $S_3$. b) $ms_2$: Let the three services $S_1$, $S_2$, and $S_3$ be assigned to three different VMs $vm_1$, $vm_2$ and $vm_3$. $S_2$ starts to execute just after $S_1$ starts, $S_1$ and $S_2$ can be considered to start their execution at the same time. If $data_2$ is available ($S_2$ executes faster) before $data_1$, $data_2$ will be stored in $S_2$'s database service till $data_1$ has been transferred to $S_3$.

2. Another important feature is *data forwarding* [5]. For an input data, the source service can be chosen among the services that produce or consume this input data. All the consumers of this input data can be forwarders. For example, Fig. 7(b) shows a scheduling string. $S_2$ and $S_3$ both have the input data from $S_1$. $S_2$ may forward $data_1$ from $S_1$ to $S_3$, i.e. shown as the dashed line in Fig. 7(b). This kind of data forwarding is not allowed in our work. Data must be only transferred from the original data producer to its consumers.

Stop criterions for the proposed approach are: 1) Iterate until the constraints are met (i.e. $D(c) = 0$). 2) If this does not happen within $MAXGEN$ generations, then iterate until the best fitness value remains unchanged for a given number ($MAXGEN$) of generations. 3) If neither 1) nor 2) happens within $MAXGEN$ generations, then no solution will be returned.

## 3.2   Handling Multiple Data Flow Graphs

Assume the composite service (e.g. shown in Fig. 2) has multiple data-flow graphs (shown in Fig. 3). For each data-flow graph, an optimal composition solution can be generated using the proposed GA-based approach. Since each of the optimal solution only covers a subset of the composite service, further actions are needed to aggregate these partial composition solutions into an overall solution. Assume the composite service has $f$ data-flow graphs (i.e. $dfg_1, dfg_2, \ldots, dfg_f$). The approach adopts the following strategies to aggregate multiple solutions into an overall solution:

– Given an abstract service $S_i$, if $S_i$ only belongs to one data-flow graph (e.g. $dfg_j$), then the proposed approach selects $dfg_j$'s solution $chromosome_j$ to execute abstract service $S_i$.

– Given an abstract service $S_i$, if $S_i$ belongs to more than one data-flow graphs, then there are many solutions can be used to execute $S_i$. The proposed approach will select the most frequently used solution (from execution history), or ask end users to select a preferable solution.

## 4    Experiment and Evaluation

Our experiments consist of two parts. First, comparisons are conducted between the proposed approach and other approaches in small-scale scenarios. Second, comparisons are conducted in large-scale scenarios. All the experiments are conducted on computers with Intel Core 2 Duo 6400 CPU (2.13GHz and 2GB RAM).

### 4.1    Creation of Experimental Scenarios

Randomly generated scenarios are used for the experiments. Each scenario contains a control-flow graph and a data-flow graph. QoS values of different concrete services, virtual machines, database services and network services for each abstract service are generated randomly with uniform probability. A scenario generation system is designed to generate the scenarios for experiments. The system first determines a root pattern (i.e. sequence, conditional, parallel, loop patterns) with uniform probability for the control-flow. Within this root, the system chooses with equal probability to either place an abstract services into it or to choose another composition pattern as substructure. This procedure ends until the generation system has spent the predefined number ($n$) of abstract services. All the conditional patterns have 2 possible options, either of them has the probability of 0.5. Each loop pattern will run for twice. There are $k$ candidate concrete services to implement each abstract service. The number of data transferred between each abstract services in the flow graph is $d$. Each concrete service can be lodged in $m$ virtual machines, $p$ database services and $q$ network services. These variables are predefined and used as input (denoted as $\{n, k, d, m, p, q\}$) to the generation system. Small-scale scenarios have the input $\{5, 2, 6, 3, 3, 3\}$. Large-scale scenarios have 100 abstract services. Each abstract service can be executed by 30 concrete services. 120 data items are transferred between services and each concrete service is suitable to run in 20 different VMs, 20 different database services and 20 network services. The four QoS attributes and the four QoS constraints have same weight equals 1. The execution QoS, network QoS and storage QoS were randomly generated with uniform distribution from the following intervals: $Q^1(Time) \in [100, 2000]$, $Q^2(Price) \in [200, 1000]$, $Q^3(Availability) \in [0.9750, 0.9999]$ and $Q^4(Reputation) \in [1, 100]$.

Every approach runs 50 times for each scenario. All the results shown below are the average values from these experiments. Each experiment for the GA-based approach starts from a different initial population each time. The probability of crossover $p_{cross} = 0.4$ is the same for the matching string and scheduling string. The probability of mutation $p_{mut} = 0.1$ is also the same for the matching string and scheduling string. The approach uses rank-based roulette wheel

schema for selection. The angle ratio of the sectors on the roulette wheel for two adjacently ranked chromosomes, i.e. R, was chosen to be $1 + 1/P$, where $P$ is the population size. By using this simple formula, the angle ratio between the slots of the best and median chromosomes for $P = 50$ (and also for $P = 200$ for large-scale scenarios) is very closely to the optimal empirical ratio value of 1.5 in [10]. $MAXFIT$ equals to 150. $MAXGEN$ equals to 1000. *Exhaustive search approach* would traverse all the possible solutions to the composition problem and find the optimized solution that has the smallest fitness value. Although this approach would always find the most optimal composition solution, the execution time is extremely high. *Random selection approach* is also a GA-based approach. This approach would randomly select chromosomes to form a new generation. Comparisons with these approaches show the effectiveness and efficiency of the proposed approach. Integer Programming (IP) approaches have been proposed to solve QoS-aware service composition in SOC. The IP approach is implemented using LPSolve [11], which is an open source integer programming system. Comparisons with IP approach show the scalability of the proposed approach.

## 4.2   Experiments Results

Small-scale experiments are conducted on 10 different test datasets. We only show two of them in Fig. 8(b) to make the graph much easier to read. Fig. 8(a) shows the results between the proposed approach and the exhaustive search approach. Proposed GA-based approach would always find near-optimal solution compared to exhaustive search algorithms. Fig. 8(b) shows the comparisons between the proposed approach and the random selection solution. As shown in this figure, proposed approach will always reach an optimized fitness value while random selection seldom converges. To sum up, the proposed GA based approach will always reach an optimal fitness value and the converged point is very close to the actual optimal point. Fig. 9 shows the efficiency of the proposed approach. These experiments are conducted on small-scale scenarios. Each test dataset has the same configuration, except for the number of concrete services for each abstract service. As shown in Fig. 9, the execution time increases quickly at the beginning, but keeps stable when the number of concrete services for each abstract service is larger than 200.

As shown in Fig. 10(a), IP approach performs as good as the GA based approach at the beginning. Notice that, when the number of the abstract services becomes more than 40, IP approaches would cost exponential growing time to solve the composition problems. Fig. 10(b) shows the fitness value's trend corresponding to the increment of the number of the abstract services. Both IP approach and GA based approach behave well when the number of abstract services is relatively small. When the number of abstract services increases, the optimal fitness value obtained from GA based approach also increases. This is because population size and other related variables stay the same when the number of the abstract services varies. Hence, GA based approach are more scalable and efficient than IP approaches.
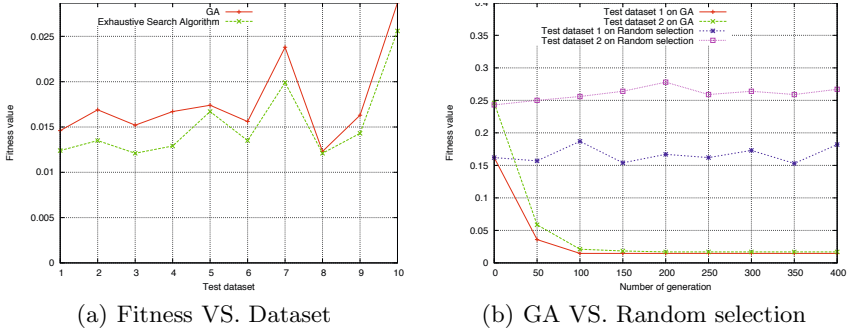
(a) Fitness VS. Dataset          (b) GA VS. Random selection
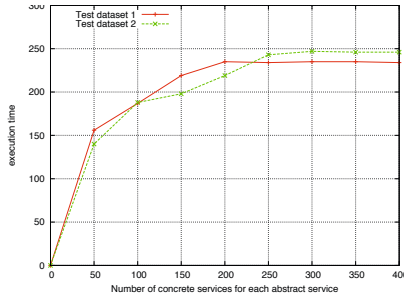
**Fig. 8.** Experiment Result 1



**Fig. 9.** Time VS. Concrete services

## 5   Related Work

Most composition approaches in SOC use linear programming methods. [3] presents two approaches: one focuses on local optimization, the other on global optimization. They use integer programming to solve the global optimization problem. The limit of this approach is that all QoS attributes need to be linearized as integer programming is a linear programming approach. [12] proposes an improved approach based on [3], using Mixed Linear Programming (MILP) approach. They also introduce several concepts such as loop peeling and negotiation mechanisms to address situation where no feasible solution can be found. [13] proposes an approach to decompose global QoS constraints into local constraints with conservative upper and lower bounds. These local constraints are resolved by using an efficient distributed local selection strategy.

All of the aforementioned approaches only consider the service composition problems in small-scale scenarios. These linear programming approaches are not suitable to handle large-scale scenarios problems, e.g. service composition in cloud computing. [14] was the first to use GA for optimization of QoS-aware compositions in SOC. The results show that their GA implementation scales

(a) GA VS. Integer Programming on Time

(b) GA VS. Integer Programming on Fitness
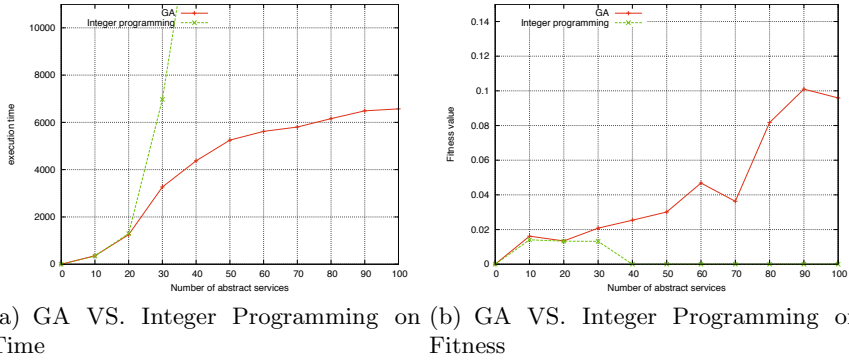
**Fig. 10.** GA VS. Integer Programming Approach

better than linear programming. [15] presents a GA and a Culture Algorithm (CA) for Web service compositions. The first algorithm is similar to [14], the latter uses a global belief space and an influence function that accelerate the convergence of the population. [6] presents a mutation operator which consider both the local and global constraints to accelerate the converge of the population.

Existing GA-based approaches are solely focus on service composition in application level, which do not consider the computing resources composition. Service composition in cloud computing involves application service composition and computing resources matching and scheduling. In this paper, a genetic algorithm based approach is proposed to compose services in cloud computing, by combining QoS-aware service composition approaches and resources matching and scheduling approaches.

## 6 Conclusion

A genetic algorithm based approach is presented for service compositions in cloud computing. Service compositions in cloud computing involve the selections of application services and utility computing services. The chromosome size is bound to the number of $n$ of abstract services. The number of possible application services and utility computing services only augments the search space. For small-scale scenarios, the proposed approach finds optimal solutions. For larger-scale problems, it outperforms the integer programming approach. This is a beginning to propose robust service composition approaches in cloud computing. Future work may focus to eliminate several assumptions: 1) QoS values for each component are known in this research. Calculating the QoS values at runtime is one direction; 2) penalty factor in the fitness function is static. More dynamic fitness functions can be used to improve the performance of the approach. 3) novel crossover and mutation operators may accelerate the converge.

# References

1. Buyya, R., Yeo, C., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25(6), 599–616 (2009)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: Above the clouds: A berkeley view of cloud computing. Tech. rep. (February 2009)
3. Zeng, L., Benatallah, B., Ngu, A., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Transactions on Software Engineering 30(5), 311–327 (2004)
4. Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., Dustdar, S.: An end-to-end approach for QoS-aware service composition. In: Proceedings of 13th IEEE International EDOC Conference, pp. 1–4 (September 2009)
5. Wang, L., Siegel, H., Roychowdhury, V., Maciejewski, A.: Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. Journal of Parallel and Distributed Computing 47(1), 8–22 (1997)
6. Rosenberg, F., Muller, M., Leitner, P., Michlmayr, A., Bouguettaya, A., Dustdar, S.: Metaheuristic Optimization of Large-Scale QoS-aware Service Compositions. In: 2010 IEEE International Conference on Services Computing, pp. 97–104. IEEE, Los Alamitos (2010)
7. Cormen, T.: Introduction to algorithms. The MIT press, Cambridge (2001)
8. Coello, C., Carlos, A.: Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. Computer Methods in Applied Mechanics and Engineering 191(11-12), 1245–1287 (2002)
9. Srinivas, M., Patnaik, L.: Genetic algorithms: A survey. Computer 27(6), 17–26 (1994)
10. Whitley, D., et al.: The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In: Proceedings of the Third International Conference on Genetic Algorithms, vol. 1, pp. 116–121, Citeseer (1989)
11. Berkelaar, M., Eikland, K., Notebaert, P., et al.: lpsolve: Open source (mixed-integer) linear programming system. Eindhoven U. of Technology
12. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Transactions on Software Engineering, 369–384 (2007)
13. Alrifai, M., Risse, T.: Combining global optimization with local selection for efficient QoS-aware service composition. In: Proceedings of the 18th International Conference on World Wide Web, pp. 881–890. ACM, New York (2009)
14. Canfora, G., Di Penta, M., Esposito, R., Villani, M.: An approach for QoS-aware service composition based on genetic algorithms. In: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, pp. 1069–1075. ACM, New York (2005)
15. Gaber, J., Bakhouya, M.: An affinity-driven clustering approach for service discovery and composition for pervasive computing. In: ACS/IEEE International Conference on Pervasive Services, pp. 277–280. IEEE, Los Alamitos (2006)