WCCI 2010 IEEE World Congress on Computational Intelligence
July, 18-23, 2010 - CCIB, Barcelona, Spain CEC IEEE

A Hybrid Genetic Algorithm for the Optimal Constrained Web Service Selection Problem in Web Service Composition

Maolin Tang, *Senior Member, IEEE* and Lifeng Ai

Abstract— **Web service composition is an important problem in web service based systems. It is about how to build a new value-added web service using existing web services. A web service may have many implementations, all of which have the same functionality, but may have different QoS values. Thus, a significant research problem in web service composition is how to select a web service implementation for each of the web services such that the composite web service gives the best overall performance. This is so-called optimal web service selection problem. There may be mutual constraints between some web service implementations. Sometimes when an implementation is selected for one web service, a particular implementation for another web service must be selected. This is so called *dependency constraint*. Sometimes when an implementation for one web service is selected, a set of implementations for another web service must be excluded in the web service composition. This is so called *conflict constraint*. Thus, the optimal web service selection is a typical constrained combinatorial optimization problem from the computational point of view. This paper proposes a new hybrid genetic algorithm for the optimal web service selection problem. The hybrid genetic algorithm has been implemented and evaluated. The evaluation results have shown that the hybrid genetic algorithm outperforms other two existing genetic algorithms when the number of web services and the number of constraints are large.**

I. Introduction

Web service technology is based on open XML standards (i.e. SOAP, WSDL, and UDDI) and has features such as interoperability, decoupling and just-in-time integration, which make it possible to build new value-added web services using existing web services. This is so called *Web service composition*. For example, a travel booking web service can be built by aggregating a flight booking web service, a car rental web service, a travel insurance web service, an accommodation booking web service, a payment web service, and an itinerary planning Web service.

A web service may have multiple implementations offered by different providers. The implementations have the same functionality, but may have different Quality of Service (QoS) values. For example, they may have different response time, price, reputation, availability and so on. Thus, a significant research problem in the web service composition is how to select an implementation for each of the web services in the composite web service. Generally, an implementation of a web service may not dominate all the other implementations of the same web service for all the QoS criteria. An implementation may be better than some other implementations in terms of some QoS criteria, but may not be as good as them in terms of other QoS criteria. Thus, when selecting an implementation for a web service, it is difficult or impossible to get some good QoS values without compromising the others. Therefore, the objective of the web service selection problem is to select an implementation for each of the web services in the composite web services such that the overall QoS is maximal. This web service selection problem is also called QoS-aware web service composition.

In the web service selection there might be some web service implementations that are dependent on each other. When selecting an implementation for one web service, we must select a particular implementation for another web service in the composite web service. For example, when building a travel booking web service, if we select a particular travel insurance web service that only accepts payments made by Master cards, then we must select a payment web service that accepts Master cards. This kind of constraints is called *dependency constraint*. In addition, in the web service selection there might be some web service implementations that conflict with each other. When selecting an implementation for one web service, we must not select a particular implementation for another web service in the composite web service. For example, when building a travel booking web service, if we select a particular flight booking web service implementation that does not accept deposits made by Master cards, then we must not select an implementation for the payment web service that supports Master cards. This type of constraints is called *conflict constraint*. In the web service selection, both dependency constraints and conflict constraints must be considered.

Although various optimal web service selection problems have been intensively studied and different approaches have been proposed in the past few years [1], [2], [3], [4], [5], [6], [6], [7], [8], [9], [10], [11], the study on the optimal web service selection problem with constraints remains open. From the computational point of view, the web service selection problem is a typical constrained combinatorial optimization problem. Thus, genetic algorithms might be efficient and effective for solving the problem. In our preliminary research on the problem, we have proposed two genetic algorithms for the problem, using different techniques to handle the constraints. In this paper we present a new genetic algorithm for the constrained web service selection problem, aiming to further improve the quality of solutions. Different from the existing genetic algorithms, this new genetic algorithm

Maolin Tang is with the Faculty of Science and Technology, Queensland University of Technology, Gardens Point Campus, 2 George Street, Brisbane, Australia (phone: +61 7 31385225; email: m.tang@qut.edu.au).
Lifeng Ai is with the Faculty of Science and Technology, Queensland University of Technology, Gardens Point Campus, 2 George Street, Brisbane, Australia (phone: +61 7 31387753; email: l.ai@qut.edu.au).

is a hybrid one. It utilizes a local optimizer to improve the individuals in the population of the genetic algorithm. This new genetic algorithm algorithm has been implemented and evaluated by comparing it with the two existing genetic algorithms. Evaluation results have shown that the new hybrid genetic algorithm did produce better solutions than the two existing genetic algorithms although its computation time is slight longer than that of the other two genetic algorithms.

The remainder of the paper is organized as follows. First of all, we formulate the research problem in Section II. Then, we review related work in Section III. After that, we present our new hybrid genetic algorithm and evaluation results in Section IV and Section V, respectively. Finally, we conclude this this research in Section VI.

II. Problem Formulation

When building a new composite web service, the following process is usually is followed. First of all, we design a workflow for the composite web service. Fig. 1 is an example of workflow, which consists of 10 web services $\overline{W}_1$, $\overline{W}_2$, $\cdots$, $\overline{W}_{10}$. Then, we get the information about all available implementations for each of the web services in the workflow. This can be done by using a web service discovery tool or a web service broker. The information includes their URLs, the inter-dependencies and mutual conflicts between the web service implementations, as well as their QoS values of interest. Finally, we use a web service composition tool to select the optimal combination of web service implementations accommodating the constraints, which is the problem to be addressed in this paper.

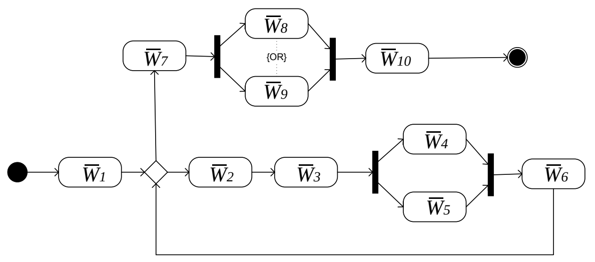


Fig. 1. An instance of workflow for web service composition

In this problem formulation, we follow the terminologies used by the web service community. In the rest of paper, when we say *abstract web service*, we refer to a web service in a workflow and when say say *concrete web service*, we mean the implementation of an abstract web service. In addition, in the problem formulation, we only consider five most popular QoS attributes. However, the problem formulation and the hybrid genetic algorithm can be easily extended to include any new QoS attributes or to exclude any of the five QoS attributes.

Given

- a workflow of a composite web service, which contains a set of abstract web services $\overline{W} = \{\overline{W}_1, \overline{W}_2, \cdots, \overline{W}_n\}$, where n is the total number of abstract web services in the workflow;
- all the available concrete web services for each of the abstract web services $W = \{(w_{11}, w_{12}, \cdots, w_{1k_1}), (w_{21}, w_{22}, \cdots, w_{2k_2}), \cdots, (w_{n1}, w_{n2}, \cdots, w_{nk_n})\}$, where $\overline{w}_{ij}$ represents the j^{th} concrete web service of abstract web service W_i and k_i is the total number of concrete web services of abstract web service $\overline{W}_i$;
- QoS values for response time, price, reputation, reliability and availability for each of the concrete web services w_{ij}, v^1_{ij}, v^2_{ij}, v^3_{ij}, v^4_{ij} and v^5_{ij}, respectively, where $1 \le i \le n$ and $1 \le j \le k_i$;
- weights for QoS criteria, c^1, c^2, c^3, c^4, and c^5 for response time, price, reputation, reliability and availability, respectively, where $c^1 + c^2 + c^3 + c^4 + c^5 = 1$;
- a set of conflicts between the concrete web services $C = \{(w_{i_1 j_1}, w_{i_2 j_2})|$ if the j_1^{th} concrete web service is selected for abstract web service x_{i_1}, then abstract web service w_{i_2} must not select the j_2^{th} concrete web service$\}$, where $1 \le i_1, i_2 \le n$, $1 \le j_1 \le k_{i_1}$ and $1 \le i_2 \le k_{i_2}$;
- a set of dependencies between the concrete web services $D = \{(w_{i_1 j_1}, w_{i_2 j_2})|$ if the j_1^{th} concrete web service is selected for abstract web service x_{i_1}, then abstract web service w_{i_2} must select the j_2^{th} concrete web service$\}$, where $1 \le i_1, i_2 \le n$, $1 \le j_1 \le k_{i_1}$ and $1 \le i_2 \le k_{i_2}$;

Find

a selection plan $X = (x_1, x_2, \cdots, x_n)$, where x_i is a concrete web service of abstract web service $\overline{W}_i$ and $1 \le i \le n$, such that

$F(X) = \sum_{k=1}^{2}(\frac{v_k^{max} - v_k(X)}{v_k^{max} - vk^{min}} * c^k) + \sum_{k=3}^{5}(\frac{v_k(X) - v_k^{min}}{v_k^{max} - v_k^{min}} * c^k)$ is maximal, where v_k is the aggregated QoS value for criteria k, and v_k^{max} and v_k^{min} represent the possible maximal and minimal aggregated QoS values of criterion k, respectively, where $1 \le k \le 5$ (the aggregated QoS values calculation follows the methods presented in [2])

Subject to

all the constraints in C and D are satisfied.

III. Related Work

In our previous research on the web service selection problem, we have proposed two genetic algorithms [9], [10]. The major difference between the two genetic algorithms is that they adopt different strategies to handle infeasible individuals. In both of the genetic algorithms, each individual in the population represents a selection plan. Since there may be dependency constraints and conflict constraints between the selected concrete web services, new web service selection plans (individuals) generated by the initial population procedure, the crossover and the mutation operators may not be feasible. Thus, the genetic algorithm must use some strategies to deal with the infeasible individual problems.

The strategy adopted by the genetic algorithm in [9] is that no infeasible individuals (web service selection plans) are allowed in the population. Since it cannot be guaranteed that the initial population generator, the crossover operator and the mutation operator can always produce feasible individuals, a repairing technique is used to fix up all violations of dependency constraints and conflict constraints in an infeasible individual. The basic idea behind the repairing technique is iteratively identifying a pair of genes (abstract web services) at which selected concrete web services are not compatible and then trying to resolve the problem by selecting an alternative concrete web service for one or both abstract web services such that the selected two concrete web services for the two abstract web service are compatible. If an infeasible individual cannot be repaired, it is excluded from the population.

In contrast, the genetic algorithm proposed in [10] allows infeasible individuals in the population. The philosophy behind it is that an infeasible individual may have some genes or schemata that are essential to build the optimal web service selection plan and therefore it cannot afford to lose them. However, infeasible individuals are considered to be less fit than any feasible individuals and therefore they should have less chance to be selected for reproduction than any feasible individuals. In order to guarantee that, a penalty is introduced in the fitness function. It can be guaranteed by the fitness function definition that the fitness of a feasible individual is greater than that of any infeasible individuals, and that for a pair of feasible individuals the fitness value of the individual that gives better overall performance is greater than that of the individual that has worse overall performance, and that for a pair of infeasible individuals the fitness value of the individual that has less number of violations of the constraints is greater than that of the individual that has more number of violations of the constraints if their overall performances are the same.

In this paper we propose a new genetic algorithm for the web service selection problem. Different from the two existing algorithms, this genetic algorithm is a hybrid one as it utilizes a local optimizer to improve the fitness value of those individuals in the population. The hybrid genetic algorithm is explained in detail in the following section.

IV. A Hybrid Genetic Algorithm

This section elaborates our hybrid genetic algorithm. This hybrid genetic algorithm uses a local optimizer to improve the individuals in the population and utilizes a knowledge-based crossover operator.

A. Genetic Encoding

An individual in the population of our hybrid genetic algorithm represents a web service selection plan and it is encoded in an array of n integers $x_1 x_2 \cdots x_n$, where n is the total number of abstract web services in the workflow of the composite web service. In the genetic encoding scheme, each gene represents an abstract web service in the composite web service and a value of the gene represents a concrete web services of the abstract web service. Fig. 2 illustrates the encoding scheme.

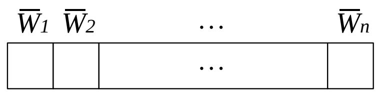


Fig. 2. Genetic encoding scheme

B. Fitness function

As discussed above, some individuals generated by the crossover and mutation operators may be infeasible. Thus, the GA must address this issue.

Infeasible individuals may have some schemata that are essential to build the optimal solution. If the infeasible individuals are excluded, the GA may not produce an optimal or near-optimal solution. Thus, the strategy adopted by our GA is to allow infeasible individuals in the population, but gives a penalty to their fitness values. The following two general guidelines are used when defining the fitness function: firstly, it should be guaranteed that an infeasible individual has less fitness value than any feasible individual. Secondly, an infeasible individual that violates more constraints should be more harshly penalized than an infeasible individual that violates less constraints. Equation 1 gives the definition of the fitness function.

$$Fitness(X) = \begin{cases} 0.5 + 0.5 * F(X), & \text{if } V(X) = 0; \\ 0.5 * F(X) - \frac{V(X)}{V_{max}}, & \text{otherwise.} \end{cases} \tag{1}$$

where $F(X)$ is the objective function defined in the problem formulation, $V(X)$ stands for the total number of constraint violations in X, and V_{max} is the maximal number of possible constraint violations. Thus, when $V(X)$ equals to zero, it implies X is a feasible individual; otherwise, X is an infeasible individual.

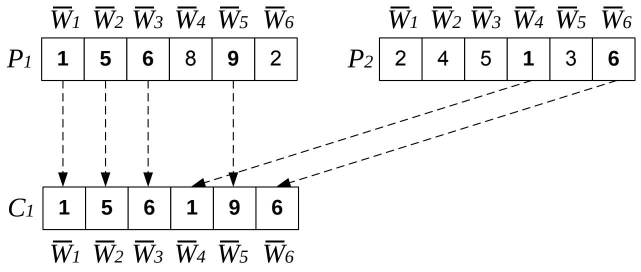
According to Equation 1, if an individual X is feasible, then its fitness value is given by the expression $0.5 + 0.5 * F(X)$. If an individual X is infeasible, then its fitness value is calculated by the expression $0.5 * F(X) - \frac{V(X)}{V_{max}}$, in which the component $\frac{V(X)}{V_{max}}$ is the penalty given to the infeasible individual X. Thus, the more constraints that an infeasible individual violates, the more penalty it receives.

It can be seen from Equation 1 that the value of the expression $0.5 + 0.5 * F(X)$ is between 0.5 and 1.0 (the value of the objective function $F(X)$ is between 0 and 1) and the value of the expression $0.5 * F(X) - \frac{V(X)}{V_{max}}$ is less than 0.5. Thus, it can be guaranteed that an infeasible individual has less fitness value than any feasible individual.

C. Genetic operators

Different from the crossover operator used in the penalty-based genetic algorithm and the repairing-based genetic algorithm, the crossover operator used in the hybrid genetic algorithm is a knowledge-based one. The knowledge-based crossover operator takes two parents, p_1 and p_2, and produces two children c_1 and c_2.

When producing c_1, firstly the crossover operator identifies all the concrete web service selections in p_1 that do not violate any constraints, and then copies these concrete web service selections to c_1. The rest concrete web service selections in c_1 are copied from p_2. Fig. 3 illustrates the ideas. In the figure, we assumed that every highlighted concrete web service selection does not conflict with any other concrete web service selection p_1. Thus, those selections are copied to c_1. Now, c_1 has the concrete web service selection for abstract web services $\overline{W}_1$, $\overline{W}_2$, $\overline{W}_3$ and $\overline{W}_5$. For the rest abstract web services $\overline{W}_4$, $\overline{W}_6$, their concrete web service selections are copied from the corresponding concrete web service selections in p_2.

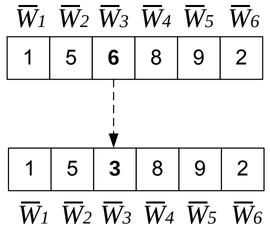


$\overline{W}_1$ $\overline{W}_2$ $\overline{W}_3$ $\overline{W}_4$ $\overline{W}_5$ $\overline{W}_6$
P_1 1 5 6 8 9 2
P_2 2 4 5 1 3 6
C_1 1 5 6 1 9 6

Fig. 3. Knowledge-based crossover operator

Similarly, when producing c_2, firstly the crossover operator finds all the concrete web service selections in p_2 that do not violate any constraints, and copies them to c_2. The rest concrete web service selections in c_2 are copied from p_1.

The mutation operator is the same as the mutation operator used in the penalty-based genetic algorithm and the repairing-based genetic algorithm. It randomly selects a concrete web service selection for an abstract web service and replaces the concrete web service selection with an alternative concrete web service of the abstract web service. Fig. 4 illustrates the mutation operator.



$\overline{W}_1$ $\overline{W}_2$ $\overline{W}_3$ $\overline{W}_4$ $\overline{W}_5$ $\overline{W}_6$
1 5 6 8 9 2
1 5 3 8 9 2

Fig. 4. Mutation operator

D. Local Optimizer

Given an individual (solution), which may or may not be feasible, the local optimizer is to optimize the individuals in the population. The local optimizer is used at the beginning of the genetic algorithm to improve the individuals in the initial population, which are randomly generated, and at the end of each generation to improve the individuals in the population.

The local optimizer improves the fitness value of an individual by increasing its overall QoS value and reducing the number of constraint violations, if any, simultaneously. This is done by systematically checking all the concrete web service selections one by one to see if there exists an alternative concrete web service that gives the individual a better fitness value. If the fitness value is improved, then the current web service selection is replaced with the alternative concrete web service. According to the definition of the fitness function, when the fitness value of an individual increases either the overall QoS value increases, or the number of constraint violations, if any, decreases, or both. Thus, the local optimizer contributes to both maximizing the overall QoS value and minimizing the number of constraint violations of an individual.

An abstract web service may have many implementations (concrete web services). Thus, if we check all the alternatives of a concrete web service selection, it would be a time consuming work as each time we check an alternative, we need to re-calculate the new fitness value of the individual, which is computationally expensive as the fitness function calculation involves a calculation of the response time, which is transformed into the problem of computing the critical path of the workflow. Thus, in order to reduce the computation time, the following strategies are adopted. If the current concrete web service selection does not violate any constraint, then we check the alternatives one by one by in the decreasing order of their weighted QoS value. Once a better alternative concrete web service is found, we replace the current concrete web service selection with the alternative immediately and then move onto the next concrete web service selection. However, if the current concrete web service selection violates any constraints, we check through all the alternatives and pick up the alternative that gives the best fitness value. By doing this, we can strengthen the constraint violation repairing ability of the local optimizer. This has been proven by experiments.

In order to improve the computation time of the local optimizer, instead of sorting all the concrete web services for each of the abstract web services every time we use the local optimizer, we sort them only once at the beginning of the genetic algorithm (before the local optimizer is invoked). Algorithm 1 is the algorithm description for the local optimizer.

E. Algorithm Description

Having defined our fitness function, genetic operators, initial population generation and local optimizer, our hybrid

Algorithm 1 Local optimizer

```
randomly generate a sequence of abstract web services, $\overline{W}_{x_1}\overline{W}_{x_2}\cdots\overline{W}_{x_n}$;
for $x$ = $x_1$ to $x_n$ do
  if the concrete web service selection at position $x$ does not violates any constraints then
    while each concrete web service $w$ of $\overline{W}_x$ do
      calculate the fitness value of the new web service selection plan;
      if the new fitness value is grater than the best fitness value then
        update the web service selection plan;
      end if
    end while
    replace the concrete web service selection with the best alternative;
  else
    while each concrete web service $w$ of $\overline{W}_x$ do
      calculate the fitness value of the new web service selection plan;
      if the new fitness value is grater than the old fitness value then
        replace the web service selection with the alternative and exit;
      end if
    end while
  end if
end for
output $X$.
```

Algorithm 2 A hybrid genetic algorithm for the web service selection problem

```
randomly create an initial population of $PopSize$ individuals, $Population$;
for each individual in $Population$ do
  optimize it using the local optimizer and then calculate its fitness value using the fitness function
end for
find the best individual $best$ in the initial population and store its fitness value into $best\_fitness$;
while termination condition is not true do
  for ∀x∈ $Population$ do
    calculate its fitness value, $F(x)$;
    if the fitness value is greater than $best\_fitness$ then
      $best = x$;
      $best\_fitness$ = $F(x)$;
    end if
  end for
  select individuals from $Population$ using the roulette selection strategy and pair them up;
  for each pair of the selected parents do
    probabilistically use the crossover operator to produce two children, $child1$ and $child2$;
    probabilistically use the mutation operator to mutate $child1$ and $child2$;
    replace the parents with the children;
  end for
end while
output $best$.
```

genetic algorithm for the web service selection problem is now presented as Algorithm 2.

V. Evaluation

In order to test the performance of our hybrid genetic algorithm, we implemented it in Microsoft Visual C# 2005. The penalty-based genetic algorithm and the repairing-based genetic algorithm were also implemented in the same programming language previously. The computation time and optimality of the new genetic algorithm were tested in comparison with the penalty-based genetic algorithm and the repairing-based genetic algorithm.

The parameter settings for the three genetic algorithms are shown in Table I. The termination condition of the three genetic algorithms was 'no improvement on the best solution in 15 consecutive generations'. All the experiments were conducted in a desktop computer with a 2.33 GHz Intel Core 2 Duo CPU and a 1.95 GB RAM.

The computation time and optimality of the genetic algorithms depend on the size and the complexity of the web service selection problem. The size of the problem is dependent on two parameters, the number of abstract web services in the workflow and the number of concrete web services for each of the abstract web services. The complexity of the problem largely depends on the number of dependencies and conflict constraints in the problem. Thus, we generated three sets of test problems.

TABLE I
GA Parameters Setting

Parameters	PGA	RGA	HGA
Population size	100	100	30
Crossover rate	0.90	0.90	0.90
Mutation rate	0.15	0.15	0.15

The first set of test problems included 10 test problems with different number of abstract web services. The number of abstract web services ranged from 10 to 100 with an increment of 10. This was done by concatenating a certain number of the composite web service workflow shown in Fig. 1. For example, when generating a composite web service workflow with 30 abstract web services, we concatenated three copies of the workflow shown in the figure. The number of concrete web services for each of the abstract web services was fixed to 30. The number of constraints was fixed to 60. This set of test problems was used to test how the computation time and optimality of the genetic algorithms would change with the number of abstract web services.

The second set of test problems also included 10 test problems. The number of concrete web services for each of the abstract web services ranged from 10 to 100 with increment of 10. The number of abstract web services was fixed to 10 and the total number of constraints was fixed to 60. This set of test problems was designed to test how the computation time and optimality of the genetic algorithms would change with the number of concrete web services.

The third set of test problems consisted of 10 test problems with different numbers of constraints. The number of constraints in the test problems ranged from 50 to 500 with increment of 50. The number of abstract web services was fixed to 10, and the number of concrete web services for each of the abstract web services was fixed to 10. Since the number of abstract web services and the number of concrete web services were all fixed, the number of constraints reflected the constraints density. This set of test problems was used to test how the computation time and optimality of the genetic algorithms would change with the complexity of the problem.

In all the generated test problems, the execution path was randomly selected and the number of iterations of the loops was a randomly generated value between 0 and 6. The QoS values for concrete web services were randomly generated as well. The ranges of the values for response time, price, reputation, availability and reliability were $[1, 10]$, $[1, 10]$, $[1, 5]$, $[0.9, 1]$, and $[0.9, 1]$, respectively, and the weighting given to the QoS criteria were fixed to 0.4, 0.3, 0.1, 0.1, and 0.1, respectively.

Firstly, we used the three genetic algorithms to solve each of the test problems in the first test set. Considering the stochastic nature of the genetic algorithms, we repeated each of the experiments 30 times and then calculated the average fitness value and average computation time for each of the experiments for each of the genetic algorithms. Table II shows the average fitness values obtained by the three genetic algorithms for the 10 test problems in the first test set and Fig. 5 displays how the average computation times of the three genetic algorithms changed when the number of abstract web services changed from 10 to 100. In the figures and tables in this paper, PGA, RGA and HGA stand for the penalty-based genetic algorithm, the repairing-based genetic algorithm and the hybrid genetic algorithm, respectively.

It can be seen from the table that the hybrid genetic algorithm found the best fitness value for six of the 10 test problems, the penalty-based genetic algorithm and the repairing-based genetic algorithm found two each. Overall, the hybrid genetic algorithm can find better quality of solutions than the other two genetic algorithms. It can be seen from the figure that the computation time of the three genetic algorithms were not correlated with the number of abstract web services. In addition, the hybrid genetic algorithm was the slowest one of the three genetic algorithms. However, the longest time it took to solve the test problem was less than 40 seconds, which is acceptable as the optimal web service selection is done at design time rather than at runtime.

Then, we used the three genetic algorithms to solve each of the test problems in the second test set. We repeated each of the experiments 30 times and then calculated the average fitness value and average computation time for each of the experiments for each of the genetic algorithms. Table III shows the average fitness values obtained by the three genetic algorithms for the 10 test problems in the second test set and Fig. 6 displays how the average computation times of the three genetic algorithms changed when the number of concrete web services changed from 10 to 100.

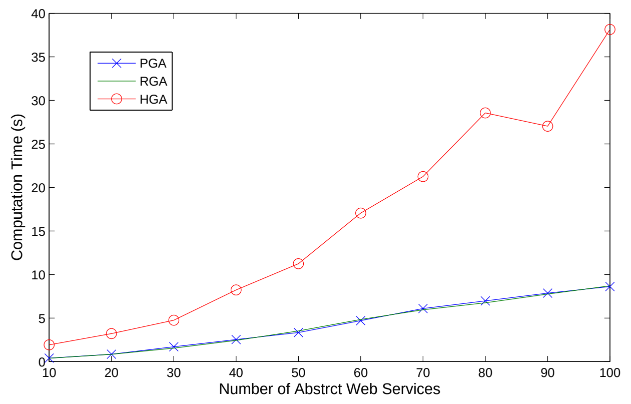


Fig. 5. The effect of the number of abstract web services on the computation time

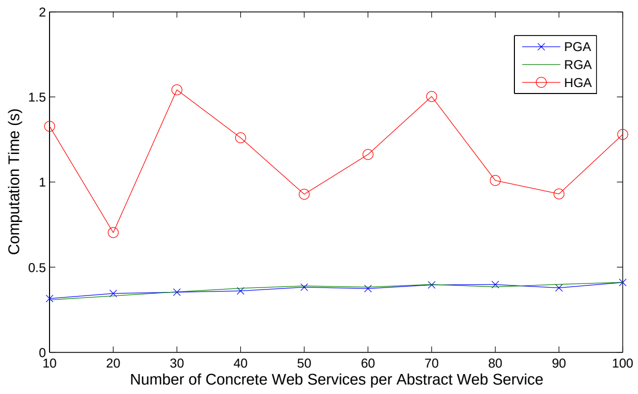


Fig. 6. The effect of the number of concrete web services per abstract web service on the computation time

It can be seen from the table that the hybrid genetic algorithm outperformed the other two genetic algorithms as it found the best fitness value for all the 10 test problems. It can be seen from the figure that the computation time of the three genetic algorithms increased linearly with the number of concrete web services per abstract web service. The hybrid genetic algorithm took longer time than the other two genetic algorithms for all of the test problems in the second test set. However, the longest one was only less than 2 seconds, which still is quick enough.

Finally, we used the three genetic algorithms to solve the test problems in the third test set one by one. We repeated each of the experiments 30 times and then calculated the average fitness value and average computation time for each of the experiments for each of the genetic algorithms. Ta-

TABLE II

COMPARISON RESULTS OF PGA, RGA AND HGA FOR TEST PROBLEMS WITH DIFFERENT NUMBERS OF ABSTRACT WEB SERVICES

Problem	Abstract Services #	PGA		RGA		HGA	
		Avg. Fitness	Dev. (%)	Avg. Fitness	Dev. (%)	Avg. Fitness	Dev. (%)
1	10	0.88703	0.07	0.88722	0.07	0.89033	0
2	20	0.86601	0.21	0.86726	0.13	0.86769	0.03
3	30	0.85499	0.22	0.85506	0.19	0.85694	0.03
4	40	0.85978	0.16	0.85967	0.17	0.86154	0.06
5	50	0.85580	0.18	0.85660	0.13	0.85643	0.10
6	60	0.85257	0.14	0.85327	0.12	0.85274	0.11
7	70	0.86456	0.17	0.86512	0.16	0.86647	0.06
8	80	0.85821	0.15	0.85816	0.17	0.85785	0.11
9	90	0.85844	0.19	0.85953	0.19	0.86336	0.05
10	100	0.85137	0.14	0.85299	0.13	0.85652	0.08

TABLE III

COMPARISON RESULTS OF PGA, RGA AND HGA FOR TEST PROBLEMS WITH DIFFERENT NUMBER OF CONCRETE WEB SERVICES FOR EACH ABSTRACT WEB SERVICE

Problem	Concrete Services #	PGA		RGA		HGA	
		Avg. Fitness	Dev. (%)	Avg. Fitness	Dev. (%)	Avg. Fitness	Dev. (%)
1	10	0.84133	0.39	0.84013	0.14	0.84747	0
2	20	0.87848	0.02	0.87856	0.01	0.87859	0
3	30	0.88633	0.08	0.88657	0.03	0.89116	0.20
4	40	0.87409	0.24	0.87471	0.21	0.88117	0
5	50	0.89136	0.39	0.89266	0.17	0.89361	0
6	60	0.90001	0.26	0.90025	0.17	0.90131	0.04
7	70	0.90818	0.18	0.90829	0.20	0.90950	0.02
8	80	0.90812	0.29	0.90811	0.30	0.91196	0.01
9	90	0.91296	0.37	0.91552	0.35	0.91916	0
10	100	0.90521	0.29	0.90472	0.35	0.90981	0.08

ble IV shows the average fitness values obtained by the three genetic algorithms for the 10 test problems in the second test set and Fig. 7 displays how the average computation times of the three genetic algorithms changed when the number of constraints changed from 50 to 500.

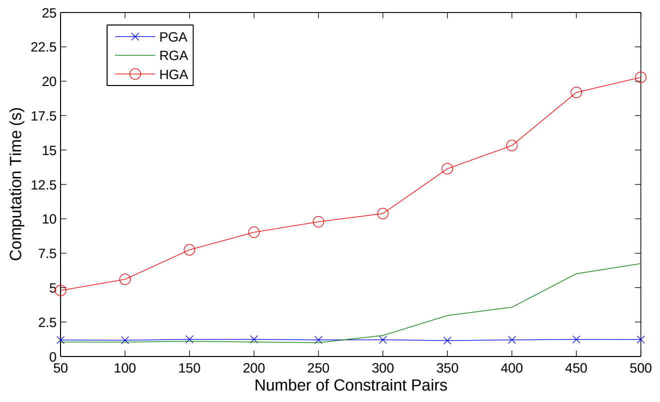


Fig. 7. The affect of constraint density on the computation time

It can be seen from the table that in overall the hybrid genetic algorithm outperformed the other two genetic algorithms as it found the best fitness value for seven of the 10 test problems. It can be seen from the figure that basically the computation time of the three genetic algorithms increased linearly with the number of constraints. The hybrid genetic algorithm took longer time than the other two genetic algorithms for all of the test problems in the third test set. However, the longest one was only less than 22 seconds, which still is quick enough.

Based on the above experimental results for the three sets of test problems, the following conclusions can be drawn:

- The hybrid genetic algorithm is as scalable as the other two genetic algorithms. Its computation time does not change significant when the number of abstract web services increases, and increases linearly when the number of concrete web services increases or when the number of constraints increases.
- The hybrid GA is also effective as the other two genetic algorithms. In overall, the hybrid genetic algorithm can find slightly better solutions than the other genetic algorithms.

TABLE IV

COMPARISON RESULTS OF PGA, RGA AND HGA FOR TEST PROBLEMS WITH DIFFERENT CONSTRAINT DENSITIES

Problem	Constraint Pairs #	PGA		RGA		HGA	
		Avg. Fitness	Dev. (%)	Avg. Fitness	Dev. (%)	Avg. Fitness	Dev. (%)
1	50	0.82891	0.14	0.82879	0.11	0.83122	0.07
2	100	0.82891	0.18	0.82843	0.08	0.82972	0.06
3	150	0.81241	0.39	0.81447	0.15	0.81671	0.04
4	200	0.80968	0.40	0.81337	0.06	0.81469	0.01
5	250	0.80215	0.44	0.80764	0.13	0.80844	0
6	300	0.73099	17.26	0.80484	0.08	0.80615	0.04
7	350	0.82891	19.19	0.79819	0.33	0.80005	0.02
8	400	0.36894	0.70	0.77433	0.30	0.76342	8.99
9	450	0.27321	0.59	0.30328	14.48	0.27315	0
10	500	0.25425	0.61	0.28810	14.28	0.26436	0.01

- The hybrid genetic algorithm is more stable than the other two genetic algorithms. This can be reflected in the derivations shown in the tables.
- The hybrid genetic algorithm is more suitable for those web service problems with a large number of abstract web services and a large number of constraints.

VI. CONCLUSION

This paper has proposed a new genetic algorithm for the optimal web service selection problem. Different from the other two genetic algorithms proposed previously, this new genetic algorithm is a hybrid one, utilizing a local optimizer to improve the fitness value of the individuals in the population at end of each generation, including the initial population. The local optimizer can improve the overall QoS value of, and reduce or eliminate the constraint violations in, a composite web service plan.

This paper has also shown our systematic evaluation results on the new genetic algorithm in comparison with the other two genetic algorithms. The evaluation results have shown that the new hybrid genetic algorithm can provide better quality of solutions than the other two genetic algorithms for almost all the test problems.

A deficiency of the new hybrid genetic algorithm is that its computation time is slightly longer than that of the other two genetic algorithms. This results from larger number of fitness evaluations used by the local optimizer. Thus, its computation time could be reduced by strategically applying the local optimizer to improve selective individuals rather than all the individuals in the population. This is an issue that we will investigate in the future.

In addition, the evaluation has shown that the performance of the new genetic algorithm may not be as stable as that of the repairing-based genetic algorithm when the constraints density is very high. Thus, a potential to further improve the performance of new genetic algorithm is to embed the repairing mechanism in the local optimizer.

ACKNOWLEDGEMENT

The authors wish to thank Nan Tian for his help with the hybrid genetic algorithm implementation and the experiments.

This research was carried out as part of the activities of, and funded by, the Smart Services Cooperative Research Center (CRC) through the Australian Government's CRC Programme (Department of Innovation, Industry, Science and Research).

REFERENCES

[1] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, "Constraint driven web service composition in METEOR-S," in *Proc. 2004 IEEE International Conference on Services Computing*, Sept. 2004, pp. 23–30.

[2] M. Jaeger, G. Rojec-Goldmann, and G. Muhl, "QoS aggregation for web service composition using workflow patterns," in *Proc. the* 8^{th} *IEEE International Conforence on Enterprise Distributed Object Computing Conference*, Sept. 2004, pp. 149–159.

[3] E. Maximilien and M. Singh, "A framework and ontology for dynamic Web services selection," *IEEE Internet Computing*, vol. 8, no. 5, pp. 84–93, Sept.-Oct. 2004.

[4] K. Verma, R. Akkiraju, R. Goodwin, P. Doshi, and J. Lee, "On accommodating inter service dependencies in web process flow composition," in *Proc. AAAI Spring Symposium on SWS*, 2004, pp. 37–43.

[5] L. Zeng, B. Benatallah, A. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, "QoS-aware middleware for web services composition," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, pp. 311–327, May 2004.

[6] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An approach for QoS-aware service composition based on genetic algorithms," in *Proc. the 2005 conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2005, pp. 1069–1075.

[7] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, vol. 33, no. 6, pp. 369–384, June 2007.

[8] T. Yu, Y. Zhang, and K.-J. Lin, "Efficient algorithms for web services selection with end-to-end qos constraints," *ACM Trans. on Web*, vol. 1, no. 1, p. 6, 2007.

[9] L. Ai and M. Tang, "QoS-based web service composition accommodating inter-service dependencies using minimal-conflict hill-climbing repair genetic algorithm," in *Proc. IEEE Fourth International Conference on e-Science*, Dec. 2008, pp. 119–126.

[10] ——, "A penalty-based genetic algorithm for QoS-aware web service composition with inter-service dependencies and conflicts," in *Proc. International Conference onComputational Intelligence for Modelling, Control and Automation,*, Dec. 2008, pp. 738–743.

[11] M. Aiello, E. el Khoury, A. Lazovik, and P. Ratelband, "Optimal QoS-aware web service composition," in *IEEE International Conference on E-Commerce Technology*, 2009, pp. 491–494.