A GENETIC ALGORITHM BASED TECHNIQUE FOR QOS-AWARE WEB SERVICE COMPOSITION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AHMET ERDİNÇ YILMAZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2013

Approval of the thesis:

A GENETIC ALGORITHM BASED TECHNIQUE FOR QOS-AWARE WEB SERVICE COMPOSITION

submitted by **AHMET ERDİNÇ YILMAZ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** ________________

Assoc. Prof. Dr. Pınar Karagöz
Supervisor, **Computer Engineering Department, METU** ________________

Examining Committee Members:

Prof. Dr. Özgür Ulusoy
Computer Engineering Department, Bilkent Uni. ________________

Assoc. Prof. Dr. Pınar Karagöz
Computer Engineering Department, METU ________________

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU ________________

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU ________________

Assoc. Prof. Dr. Hakan Ferhatosmanoğlu
Computer Engineering Department, Bilkent Uni. ________________

Date: ________________

ABSTRACT

A GENETIC ALGORITHM BASED TECHNIQUE FOR QOS-AWARE WEB SERVICE COMPOSITION

Yılmaz, Ahmet Erdinç
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Pınar Karagöz

September 2013, 61 pages

Web Service technology is one of the most rapidly developing technologies. Since Web Services are defined by several XML-based standards to overcome platform dependency, they are very eligible to integrate with each other in order to establish new services. This composition enables us to reuse existing services, which results in less cost and time consumption. Currently the main issues with Web Service Composition is to define workflow of the services and maximizing the overall Quality of Service (QoS) of the composed service. Most common elements of QoS are execution cost, execution time, availability, successful execution rate, reliability and throughput. Since the selection of the optimal execution plan that maximizes the composition's overall QoS is NP-hard problem, applying optimization techniques is very popular. In this thesis, we propose an improved Genetic Algorithm to optimize the overall QoS of the composed service.

Keywords: Web Service, Web Service Quality, Genetic Algorithm

ÖZ

WEB SERVİS BİRLEŞİMİNDE KALİTE OPTİMİZASYONU İÇİN EVRİMSEL ALGORİTMA TABANLI BİR TEKNİK

Yılmaz, Ahmet Erdinç
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Pınar Karagöz

Eylül 2013 , 61 sayfa

Günümüzde en hızlı gelişen teknolojilerden biri de Web Servisi teknolojileridir. Web servisler XML tabanlı standartlarla tanımlanması sayesinde, platform bağımlılık sorununu çözerek diğer web servisleri kullanarak yeni servisler uyarlamaya uygundur. Bu uyarlama, eski servislerin tekrar kullanılması ile kaynakların daha az tüketilmesini sağlamaktadır. Web servis alanındaki ana unsurlar servislerin nasıl bağlanacağını belirleyen birleşik servis planın tanımlanması ve oluşturulan birleşik servisin toplam servis kalitesi puanını olabildiğince arttırmaktır. Servis kalitesini belirleyen bileşenler arasında servis ücreti, çalışma zamanı, bulunabilirlik, başarılı sonuç verme, güvenilirlik ve verimlilik en popüler olanlarıdır. Optimal birleşik servis planını bulmak NP-Zor problem olması sebebi ile optimizasyon tekniklerinin kullanılması çok popülerdir. Bu tezde, birleşik web servisi oluştururken kalite servis puanını optimize etmeyi amaçlayan yeni bir geliştirilmiş genetik algoritma sunulmaktadır.

Anahtar Kelimeler: Web Servis, Web Servis Kalitesi, Genetik Algoritma

To my wife

ACKNOWLEDGMENTS

I would like to thank to Pınar Karagöz for her supervision and support through the development of this thesis. I greatly appreciate her contribution to this study and her guidance through not only development of this thesis but also in my all academic life.

I would like to thank my wife Tüluğ for her precious love. She is the sole reason for me to live and the only light in this life since I have seen her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

SOA	Service Oriented Architecture
QoS	Quality Of Service
GA	Genetic Algorithm
SA	Simulated Annealing
HS	Harmony Search
HMCR	Harmony memory consideration rate
PAR	Pitch adjustment rate
bw	Pitch adjustment bandtwidth
PSO	Particle Swarm Optimization
ACO	Ant Colony Optimization
ACS	Ant Colony System
CBR	Case-based Reasoning
BPEL	Business Process Execution Language
IP	Integer Programming
UDDI	Universal Description Discovery and Integration
CBR	Case Based Reasoning
CSMS	Composite Service Management System
OSGI	Open Services Gateway Initiative

CHAPTER 1

INTRODUCTION

Service composition is one of the solutions that builds large and complex systems that use the different services which can be implemented in different frameworks and languages. Recently Service Oriented Architecture (SOA) and use of components are the trend technologies [18] since it saves resources by reusing and integrating the services resulting in new services, which can be used for new compositions. Service composition requires two steps which are the identification of the workflow of abstract services (execution plan) and selecting the concrete services for each node in that execution plan.

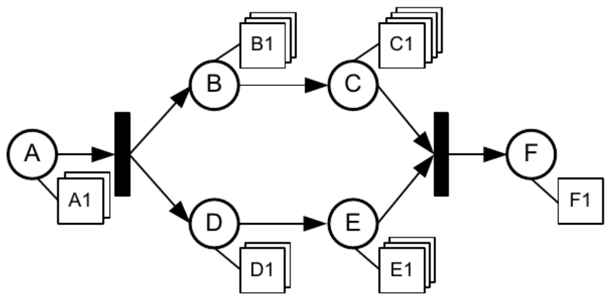


Figure 1.1: Example Service Composition [18]

In Figure 1.1, an example of service composition is presented. There are six abstract services that construct the execution plan. Abstract services B and C are connected in serial fashion. Similarly, abstract services D and C are also in serial pattern. Then they are connected in parallel. Finally resulting parallel one is connected with the abstract services A and C in serial connection producing the ultimate execution plan. There are also the concrete services for each abstract one (two for A, three for B…). However in real world, the execution plan consists of tens of thousands of concrete services with same functionalities, resulting in a need for method that can choose the most appropriate one among them [26].

Since there are many of web services that are equal in functionality, searching and selecting web service for execution plan is not an easy problem. However the functionally equal services may differ in a property that is called *Quality of Service (QoS)*. While selecting the web services, one of the basic concerns is optimizing the overall QoS value of the composition.

Finding the execution plan that optimizes the overall QoS value is a well-known NP-hard problem [18]. Therefore currently there is no an algorithm that solves the service composition problem in polynomial-time, making optimization algorithms a good choice for this problem. Although there are studies that propose various heuristic techniques they still need to be studied further to provide improvement over these techniques.

In this thesis, we present hybrid algorithms that aim to select concrete web services for a given execution plan optimizing the overall QoS value of the workflow. The most frequently used QoS parameters in the literature are “Response Time”, ”Availability” ,”Reliability”, ”Throughput” and “Price” [4, 1, 3, 16, 17], which are to be optimized during the web service composition. We assume that these values are available for each concrete service prior to execution of algorithm since there are studies that aims to calculate these values automatically [8].

1.1 Motivation

Web services and composing new complex systems using the other services are very popular in software society. Since there are many web service candidates that are functionally equivalent, selecting the most appropriate ones among them is a problem that attracts attention. Therefore, the main motivation of this thesis is to contribute to the efforts in this area. Also, although there are plenty of studies that uses heuristics for this problem, there is almost no paper make use of hybrid algorithms that may solve the current problem more efficiently. Thus, we propose improved Genetic Algorithms, which perform better than the previous ones in the literature in several dimensions . Our proposed solution converges earlier while preserving the diversity of the solutions. In addition, we show that the combination of the heuristics with adaptive techniques that changes the algorithm parameters dynamically can further improve the conventional techniques significantly.

1.2 Contributions

The main contributions of this study are:

- Two improved Genetic Algorithms for Web Service Composition under QoS Optimization are proposed and described in detail.
- Our proposed algorithms and the previous similar ones in the literature are compared and the results are discussed. Experimental results indicate improvement for fitness value, execution time and applicability on parallel workflow models by the proposed algorithms.
- Proposed algorithms are implemented in fully modularized framework OSGI which enables that resulting modules can be used in other systems easily.

1.3 Thesis Organization

This thesis is organized as follows:

Chapter 2 includes the related work on QoS Aware Web Service Composition. The studies and the algorithms used are described.

In Chapter 3, the proposed hybrid genetic algorithms are described in detail. Our algorithm implementation that is a fully modularized software library and test software are explained.

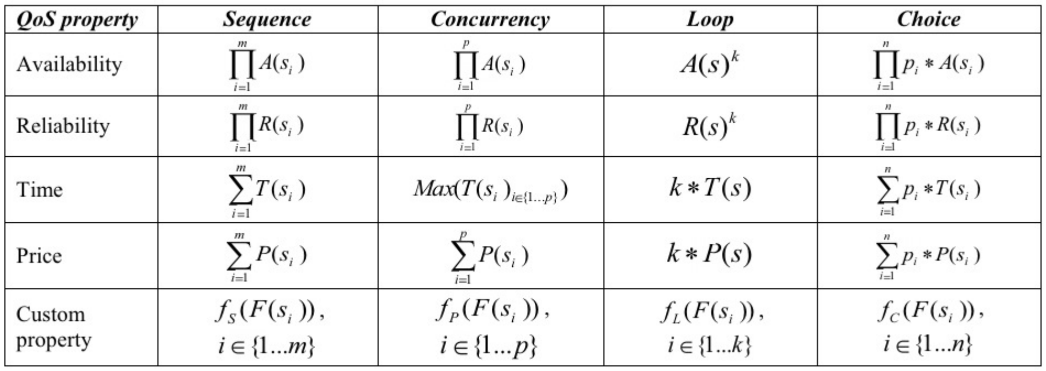
In Chapter 4, case study and evaluations of the proposed algorithms are given. Our proposed algorithms are compared with the similar algorithms in the literature.

In Chapter 5, conclusion and future work are discussed.

CHAPTER 2

RELATED WORK

QoS-Aware Web Service Composition is studied by several research groups in the literature. The aim of composition is selecting the services such that resulting QoS is maximized. Therefore the problem can be modeled as multi-dimensional, multi-objective, multi-choice knapsack problem (MMMKP). Since the problem is NP-hard, it takes huge amount of effort to solve the problem, all studies in the literature assume some simplifications in the problem such as "local QoS-optimization", "no QoS-Constraints", "single-objective optimization" and "sub-optimal execution plan". In the literature there are several algorithms that can find exact solutions such as "Linear Programming", "Integer Programming", "Dynamic Programming" and "Divide-and-Conquer based Selection". Anja Strunk states that heuristics (such as Genetic Algorithm) is promising method in QoS-aware composition. He also states that the heuristics other than the Genetic Algorithm should be experienced such as Simulated Annealing ,Taboo Search and Ant Colony Optimization [18, 19, 11]. Therefore this survey is the main start point of our study.



QoS property	*Sequence*	*Concurrency*	*Loop*	*Choice*
Availability	$\prod_{i=1}^{m} A(s_i)$	$\prod_{i=1}^{p} A(s_i)$	$A(s)^k$	$\prod_{i=1}^{n} p_i * A(s_i)$
Reliability	$\prod_{i=1}^{m} R(s_i)$	$\prod_{i=1}^{p} R(s_i)$	$R(s)^k$	$\prod_{i=1}^{n} p_i * R(s_i)$
Time	$\sum_{i=1}^{m} T(s_i)$	$Max(T(s_i)_{i \in \{1 \ldots p\}})$	$k * T(s)$	$\sum_{i=1}^{n} p_i * T(s_i)$
Price	$\sum_{i=1}^{m} P(s_i)$	$\sum_{i=1}^{p} P(s_i)$	$k * P(s)$	$\sum_{i=1}^{n} p_i * P(s_i)$
Custom property	$f_S(F(s_i))$, $i \in \{1 \ldots m\}$	$f_P(F(s_i))$, $i \in \{1 \ldots p\}$	$f_L(F(s_i))$, $i \in \{1 \ldots k\}$	$f_C(F(s_i))$, $i \in \{1 \ldots n\}$

Figure 2.1: Aggregation Formulas For Overall QoS [18]

In Figure 2.1, there can be seen the basic control patterns, which are Sequence (Serial), Concurrency (Parallel), Loop and Choice. Also there are different aggregation rules for each control pattern to compute the overall QoS value of the execution plan. For example in Sequence, the response time for overall composition, we need to sum all of the response time values in the services that constructs the pattern.

2.1 Genetic Algorithms

Genetic Algorithm (GA) is a variant of stochastic beam search in which successor states are generated by two parents that mimics the natural evolution in the nature. It is heavily used in optimization and search problems [5].

Algorithm 1 Genetic Algorithm [5]

```
function GENETIC-ALGORITHM($population$,FITNESS-FN)
    $new\_population$ = empty set
    repeat
        for $i = 1 \rightarrow SIZE(population)$ do
            $x$ = RANDOM-SELECTION($population$,FITNESS-FN)
            $y$ = RANDOM-SELECTION($population$,FITNESS-FN)
            $child$ = REPRODUCE($x$,$y$)
            if small random probability then
                $child$ = MUTATE($child$)
                add $child$ to $new\_population$
            end if
            $population$ =$new\_population$
        end for
    until some individual is fit enough, or enough time has elapsed
return the best individual in $population$, according to FITNESS-FN
end function

function REPRODUCE($x$,$y$)
    $n$ = LENGTH(x);
    $c$ = random number from 1 to $n$
return APPEND(SUBSTRING($x$,1,$c$),SUBSTRING($y$,$c+1$,$n$))
end function
```

GAs begin with randomly generated states (population of individuals). Each individual can be encoded in some format such as strings. Moreover, for specific domains specialized data structures are helpful. Algorithm runs for fixed number of generations in which new off-springs are generated by parents that are ranked by the *fitness functions*. Generally, fitness function return higher values for better individuals. The parents with higher fitness values, have higher chance for reproducing new off-springs than the ones with lower fitness values. New off-springs are generated from these parents' genomes in the operation called *crossover*. This operation enables the new individual has both genes from both parents. Then each generated individual is subject to a process called *mutation* with low probability. Mutation changes their genome in some randomly chosen point. This operator resembles the mutation in nature that increases the population diversity. This operator is important because, in order to find fitter individuals, population diversity should be high enough. *The schema theorem* explains how GA works. Assume that the individuals genome are represents as numerical strings of length eight. Then the schema "234*****" represents all individuals whose genome starts with sub-string "234". Then the individual having genome "23476585" is an *instance* of this schema. It can be shown that if the fitness of the schema "234*****" is higher than the mean of the population then the number instances of this schema tends to increase. This results in that fitter individuals dominate the population over time [5].

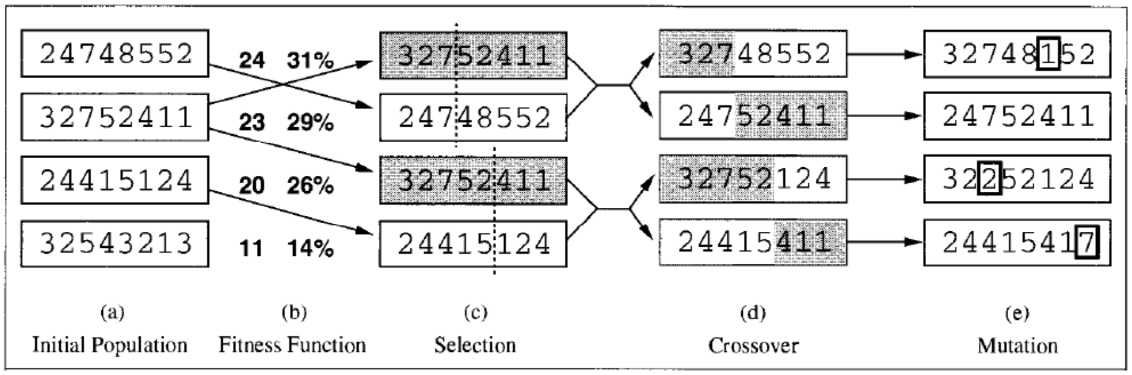


Figure 2.2: Genetic Algorithm Operators [5]

In Figure 2.2 the main operators of genetic algorithm are presented. The individuals are represented with numerical strings of length eight. Initial population consists of four individuals having fitness values 24,23,20,11, respectively. Moreover these fitness values determine the probability that individuals will generate the new off-springs in the next generation which can bee seen in (c) part (*selection* operator). Then the *crossover* operator takes place before the child individuals come up when the *mutation* operator change the genotypes slightly.

There are several studies that use GA for QoS aware web service composition. These studies can be grouped by their objective function type, *Single Objective Approach* and *Multi-Objective Approach*. In multi-objective approaches, there are more than one objective functions, possibly having conflicts with each other. In this technique, there are set of solutions that no other solution has better in all objectives. In single objective however, multiple objectives can be combined with weights in order to reduce to problem to single objective. Unlike the multi-objective one, we get one solution that is better than the all other individuals.

2.1.1 Single Objective Approach

Fanjiang, Syu and Wu et al. proposed a method that not only composes the service according to the overall QoS value but also generates the execution plan from user requirements based on GA and case-based reasoning [23]. Therefore their algorithm consists of two phases, selecting equivalent service sets with the execution plan construction and then selecting the services in each equivalent set, which optimizes the overall QoS.

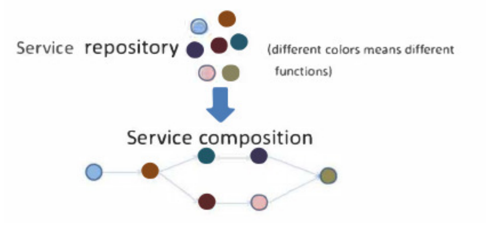


Figure 2.3: Process of Service Composition [23]

In Figure 2.3, there can be seen the first phase of the algorithm which constructs the workflow in sequential, conditional and parallel patterns. Different colors mean that services differ in functionality. They also state that if they used exact algorithms such that linear programming, the problem even would be insoluble because of the nature of the problem (NP-hard) [23]. They use unified specification for user requirements and then try to find a single solution from the service registry. At first they try to find a single service that meets the requirement, if they cannot find, their case-based reasoning method runs. However they do not explain this part of their algorithm in detail. They state that they use GA in finding the first workflow. After they found one candidate execution plan, the main part of their algorithm runs which can be seen in Figure 2.4.

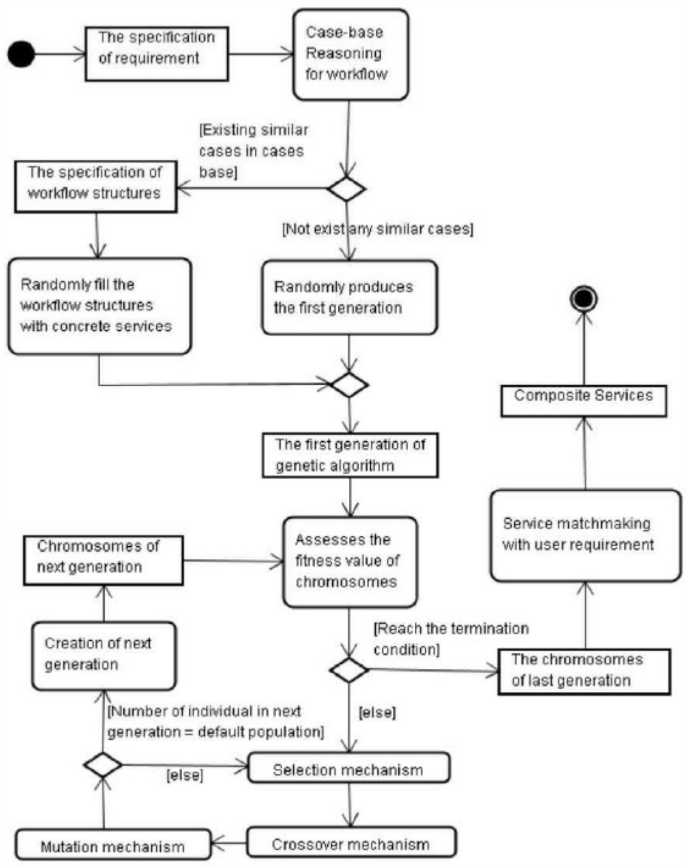

The specification of requirement
Case-base Reasoning for workflow
[Existing similar cases in cases base]
The specification of workflow structures
[Not exist any similar cases]
Randomly fill the workflow structures with concrete services
Randomly produces the first generation
Composite Services
The first generation of genetic algorithm
Chromosomes of next generation
Assesses the fitness value of chromosomes
Service matchmaking with user requirement
Creation of next generation
[Reach the termination condition]
The chromosomes of last generation
[Number of individual in next generation = default population]
[else]
[else]
Selection mechanism
Mutation mechanism
Crossover mechanism

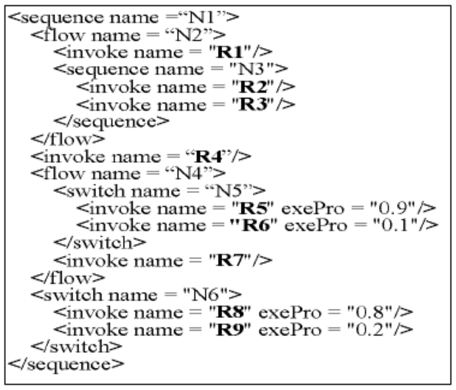
Figure 2.4: Genetic Algorithm for dynamic service composition [23]

The algorithm in Figure 2.4 tries to find a solution that optimizes the overall QoS value. Moreover they use WS-BPEL process description language to encode the chromosomes in the genetic algorithm. In order to avoid the genetic algorithm operators crossover and mutation work without resulting in invalid execution plans, they translated the WS-BPEL to tree format. The most interesting part in their view, there can be many invalid individuals in the population since the execution plans in the population change in time. This paper differs mostly in here from other studies. They use a penalty mechanism in that case, which decreases the fitness value of the individual if it has an invalid workflow. Another drawback is that, with mutation and crossover operators resulting execution plans can be very long, which are effectively useless.

Yue and Chengwen proposed an improved GA named CoDiGA with enhanced initial population, enhanced population evaluation and a different chromosome structure, matrix coding scheme [24]. They also take the global QoS Constraints into consideration in their study. They claim that their matrix coding scheme is very suitable for handling constraints. Their main considerations are slow convergence of existing genetic algorithms and their unstability. However their new method can only bring new fitness values in quick convergence when the execution plan is big. Their relation matrix encoding scheme carries information both about the relation among tasks and path information which are used in validity check of new off-springs. One interesting part of their algorithm is that the child chromosomes can have different execution plans than their parents. This is also the case in the previous study[23]. Since the execution plan can change, in run time a need for validity checking emerges, which makes the execution takes more time. In addition, all valid workflows cannot be beneficial.

$$e = \frac{\overline{Fit(g)}}{1 + H(n)} \tag{2.1}$$

As another point in their study, they propose new selection mechanism in their genetic algorithm. At the initial step they do not replace the current generation with the next one immediately, the replacement occurs when the next generations *evolution extent* is higher than the current one. The evolution extent is population mean fitness value divided by population diversity rate plus one, which can be seen in the formula 2.1. Since population with low diversity is preferred if the mean fitness values are equal, this will lead to chance to run away from global optima. Although they claim that it is useful for quick convergence, I think it should be avoided especially in earlier stages of algorithm. In later stages, however, low diversity should be preferred as we will describe in our proposed algorithms. To improve convergence rate, they also make sure that all individuals of the initial population have higher fitness values than the ones that are generated randomly. Although this technique is useful for quick convergence, it takes the population away from high density, which may result in not reaching the global optima. Instead, in some proportion, the population can be generated with high fitness values. Moreover, one different view of their algorithm is that, in every generation they allow selection between two successor generations resulting in larger search space. Then they shrink the population to the original size. They do not state explicitly, but I think that they select the best ones in the final generation. For constraint handling, they use a penalty mechanism, which is a common usage in literature. As a further research, they suggest the use of stochastic search techniques such as Simulated Annealing (SA), which is one of our proposed algorithms with improved usage.



```
<sequence name ="N1">
  <flow name = "N2">
    <invoke name = "R1"/>
    <sequence name = "N3">
      <invoke name = "R2"/>
      <invoke name = "R3"/>
    </sequence>
  </flow>
  <invoke name = "R4"/>
  <flow name = "N4">
    <switch name = "N5">
      <invoke name = "R5" exePro = "0.9"/>
      <invoke name = "R6" exePro = "0.1"/>
    </switch>
    <invoke name = "R7"/>
  </flow>
  <switch name = "N6">
    <invoke name = "R8" exePro = "0.8"/>
    <invoke name = "R9" exePro = "0.2"/>
  </switch>
</sequence>
```

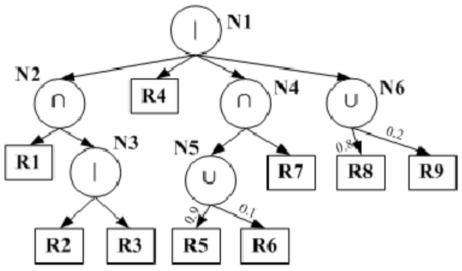


Figure 2.5: Tree expression of given work-flow [9]

In another study, Chunming, Meiling and Huowang propose a data structure which can support composition re-planning at run-time and improves the GA significantly [9]. The idea behind their study is to make the data structure carry the information not only about the static workflow information but also about the dynamic information on the QoS values. In Figure 2.5 you can see the execution plan in XML based file consisting of sequence and switch control patterns, which is transformed to the tree structure. The main point in here is that, they also need to tag the nodes in tree structure. This tree is represented in linear form in the chromosome using post-order traversal which can be seen in the Figure 2.6. In this representation each non-leaf node is a valid execution plan tree and carry information about its children and each leaf node is concrete web service. By this representation they are able to define crossover and mutation operators not producing invalid process plans.

$$R_1 R_2 R_3 N_3 N_2 R_4 R_5 R_6 N_5 R_7 N_4 R_8 R_9 N_6 N_1$$

Figure 2.6: Post-order chromosome representation of the tree structure [9]

Moreover, non-leaf nodes also carry information about the calculated fitness value, which is used again if the sub-tree remains unchanged after the crossover operator. This technique results in faster running time than the one-dimensional GAs. In addition tree structure enables them to re-plan the workflow more easily. For global constraints they use a penalty factor in fitness, which is a common technique in literature.

Another approach in genetic algorithm is on controlling the diversity factor of the population in order to avoid getting stuck in local optima and to accelerate the convergence rate. Changwen proposes an adaptive crossover rate based genetic algorithm which operates as in [25]. The main idea is that, when the population diversity falls below some extent, the crossover rate should be increased in order to create new individuals whose genomes are different. When the population diversity climbs above some value, the rate should be decreased in order to improve the convergence rate. They take into account the mean population fitness. When the mean population fitness value is high, they reduce the crossover rate in order to prevent the fit individuals being damaged. Similarly, when the mean population fitness value is low, they increase the rate in order to advance the search capability of the algorithm and the convergence speed.

$$D_m = \frac{1}{N}\sum_{i=1}^{N}\left(f_m^i - \overline{f_m}\right)^2 \tag{2.2}$$

$$\Omega = D_m / D_{max} \tag{2.3}$$

They define the population variance as in formula 2.2, where f_m^i is the fitness value of the i^{th} individual and $\overline{f_m}$ is the average fitness value of the whole population. Then they calculate the diversity of the m^{th} generation Ω, as the m^{th} generation fitness variance D_m divided by D_{max} which is the maximum fitness variance value up to m^{th} generation as given in formula 2.3. Then they propose the adaptive crossover rate in formula 2.4.

$$P_c = P_{c-old} \times \left[1 - \frac{f_i - f_{Min}}{f_{Max} - f_{Min}} \times \frac{1}{1 + exp(-k_c\Omega)}\right] \tag{2.4}$$

In this formula, P_{c-old} is the old crossover probability, f_i is the fitness of the individual of X_i, f_{Max} and f_{Min} are the maximum and minimum fitness values in the generation and k_c is a constant. Note that, the greater the P_c value the easier to find different individuals which may have greater fitness value. However if this rate is not controlled and gets very high, it results in slower convergence rate. Therefore the adaptive nature of this formula tries to control it optimally. You can see the overall flow of the improved GA in Figure 2.7.

With this dynamic crossover rate they get higher fitness values with less running time than the traditional genetic algorithm. Their approach for calculating variance in fitness would be more effective if they computed the variance from genomes instead of fitness values. Because different genomes may result in the same fitness values. By this design they prevent these individuals from contributing the population diversity. Moreover, I think adaptive crossover rate will be more successful if the mutation rate is also dynamically.

Another aspect of QoS optimization algorithms are the constraints. There can be conflicts or dependencies between concrete services such as if service A is used, service B must also be used ore if service A is used service B cannot be used [7]. Moreover there can be user limits on the QoS constraints such that total cost should be less than some value [15]. The common method in literature is applying penalty mechanism [9, 7, 15]. The main reason applying penalty mechanism instead of removing the conflicted individual from the population is that

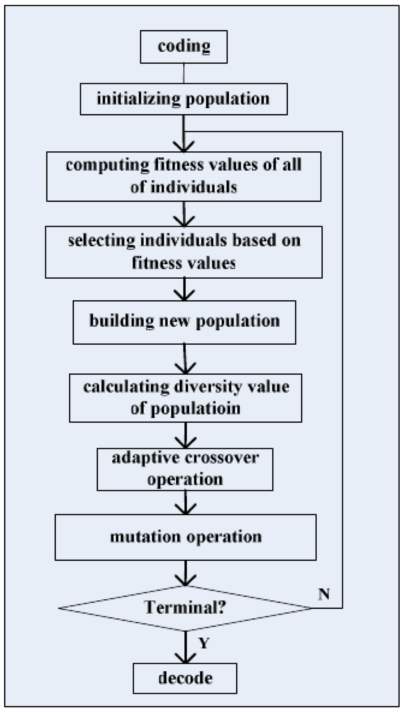

coding
initializing population
computing fitness values of all of individuals
selecting individuals based on fitness values
building new population
calculating diversity value of populatioin
adaptive crossover operation
mutation operation
Terminal?
N
Y
decode

Figure 2.7: Adaptive genetic algorithm[25]

they can have partial genes in their chromosomes which can result in global optima after several crossover end mutation operations.

Lifeng and Maolin consider the inter dependencies and conflicts problem between the web services [7]. They define set of dependencies between the services as D:

$D = \{(c_{i_1 j_1}, c_{i_2 j_2})|$ if task i_1 uses the j_1^{th} service, then task i_2 must also use the service $j_2^{th}\}$. Moreover they define a set of conflicts between services as C:

$C = \{(c_{i_1 j_1}, c_{i_2 j_2})|$ if task i_1 uses the j_1^{th} service, then task i_2 must not use the service $j_2^{th}\}$.

$$F_{obj}(X) = \sum_{l=1}^{2}\left(\frac{Q_l^{max} - Q_l(X)}{Q_l^{max} - Q_l^{min}} \star W_l\right) + \sum_{l=3}^{5}\left(\frac{Q_l(X) - Q_l^{min}}{Q_l^{max} - Q_l^{min}} \star W_l\right) \tag{2.5}$$

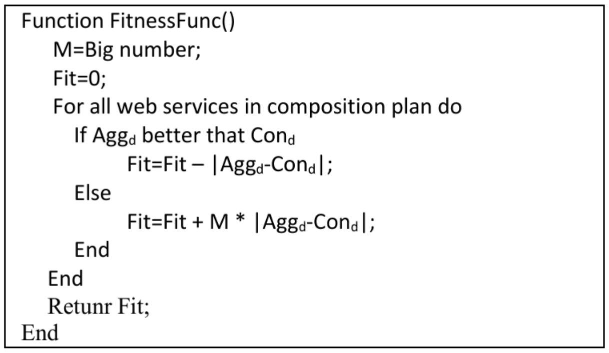
They define the objective function in a conventional way in the literature. In the first part of the function they try to minimize the negative QoS parameters such as cost, response time and in the second part of the sum, they try to maximize the positive QoS parameters such as reputation, reliability and availability in formula 2.5.

$$Fitness(X) = \begin{cases} 0.5 + 0.5 \star F_{obj}(X), & \text{if } V(X) = 0 \\ 0.5 \star F_{obj}(X) - \frac{V(X)}{V_{max}}, & \text{otherwise} \end{cases} \tag{2.6}$$

Then they finally define the penalty based fitness function in formula 2.6 where $V(X)$ is the

the number of violations of X, and V_{max} is the maximum number of constraint violations. The clever point of this fitness design is that they guarantee that any infeasible individual has less fitness value than any individual without any constraints. With these designs Lifeng and Maolin can find a feasible individual with good QoS value in their all tests.

Mahmood and Hadi, on the other hand, consider the limit constraints on the QoS values [15]. For this, they define a penalty based fitness function that assigns smaller values for feasible and desired concrete services with good overall QoS values as given in Figure 2.8. In this function, Agg_d is the aggregated QoS value that is calculated as in conventionally in the literature. They are also normalized in the interval [0,1]. M is a big number, and for every web service in the execution plan they decrease the fitness value by the factor $|Agg_d - Con_d|$ if the aggregated QoS value is worse than the given constraint. Then, if it is better, they reward the fitness value by the same factor.



```
Function FitnessFunc()
    M=Big number;
    Fit=0;
    For all web services in composition plan do
      If $Agg_d$ better that $Con_d$
            Fit=Fit – |$Agg_d$-$Con_d$|;
      Else
            Fit=Fit + M * |$Agg_d$-$Con_d$|;
      End
    End
    Retunr Fit;
End
```

Figure 2.8: Penalty based fitness function[15]

The main disadvantage of this method is that, they do not include inter service dependencies in their study, and their fitness function return value interval changes with the number of services in the execution plan. They should prefer a fitness design which gives values between the interval [0,1].

2.1.2 Multi Objective Approach

The studies that are mentioned up to here are done in single optimization objective view, by assigning weights to the QoS parameters. Unlike them, Hiroshi, Paskorn and Junichi used multi-objective optimization technique which is used for problems with conflicting multiple objectives to be optimized simultaneously [12]. For example minimizing the cost and maximizing the throughput are conflicting objectives. Since these objectives can not be met at the same time, there are multiple solutions that are better than the others called *Pareto Optimal Solutions*. In multi-objective algorithms, the result consists of these multiple solutions where the user can see the trade-offs. Moreover their study is different from the others since they also consider multiple *Service Level Agreement* (SLA) namely Platinum, Gold and Silver. These layers differ in each other with the QoS constraints that are assigned to, as given in

Figure 2.9.

Service Level	Constraints			
	Throughput (Lower Bound)	Latency (Upper Bound)	Cost (Upper Bound)	Total Cost (Upper Bound)
Platinum	120000	100	-	2000
Gold	6000	130	-	
Silver	2000	-	250	

Figure 2.9: Different SLAs [12]

Their multi-objective algorithm is mainly dependent on the *domination rank* and *density*. Domination rank is the factor that shows how the individual is better than the others in the population, and the density is that there are how many individuals that resembles to that individual. They define the fitness function $Fitness(p_k)$ as the domination rank ($DRanking(p_k)$) divided by density ($Density(p_k)$). With this design, if the individual dominates more individual then it gets higher fitness value, however, if its density value gets higher, its fitness value decreases.

```
$g = 0$
$P^0$ = Randomly generated $\mu$ individuals, $Q^0$ = $\varnothing$
repeat until $g = g_{max}$ {
  repeat until $|Q^g|$ = $\mu$ {
    $p_1$ = RWSelection($P^g$), $p_2$ = RWSelection($P^g$)
    $q$ = Crossover($p_1$, $p_2$)
    $q$ = Mutation($q$)
    $Q^g$ = $Q^g \cup q$ if $q \notin Q^g$
  }
  $P^{g+1}$ = Top $\mu$ of Sort($P^g \cup Q^g$)
  g=g+1
}

Crossover($p_1$, $p_2$){
  for $i = 1, \ldots, n$ {
    $center_i = (p_1[i] + p_2[i])/2$
    $q[i] = center_i + \frac{(Fitness(p_1) - Fitness(p_2))|p_1[i] - p_2[i]|/2}{F(p_1) + F(p_2)}$
  }
  return $q$
}
```

Figure 2.10: Multi-objective genetic algorithm [12]

You can see the overall algorithm in Figure 2.10, where g_{max} is number of generations, μ is the population size, p[i] is the i^{th} individual. They define the domination as follows:

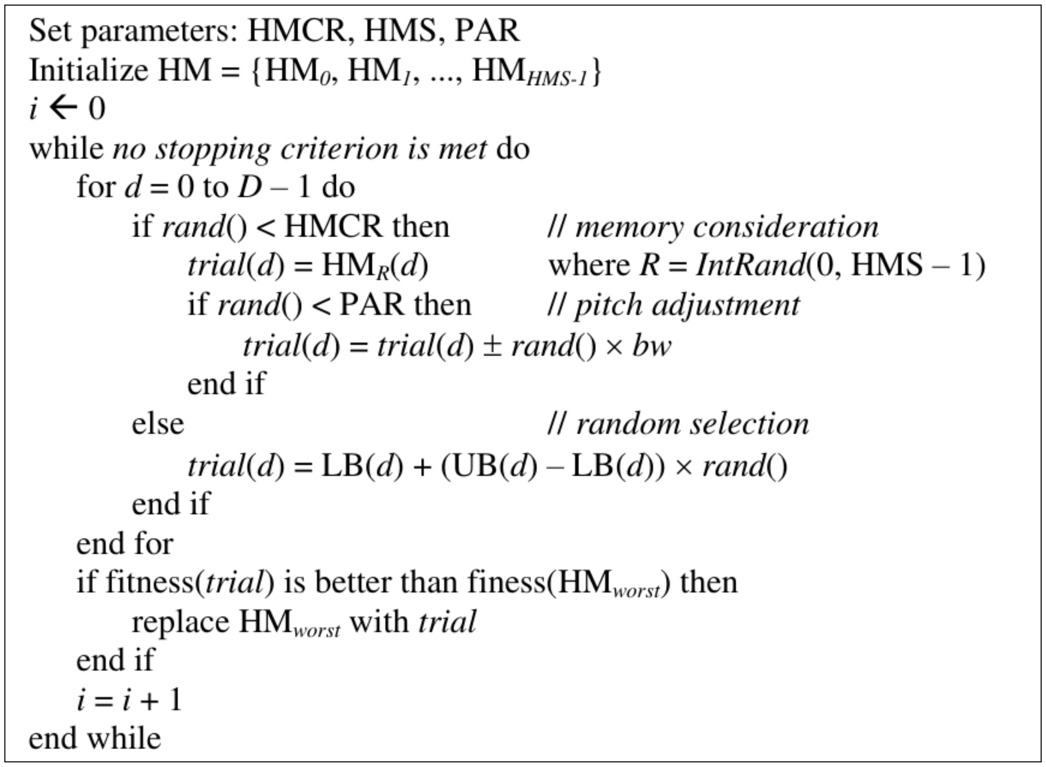
if $o^i = \left(\overrightarrow{o^i_{max}}, \overrightarrow{o^i_{min}}\right)$ is an individual with the set of objectives o^i_{max} and o^i_{min} which are to be maximized and minimized respectively then the individual i dominates j, if both of the following conditions are true:

- $\{\forall \left(o^{i,k}_{max}, o^{j,k}_{max}\right) | o^{i,k}_{max} \geq o^{j,k}_{max}\}$ *and* $\{\forall \left(o^{i,k}_{min}, o^{j,k}_{min}\right) | o^{i,k}_{min} \leq o^{j,k}_{min}\}$
- $\{\exists \left(o^{i,k}_{max}, o^{j,k}_{max}\right) | o^{i,k}_{max} > o^{j,k}_{max}\}$ *or* $\{\exists \left(o^{i,k}_{min}, o^{j,k}_{min}\right) | o^{i,k}_{min} < o^{j,k}_{min}\}$

One disadvantage of their approach is they use elite population technique where they always produce the next generation with the best individuals of the current generation and offspring. This approach is not good I think since the individuals with good fitness values always remain in the population, which may result in local optima. They should use elitism in smaller extent.

2.2 Harmony Search

Harmony Search is a relatively new heuristic algorithm that is developed by Geem et al. [27]. The algorithm mimics the music improvisation process, searching for a perfect state of harmony. This search in music is analogous to the optimization problems. This resemblance make it perfect for use in various optimization problems, particularly in engineering optimization area [20]



```
Set parameters: HMCR, HMS, PAR
Initialize HM = {$HM_0$, $HM_1$, ..., $HM_{HMS-1}$}
$i \leftarrow 0$
while no stopping criterion is met do
    for $d = 0$ to $D - 1$ do
        if rand() < HMCR then              // memory consideration
            trial(d) = $HM_R(d)$           where $R$ = IntRand(0, HMS – 1)
            if rand() < PAR then           // pitch adjustment
                trial(d) = trial(d) ± rand() × bw
            end if
        else                               // random selection
            trial(d) = LB(d) + (UB(d) – LB(d)) × rand()
        end if
    end for
    if fitness(trial) is better than finess($HM_{worst}$) then
        replace $HM_{worst}$ with trial
    end if
    $i = i + 1$
end while
```

Figure 2.11: Harmony Search Algorithm [20]

In Figure 2.11, you can see the original harmony search algorithm. Here HM is the harmony memory with size HMS. This memory stores the candidate solutions. It is analogous to the *population* in GA. This memory is initialized randomly according to the formula 2.7.

$$\begin{aligned} HM_i(d) &= LB(d) + (UB(d) - LB(d)) * rand() \\ for \quad i = 0 \quad to \quad HMS - 1 \quad & and \quad d = 0 \quad to \quad D \end{aligned} \tag{2.7}$$

Where $LB(d)$ and $UB(d)$ are the limits in search space of decision variable d and $rabd()$ is a random function returning values between 0 and 1. There are three parameters controlling the improvisation process:

- Memory consideration rate HMCR
- Pitch adjustment rate PAR
- Pitch adjustment bandwidth *bw*

In each turn, if the *rand()* is smaller than the HMCR, then a candidate is selected randomly form the memory, otherwise new individual is generated from scratch. Moreover, if *rand()* is smaller than PAR then *pitch adjustment* occurs, adjusting the individual. Finally if the individual is better than the worst fitness of the memory, it is replaced with it. This algorithm perfectly mimics the musician improvisation process, since the musician has three choices in improvisation process [21]:

- play famous music from his memory (harmony memory in HS)
- play a music similar to the existing ones (pitch adjustment)
- compose new one (random creation)

Xin-She Yang stated the similarities between HS and GA also [21]. The pitch adjustment (also randomization) is similar to the mutation operator in GA since both contribute to the diversity. Harmony memory is important as the population in GA since both of them make the fittest individuals survive in the later generations. In addition Yang investigated combining the HS algorithm with other heuristics such as *Particle Swarm Optimization*. In our study we propose a combined algorithm with GA and HS with improved capabilities.

One of the important aspect of the HS is selecting the rates carefully. The HMCR and PAR are the leading parameters that affects the convergence and optimization result of the algorithm. If the HMCR rate is too high the algorithm cannot search the space effectively leading to the local optima. If the HCMR rate is chosen too low then the algorithm spends more effort in finding new individuals resulting in slower convergence rate. Similarly low PAR results in slower convergence rate whereas high PAR may cause the search go into local optima [21]. Therefore in the literature there are plenty of studies for HS with dynamic rates one of which are study of Worasucheep [20]. The need for dynamic rates is also taken into consideration in our hybrid algorithm, which will be covered in the next section.

Although HS is used in various domains, the use of it in QoS aware service composition is not studied very well. Jafarpour and Khayyambashi investigated the use of HS in service composition and they compared the results with pure GA [13]. Their algorithm consists of five steps:

1. Initialize the parameters
2. Initialize the memory
3. Improvise new harmony
4. Update the harmony memory
5. Repeat step 3-4 until some criterion

As seen their algorithm discards the *Pitch Adjustment Rate* process. They claim that pitch adjustment with neighbour values cannot generate fitter individual. However we think just the contrary. Since this process resembles the mutation operator in GA, there is a chance that we can have fitter harmony. Moreover getting fitter harmony is not a must. Since at the update state, update can only occur if the resulting harmony is fitter than the worst harmony in the memory. Although their results are slightly better than the pure GA in both average fitness and execution time, they have a risk of getting stuck into local optima. Also, they cannot use the search capability of GA if they only use HS.

2.3 Hybrid Algorithms

Although there are many GA variations in literature investigating different aspects, there are only few studies that use hybrid approach in overall QoS optimization in service composition. We come across only two studies that propose combined algorithms. One of them is the study of Huan Liu and Farong Zhont et al., in which they propose an improved GA based on *Ant Colony Optimization* (ACO) [14].

ACO is relatively new approach for optimization problems that takes inspiration from the social behaviours of some ant species. It was firstly proposed by Marco Dorigo in his doctoral thesis in 1992 [10]. The similarity comes from the ants' behaviour of looking for food. Ants deposit a substance called *pheromone* on the way to the food. Other ants are attracted by these trails which results in finding food resources. Since these pheromone trails evaporate in time, ant colonies tend to find more qualitive food resources which attract more ants. This phenomenon gives the idea that sharing quality information of the path and giving positive feedback can lead to finding optimal solution. This technique is widely used in many problems such as famous traveling salesman problem, NP-Hard problems, applications to telecommunication networks and to industrial problems [14, 10].

Huan Liu and Farong Zhont et al. used ACO in generating fitter individuals in initial population of GA since they claim that simple GA will generate many redundant solutions which result in slow convergence rates. However we think that generating all individuals by ACO in initial population can lead to similar preliminary solutions. This can result in local optima since the GA is forced to operate relatively in smaller search space. Alternatively, they can generate only a small portion of the population by this method. There are many variations of ACO algorithm. The one that they used in their study is *Ant Colony System* which is called ACS [14, 10].

They define the composite service $S = S_1, S_2, \ldots, S_{NA}$ where each component S_i can be replaced with $CS_{i1}, CS_{i2}, \ldots, CS_{im}$. Then they divide all service compositions into $NA+2$ layers. By their definition the candidate service in layer $k-1$ can only be connected to the services in layer k. The pheromone of the connection of the service a of layer k and service b of layer $k+1$ is defined as τ_{ab}^{k}. They use Na as the colony size and $A(n,k)$ as the service which is selected by ant n at layer k.

Firstly they initialize each of the τ_{ab}^{k} with the small value $\tau 0$. Actually this is one of the drawbacks of the ACO algorithm. The initial pheromones cannot be known. They let the all ants select the service 0 at the first layer. If the candidate service selected by ant n at the last layer is $A(n,k-1) = a$, they decide the candidate service selected by this ant by the formula 2.8,

$$A(n,k-1) = \begin{pmatrix} argmax(\tau_{ab}^{k-1}), & \text{if } q < Q_0 \\ S_r, & \text{otherwise} \end{pmatrix} \tag{2.8}$$

where q is a random number between $[0,1]$, Q_0 is pseudo-random selection rate and S_r means pseudo random selection which is presented in formula 2.9,

$$P(a,b) = \frac{\tau_{ab}^{k-1}}{\sum_{x=0}^{m} \tau_{ax}^{k-1}} \tag{2.9}$$

where $m+1$ is the number of candidate service in layer k, $P(a,b)$ is the probability of ant n selecting candidate service a from layer $k-1$ and selecting b in layer k. Moreover in order to decrease the probability that the ants select the same path in one turn, in ACS a special process is conducted. It is called *local pheromone update* which is presented in the formula 2.10,

$$\tau_{A(n,k-1),A(n,k)} \leftarrow (1-\rho) \times \tau_{A(n,k-1),A(n,k)} + \rho \tau_0 \tag{2.10}$$

where ρ is the decrease rate on the path. This update process enables the algorithm operates in larger search space. Until now, the ants do not share the quality information of the path they choose. After each iteration, they calculate the QoS value $f(n)$, the fitness function, and they execute the global pheromone update which can be seen in formula 2.11,

$$\tau^k_{i,j} \leftarrow (1-\alpha) \times \tau^k_{i,j} + \alpha \times const \tag{2.11}$$

where $const = 0.35$, $i = A(n_b), k-1)$, $\quad j = A(n_b, k)$ and n_b is the ant with the best fitness value. By this process, ants can share the path quality information and they can use it in the next iteration.

However there many points that can be criticized in their study:

- their usage of ACO in initial population leads to elite population which can lead to immature search.
- their fitness function is not normalized therefore in very small changes, it can result in huge difference.
- they compared the result of their algorithm with plain ACO, they should have with plain GA. In our experiments we realize that, their approach is not superior to plain GA.
- their approach only applicable to serial aggregation flow, and cannot be used in parallel control flows which is a very important shortcoming.

In another study [22], Xinfeng Ye and Monla suggests combination of Integer Programming, case-based Reasoning and genetic algorithm techniques. They used the scheme that is presented in Figure 2.12.

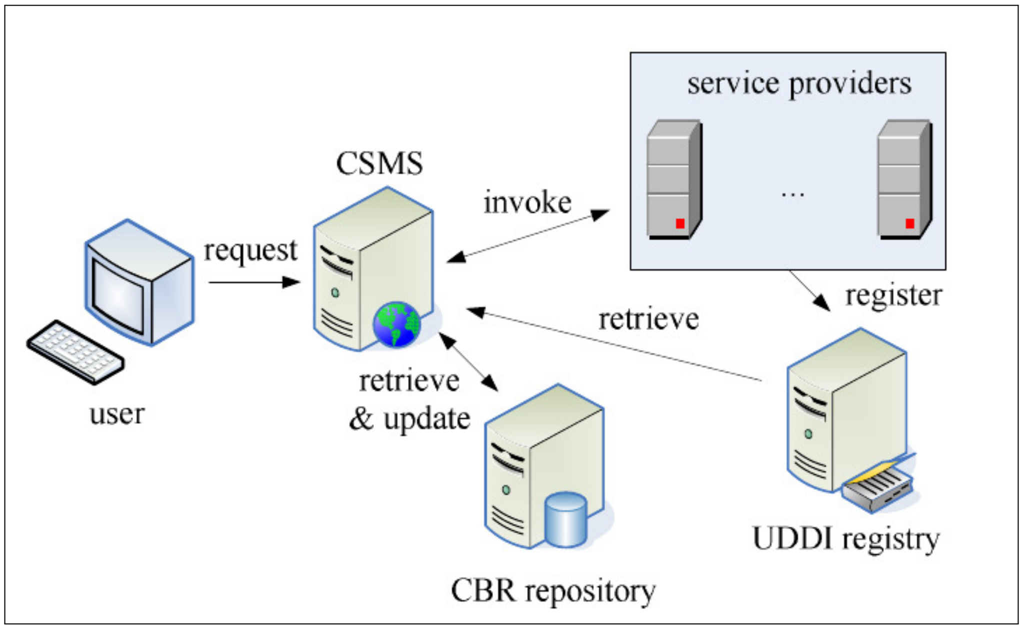


Figure 2.12: Conceptual Diagram of Suggested Scheme[22]

In the proposed scheme, all services are registered in the UDDI registry and they assume that all services uses the same ontology. The case-based reasoning (CBR) repository stores each of the concrete execution plan. The Composite Service Management System (CSMS) generates these execution plans. It uses an IP solver to create new ones or reuse the existing plans in the CBR repository. Also the newly generated execution plans are stored in the CBR repository for future use which constructs the Case-based Reasoning part of their study. In CBR systems, solutions for previous problems are stored and used for new problems which are similar to older ones [6]. In their study they used the similarity function 2.12,

$$\begin{aligned}(digest(plan) = digest(CS)) \wedge (time_{sim}(t_{plan}, t_{cs}) \leq \varepsilon_{time}) \wedge \\ (weight_{sim}(WS_{plan}, WS_{CS}) \leq \varepsilon_{weight})\end{aligned} \tag{2.12}$$

where CS is composite service, WS_{plan} and WS_{CS} denote the sets of weights assigned to QoS properties by an existing plan and composite service respectively. $digest$ is a function that is used for checking the equality of the composite services. It calculates a value describing the composite service's following features:

- the set of tasks
- the composition pattern

Therefore after calculating the digest values from the BPEL file describing the CS, they simply compare these values for equality. They also claim that, the time that the CS is executed also important for CBR similarity calculation and they used the Formula 2.14. The reason behind this usage is that, they claim that QoS properties can vary during the day. In the $time_{sim}(t_{plan}, t_{cs})$, the smaller value means a higher similarity between plan and CS.

$$time_{sim}(t_{plan}, t_{cs}) = \begin{cases} 0 & \text{if } t_{plan} = nil \\ |t_{plan} - t_{cs}| & \text{if } |t_{plan} - t_{cs}| \leq 12 \\ 24mod(|t_{plan} - t_{cs}|) & \text{otherwise} \end{cases} \tag{2.13}$$

Finally they introduce $weight_{sim}(WS_{plan}, WS_{CS})$ which calculates the similarity of the weights assigned to the QoS values that is given in Formula 2.14,

$$\begin{aligned} & weight_{sim}(WS_{plan}, WS_{CS}) = \frac{diff}{\sum_{i \in PS, W_i \in WS_{plan}} (SV_i \star W_i)} \\ & where \\ & diff = |\sum_{i \in PS, W_i \in WS_{plan}} (SV_i \star W_i) - \sum_{i \in PS, W_i \in WS_{CS}} (SV_i \star W_i)| \end{aligned} \tag{2.14}$$

where PS is the QoS properties set, SV_i is the scaled QoS value, WS_{plan} and WS_{CS} are the sets of weights which are assigned to QoS properties of the plan and composite service.

The authors propose that the users assign ranks to the composite services. For example they assign higher ranks to the CS which has higher QoS values. Since they automatically execute the resulting service, after CSMS finds relevant solutions for the user request they applied *roulette-wheel* selection which can be summarized as follows :

1. R is a random number between $(0,1]$
2. $\{1,\ldots,n\}$ is collection of plans returned from CSMS such that $\forall i \in \{1,\ldots,n\} \forall j \in \{1,\ldots,n\}, (i,j) \rightarrow (Score_i \leq Score_j)$
3. P_k is the selection probability calculated from the formula 2.15
4. return the plan $k \in \{1,\ldots,n\}$ such that $\sum_{i=1}^{k-1} P_i < R < \sum_{i=1}^{k} P_i$

$$P_i = \frac{W_i \star Score_i}{\sum_{j=1}^{n} (W_j \star Score_j)} \tag{2.15}$$

where W_i is the assigned weight to the plan i, n is the number of plans matched and $Score_i$ is the QoS score of plan i.

Their study is different from the other studies in several aspects, and some points are not very clear. They do not tell about how the execution plans are generated. All other studies mentioned up to this study, assume that the execution plan is given prior to the QoS optimization process. Moreover, they state that they use CBR for finding similar solutions for the current problem, however they do not mention about the CBR repository size. We think that the repository size must be very large in order to compare. Also scalability is the one of the main issue. Since the problem nature is NP-hard, the usage of heuristics are very common in literature. However their study does not consider this fact.

CHAPTER 3

PROPOSED METHODS

In this chapter, we describe our proposed algorithms that aim to optimize overall QoS value of the execution plan more effectively than the methods that are described earlier. Our approach further improves the original algorithm, resulting quick convergence with higher average fitness values. The proposed improved algorithms use the heuristics *simulated annealing* and *harmony search* as smart mutation operator. These heuristics are altered such that, in the earlier generations, the algorithms aggressively search the solution space, accepting the individuals with lower fitness values. However, in the later generations the acceptance of the solutions with lower fitness values decreases. Instead, the existing solutions are further improved with the heuristics. Since we compare our results with the original GA and the ACO improved GA, these algorithms are also implemented.

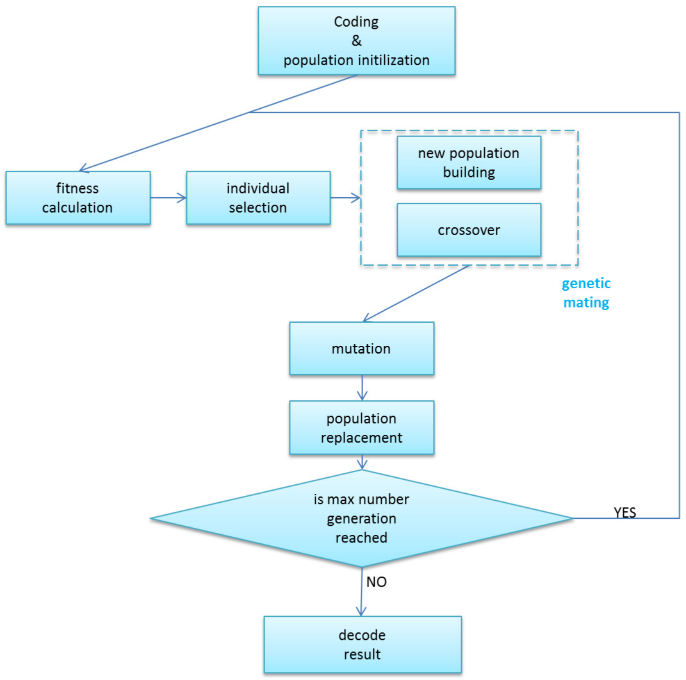


Figure 3.1: Pure Genetic Algorithm

The original genetic algorithm implementation can be seen in Figure 3.1. The algorithm

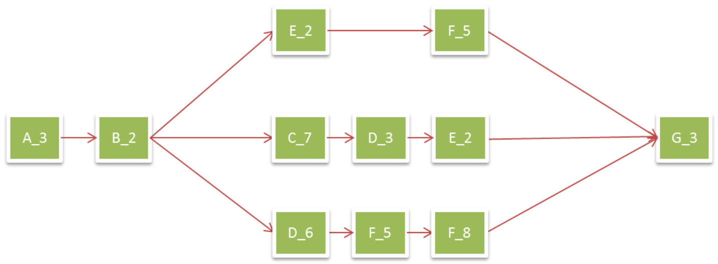


Figure 3.2: Example Chromosome Representation

starts with *coding and population initialization* phase. In this phase the data structures are built. An example of execution plan can be seen in Figure 3.2. In this plan, three serial sub-plans are connected in parallel form. Then this parallel sub-plan is reconnected with two serial sub-plans in a serial fashion. Since our algorithm gets the execution plan as input, the initial population is generated randomly according to the execution plan. We do not apply any heuristic in this phase, since the initial diversity is very important to search the solution space effectively. Then the main loop of the algorithm starts. We first calculate the fitness of the each individual of the population with the fitness function, which is presented in formula 3.1. We use normalized and weighted fitness function design. We normalize the QoS values in [0,1] range. Since our fitness function gives higher values to fitter solutions, we define $normalize_+()$ function to be applied to the QoS parameters that we want to maximize which is "throughput", on the other hand we define $normalize_-()$ to be applied to the QoS parameters that we want to minimize which are "cost" and "responseTime".

$$fitness = \frac{\begin{array}{c} W_1 * norm_-(cost) + W_2 * norm_-(responseTime) \\ + W_3 * reliability + W_4 * norm_+(throughput) \end{array}}{W_1 + W_2 + W_3 + W_4} \tag{3.1}$$

$$normalize_-(q) = \begin{cases} 0 & \text{if } (q_{max} - q_{min}) = 0 \\ 1 - \frac{q - q_{min}}{q_{max} - q_{min}} & \text{otherwise} \end{cases} \tag{3.2}$$

$$normalize_+(q) = \begin{cases} 0 & \text{if } (q_{max} - q_{min}) = 0 \\ \frac{q - q_{min}}{q_{max} - q_{min}} & \text{otherwise} \end{cases} \tag{3.3}$$

After *fitness calculation* process, we apply *individual* selection process in order to determine the individuals for *genetic mating*, which includes *new population building* and *crossover* processes. In the selection phase, we firstly calculate rank values according to Algorithm 2. Then we apply selection method which can be seen in Algorithm 3 in order to find the parents.

Algorithm 2 Fitness Ranking

```
function FITNESSRANKING($individual\_index$)
    $ranking = 0$
    for $i = 1 \to SIZE(population)$ do
        if fitness(individual_index) > fitness($i$) then
            $ranking = ranking + 1$
        end if
    end for
return $ranking$
end function
```

Algorithm 3 Ranking Based Parent Selection

```
$rsc = 20$, $parentFound = false$;
while !parentFound do
    index = getRandom(population size);
    if $index < rsc$ then
        $indexParent = index$
        $parentFound = true$
    else
        if $fitnessRanking(index) + 1 > getRandom(populationsize)$ then
            $indexParent = index$
            $parentFound = true$
        end if
    end if
end while
return $indexParent$
```

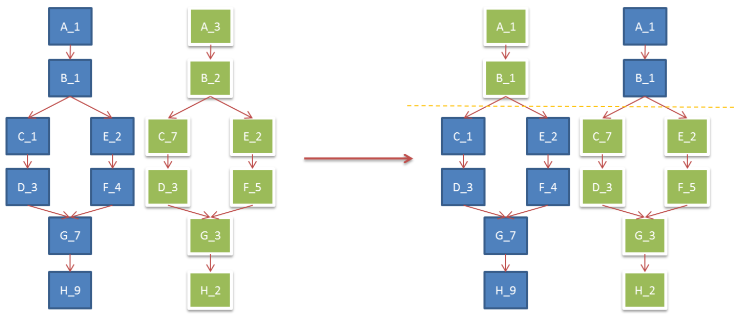


Figure 3.3: One Point Crossover

After selecting the parents, *crossover* operator is applied in order to generate new offsprings. This operator is implemented in two ways as *one point crossover* and *two point crossover*, which are illustrated in Figures 3.3 and 3.4. In one point crossover, after two parents are selected a random crossover point is selected. Then two offsprings are created such that their genes are swapped before this crossover point as seen in Figure 3.3. Similarly, in two point crossover operation, two points are randomly selected and after new offsprings are created swap operation occurs in each points. These crossover operations are the main part of the genetic algorithm that enables the inheritance of the gene parts that is responsible for better fitness values.

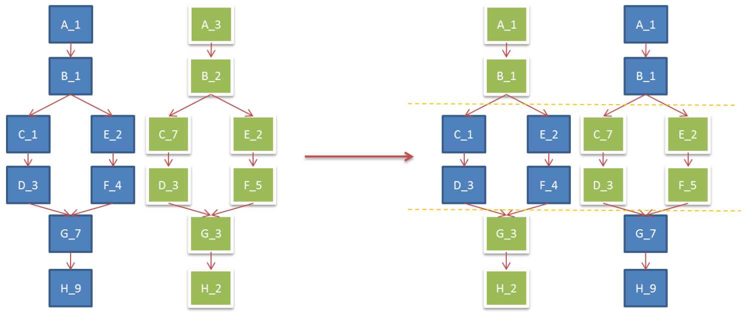


Figure 3.4: Two Point Crossover

After the genetic mating is finished and new population with the offsprings are created, the *mutation* operator is applied to each individual with certain probability. The mutation operator is responsible for the maintaining the diversity of the population. As seen in Figure 3.5, two concrete services (colored as blue) are changed as the result of mutation operation.

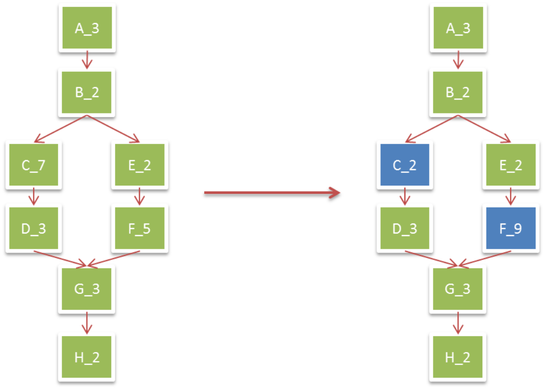


Figure 3.5: Mutation

When mutation operator is applied to the some of the individuals, new population is replaced with the current one. This process is the final operation for creating the new generation. This generation is expected to have higher average fitness value than the previous one, since genetic mating is applied intensively between the fitter individuals. Moreover, we always apply *elitism* while replacing the old generation. We keep the best two solutions of the previous generation unchanged in the new generation in order not to lose the best solutions. Then we check whether the max generation count is reached. If it is not reached yet, we simply return to the fitness calculation and repeat process to here. If we reach the max generation count, we decode the best result to the human readable form and output it. This phase ends the algorithm. The important part here is the determining the max generation count. Conducting some experiments, we see that after some generation count, the average fitness value remains unchanged. Therefore further improvement is not necessary.

3.1 Genetic Algorithm with Simulated Annealing

The suggested algorithm presented in this section is the fusion of GA and SA. Simulated annealing is used here in place of the mutation operator. Mutation process in the original genetic algorithm operates as fully randomized fashion. The mutation operator does not care whether the resulting individual after the mutation occurs, has higher fitness value or not. Simply it changes the genome randomly in order to improve the diversity of the population. The critical point here is that, always generating fitter individuals in mutation operation will most likely prevent the core algorithm from producing the best solution. In other words, the global optima may be reached from the genes of individuals who have lower fitness values.

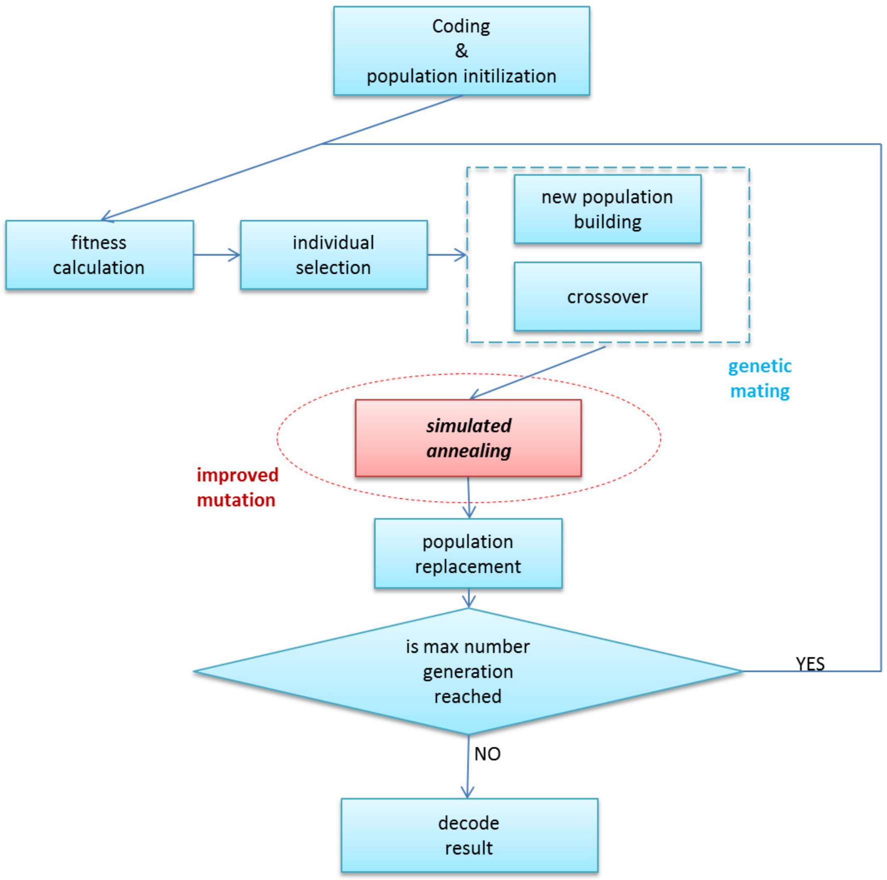


Figure 3.6: Genetic Algorithm with Simulated Annealing

However we see that mutation operator, especially if used in high probability, may decrease the the average fitness values. In order to prevent this, we aim to control the mutation operator such that, in earlier generations it changes the population randomly without considering the overall fitness value of the population. In contrast, in later generations we require the mutation operator to function such that it lets the mutations only improve the individual since we are close to end of algorithm. We observe that, the closer to the end of algorithm the lower average fitness value we get if let the mutation operator decrease the individuals fitness.

In order to control the mutation operation, we replace it with Algorithm 4 which is applied right after the genetic mating process. The resulting algorithm can be seen in Figure 3.6.

Algorithm 4 Simulated Annealing Core

```
1: saPercantage
2: iterationCount
3: initialPopulation
4: P = getImprovementPopulation(saPercantage,initialPopulation)
5: for $i = 0 \rightarrow iterationCount$ do
6:     for all individual p in P do
7:         $m_p = mutate(p)$
8:         if $fitness(m_p) > fitness(p)$ then
9:             accept $m_p$
10:        else
11:            $worseningProb = e^{\frac{fitness(m_p)-fitness(p)}{C*(maxGen/currentGen)}}$
12:            if $getRandom() > worseningProb$ then
13:                accept $m_p$
14:            else
15:                accept $p$
16:            end if
17:        end if
18:    end for
19: end for
20:
```

In simulated annealing improvement part, we do not apply the process to the whole population, instead we deal with the some part of the population. The reason behind this is that, in original mutation, change is applied to the individual under certain probability. Therefore we keep the same logic here. We firstly, randomly determine the population part that will be improved by the SA (the *getImprovementPopulation* method in Algorithm 4). Then, to all individuals in the selected portion of the population, we apply mutation which is repeated *iterationCount* times. If the fitness gain is positive, we immediately accept the mutated individual. Otherwise we accept the mutated individual with the *worseningProb*. *worseningProb* gets higher while the *generationCount* gets close to the maximum generation count. The $C * (currentGen/maxGen)$ on line 11 is the *temperature* function, which controls the probability of the acceptence rate of mutated solutions with low fitness. By this way, the acceptance rate of the mutated individuals with low fitness value gets decreased towards the end of algorithm, which fulfills our main goal.

3.2 Genetic Algorithm with Harmony Search (with dynamic rates)

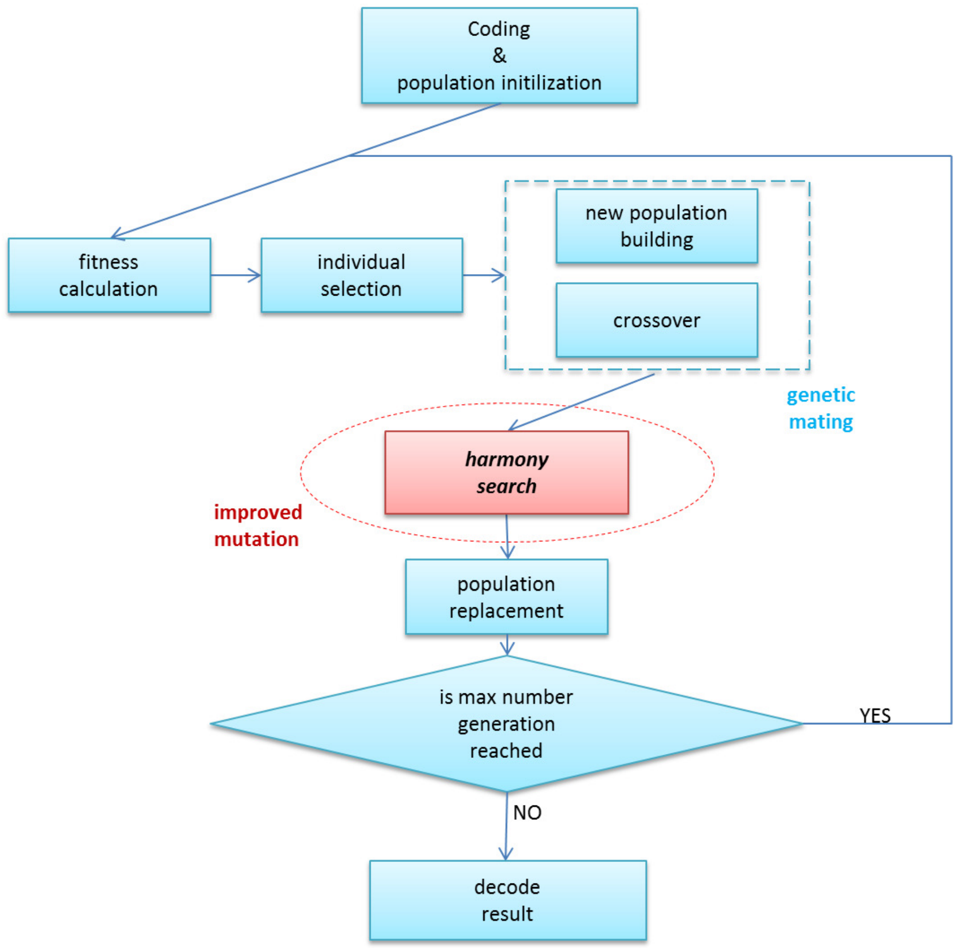


Figure 3.7: Genetic Algorithm with Harmony Search

Similar to the SA improvement, harmony search is replaced with the mutation operator in original genetic algorithm as seen in Figure 3.7. Harmony search, in the earlier phases of the algorithm lets the mutated individuals to be accepted even if they have lower fitness values. On the other hand, in the later phases, it operates in order to improve the current solutions. This results in improved convergency rate compared to pure GA implementation which is our main concern.

In HS improvement phase, which can be seen in Algorithm 5, we firstly compute the *harmony memory consideration rate* $(HMCR)$. The important point in here is that, we change the $HMCR$ dynamically. This dynamic rate results in aggressive search in the earlier phases and improvement in the later phases. With the probability $HMCR$, we explore the population instead of generating new solution. This part is *memory consideration* part of the algorithm. In the memory consideration part we randomly select a solution and with the PAR probability we apply *pitchAdjustment*. This process simply is the mutation process in the original GA. With $(1-HMCR)$ probability we introduce new candidate solution. This phase is the *random* *creation* part of the original HS algorithm. Then, in contrast to original HS, we introduce controlled acceptence mechanism. In original HS, only the individuals with higher fitness values are accepted. However, since we need to accept the individuals with lower fitness values, we inject the same control here. With the probability rate seen in Formula 3.4, we include the bad solutions. Since this rate reaches to higher values towards the end of the algorithm, the acceptence rate of bad solutions will decrease.

$$e^{\frac{fitness(m_p)-fitness(p)}{C*(maxGen/currentGen)}} \tag{3.4}$$

Algorithm 5 Harmony Search Core

```
$HMCR \leftarrow (currentGeneration/maxGeneration)$
$iterationCount$
$P \leftarrow initialPopulation$
for $i = 0 \rightarrow iterationCount$ do
    if $getRandom() < HMCR$ then
        $p \leftarrow getRandomIndividual(P)$
        if $getRandom() < PAR$ then
            $p_m \leftarrow pitchAdjustment(p)$
        end if
    else
        $p_m \leftarrow generateNewSolution()$
    end if
    $worseningProb = e^{\frac{fitness(m_p)-fitness(p)}{C*(maxGen/currentGen)}}$
    if $fitness(m_p) > fitness(p)$ then
        accept $m_p$
    else
        if $getRandom() > worseningProb$ then
            accept $m_p$
        end if
    end if
end for
```

3.3 Application Implementation

The improved algorithms mentioned in Section 3.1 and Section 3.2 are implemented in Java programming language. Since our implementations are compared with pure genetic algorithm and with hybrid approach that uses ACO and GA in the literature, we also implemented these algorithms.

OSGI framework provides the developers with dynamic component system for JAVA pro-

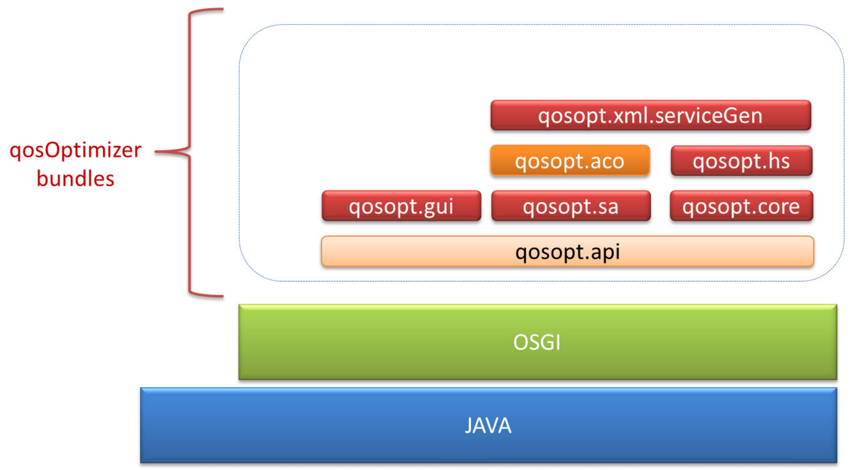


Figure 3.8: QoSOptimizer Design

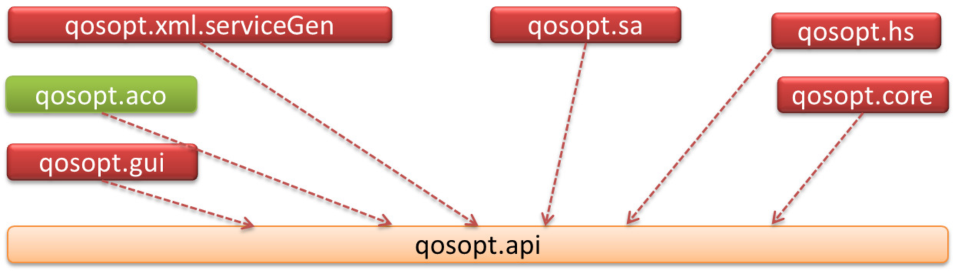


Figure 3.9: QoSOptimizer Dependency Graph

gramming language. It enables the development models where applications are dynamically composed of many reusable components [2]. In addition, the framework enables the modules to communicate with other services which enables loose coupling. We have developed seven modules that lay on top of the OSGI framework. These bundles all together forms our application qosOptimizer which can be seen in Figure 3.8.

All modules have their own responsibilities, hiding their internal details from each other. They provide services that can be used by other modules. The "*qosopt.api*" module is the application programming interface bundle. All service and data definitions that the implementation modules need reside here. The other modules depends only to this module, which enable us changing the implementation details without changing the code in the dependee modules.

The "*qosopt.xml.serviceGen*" module is responsible for providing the data that the algorithm runs with. It converts the XML based service execution plan and XML based concrete services to the internal data structures. Since this module also implement the services responsible for data supply, it is very easy to write and integrate some other data supply module.

The "*qosopt.core*" module is the main module that implements our proposed modified genetic algorithm. This bundle can be configured at run time, in order to enable the improvements such as ACO, SA and HS. It provides "*IQoSOptimizer*" service for execution of algorithms with provided parameters.

The “*qosopt.hs*” module includes the harmony search implementation. It provides “*IHarmonySearchImprovement*” service, which is called from “*qosopt.core*” module if HS improvement is enabled.

The “*qosopt.sa*” module includes the simulated annealing implementation. It provides “*ISimulatedAnnealingImprovement*” service which is used from “*qosopt.core*” module if SA improvement is enabled.

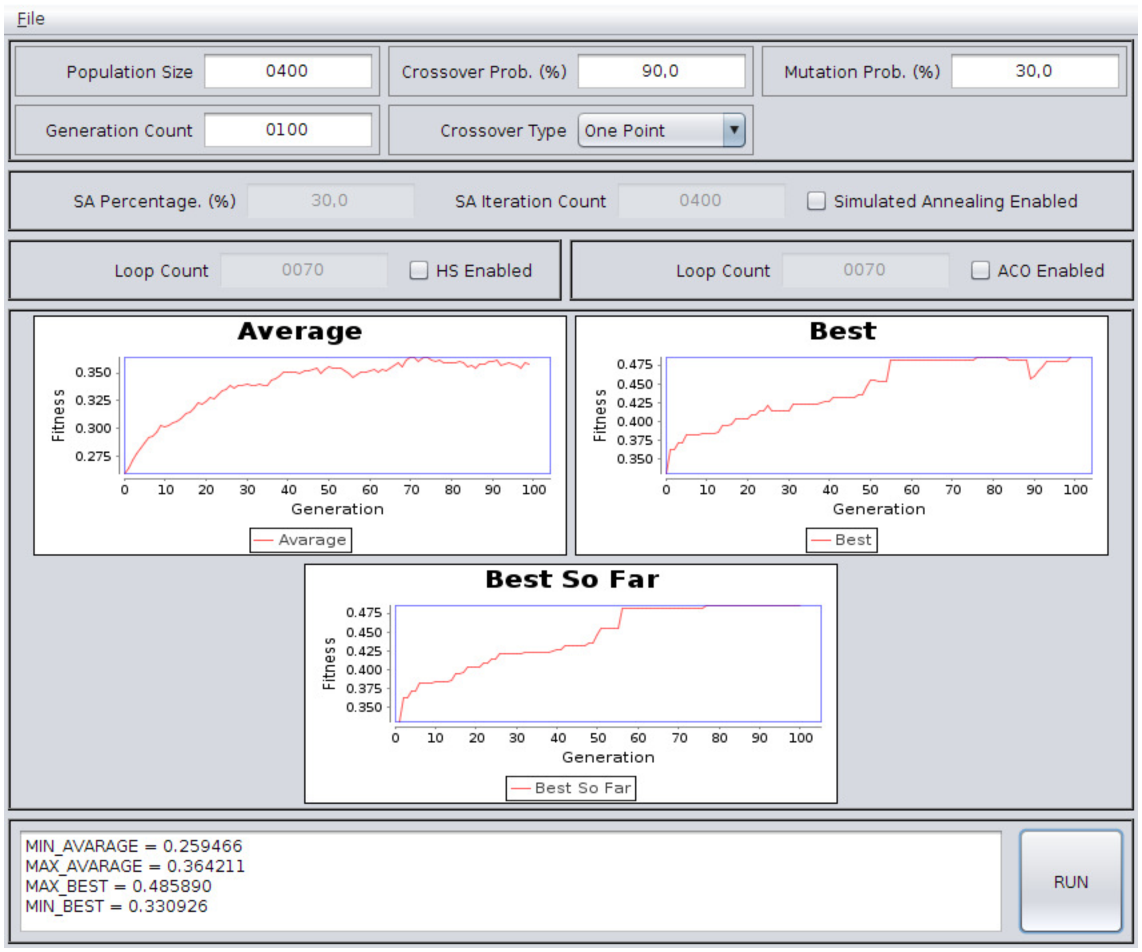
The “*qosopt.aco*” module implements the ACO enabled GA approach which is explained in detail in the review literature section. By this implementation, we compared our algorithms with the only hybrid GA approach that we found. This module also fully isolates its implementation behind its service “*IAntColoniOptimization*”.

The “*qosopt.gui*” module is our test bundle. It uses “*IQoSOptimizer*” service in order to execute the implemented algorithms. It consists of user interfaces in order to get input from the user. Since the only point that connects this client with the core is only service definition, it is very easy to use our implementation components in some other client applications.

In Figure 3.10 main screen of our application is presented. This is the user interface of the “*qosopt.gui*” module. The interface consists of other panels which is used for setting implemented algorithm parameters and charts that visualize the result of the algorithm.

At the top most of the main screen GA parameter panel resides. The controls are used for setting the parameters that genetic algorithm needs. In the example, 400 is selected for population size, 100 is chosen for maximum generation count, 90% is chosen for crossover probability, 30% for mutation probability and one point crossover type. Then SA improvement panel comes. Since the improvement checkbox is not selected, the parameter input controls are disabled. As shown in the screen, 30% is chosen for SA percentage. SA improvement is applied to that portion of the population. Moreover 400 is selected for the iteration count. The remaining two parameter panels are HS and ACO improvement panels. Similarly, they are also disabled since the related combo-boxes are unchecked.

At the center of the main screen, there exists three charts that visualize the fitness values of the population. The first chart shows the average fitness value of the population. The second chart illustrates the best fitness value that is found in each generation. At the final chart, we show the best fitness value that is found up to that generation. The results in this example screens belong to the pure GA implementation since no improvement is selected.



Population Size 0400
Crossover Prob. (%) 90,0
Mutation Prob. (%) 30,0
Generation Count 0100
Crossover Type One Point
SA Percentage. (%) 30,0
SA Iteration Count 0400
Simulated Annealing Enabled
Loop Count 0070
HS Enabled
Loop Count 0070
ACO Enabled

```
MIN_AVARAGE = 0.259466
MAX_AVARAGE = 0.364211
MAX_BEST = 0.485890
MIN_BEST = 0.330926
```

Figure 3.10: QoSOptimizer Main Screen

CHAPTER 4

CASE STUDIES AND EVALUATION

In this chapter, the experimental results of the proposed algorithms are presented. We also compared the results with the methods in the literature. The QoS aggregation rules that are implemented are presented in Table 4.1. We have only implemented serial and parallel aggregations since more complex aggregations can be defined in terms of them. The QoS values that are used in our study are the most used ones in the literature. In this table, the formulas for each of the QoS attributes are used to compute the overall QoS value of the composite service from its constituent services.

Table 4.1: QoS Aggregation Formulas

QoS Attribute	Serial	Parallel
Cost (C)	$\sum_{i=1}^{n} C(s_i)$	$\sum_{i=1}^{n} C(s_i)$
Response Time (RT)	$\sum_{i=1}^{n} RT(s_i)$	$Max\{RT(s_i)_{i\in\{1...n\}}\}$
Reliability (R)	$\prod_{i=1}^{n} R(s_i)$	$\prod_{i=1}^{n} R(s_i)$
Throughput (T)	$Min\{T(s_i)_{i\in\{1...n\}}\}$	$Min\{T(s_i)_{i\in\{1...n\}}\}$

In the experiments, as the data sets, we use the XML based service execution plans and randomly generated QoS values. A sample execution plan and related service implementation XML is given in Appendix B. The execution plan in listing B.1 shows a composite plan including both serial and composition pattern, using 16 abstract services. The service implementations with the QoS values are in the listing B.2. Here “c” stands for *Cost*, “r” stands for *Reliability*, “t” stands for *throughput* and “rt” stands for *responseTime*. The abstract services and the concrete ones are matched with *id* attribute. For example, the *abstractService* with $id =$ ”3” has 60 concrete services defined.

4.1 Experimental Results

In this section, we have compared our algorithms GA_HS (HS improved GA) and HA_SA (SA improved GA) against GA (pure genetic algorithm) and GA_ACO (ACO enabled genetic algorithm). Since we also implemented the ACO enabled and pure GA, we used the same data sets in all algorithms.

Table 4.2: Genetic Algorithm Parameters

GA Parameter Name	Value
Population Count	200
Crossover Probability	90%
Mutation Probability	30%
Maximum Generation Count	100
Crossover Type	One Point
SA Application	30%
SA Iteration Count	400
HS Iteration Count	70
ACO Iteration Count	70

Since GA_ACO proposed in the literature is only applicable in serial execution plan, we conducted our experiments on both serial pattern and composed patterns. In all experiments we used the same algorithm parameters. These GA parameters are given in Table 4.2.

4.1.1 Average Fitness vs. Generation

In this experiment we analyze the change of average fitness value as the generation count increases. The result of this experiment shows the main improvement of this thesis.

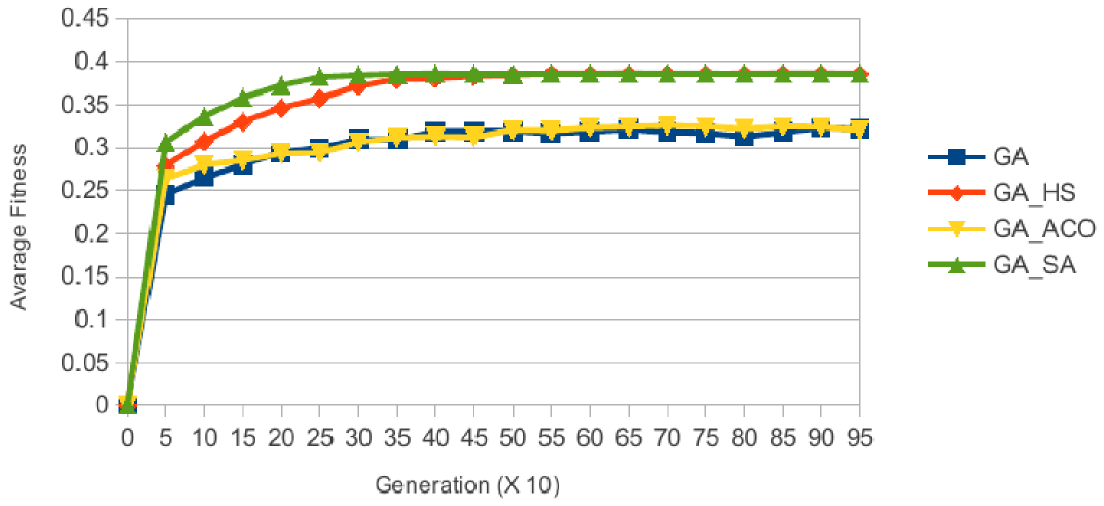


Figure 4.1: Average Fitness vs. Generation Under Serial Workflow Model

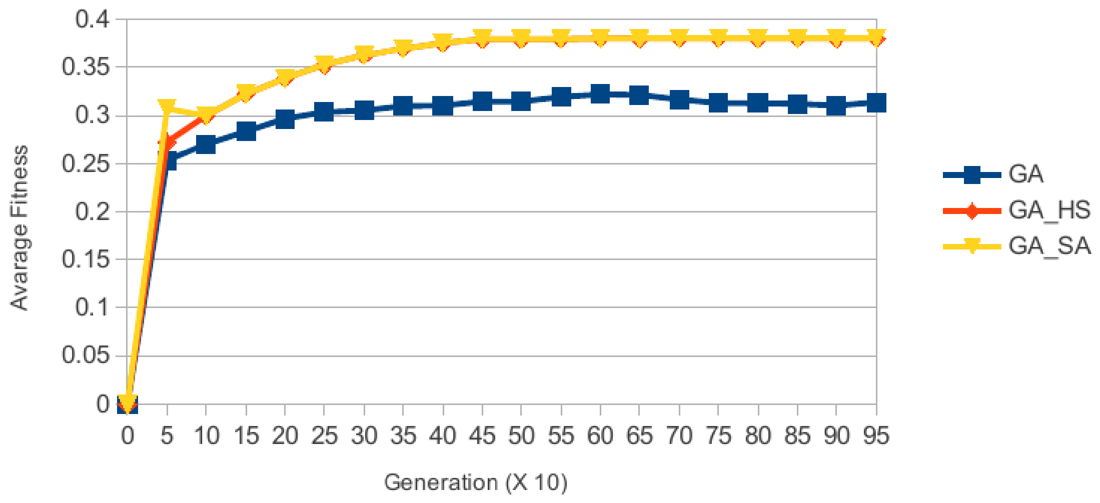


Figure 4.2: Average Fitness vs. Generation Under Serial and Parallel Workflow Model

As seen in Figure 4.1 and Figure 4.2 both of our algorithms are superior to the pure GA and ACO enabled GA algorithm. SA and HS improvements enable the population to reach the average fitness values around to 0.4, while the other ones are around to 0.3. Moreover, our proposed algorithms reach the convergence point earlier than the other algorithms. The results of HS improvement and SA improvement are almost the same except that SA improvement convergence rate is better than the HS improvement. Since the ACO implementation in the literature cannot be applied to workflow that both includes serial and parallel patterns, we have conducted two experiments in this test. In one of them we use the workflow that only includes serial pattern, in one of them we used the workflow that includes both serial and parallel patterns.

In addition, with the execution plan that consists of serial and parallel paterns, HS and SA improvements also get higher average fitness values as generation count increases.

4.1.2 Execution Time vs. Number of Tasks

In this experiment, we evaluate the change in the execution time of the algorithms as the number of tasks increases. We perform the evaluation under 10, 20, 30 and 40 number of tasks. In each experiment, the execution plans have the same structure. As the only difference, we increased the number of abstract services in each serial node. Moreover we define 10 concrete services for each abstract service in order to test only the effect of abstract service count on execution time. As seen in Figure 4.3 and in Figure 4.4 HS improvement's execution time is very close to pure genetic algorithm implementation which makes it elegant choice over pure genetic algorithm.

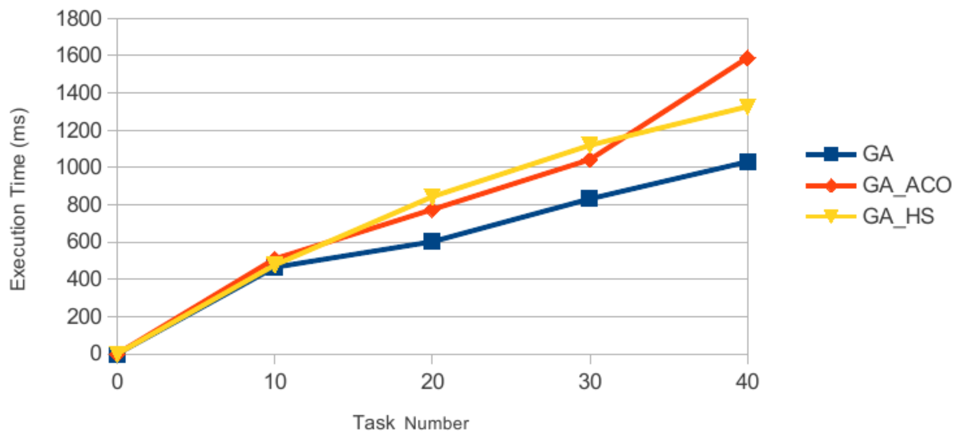


Figure 4.3: Execution Time vs. Number of Tasks Under Serial Workflow Model

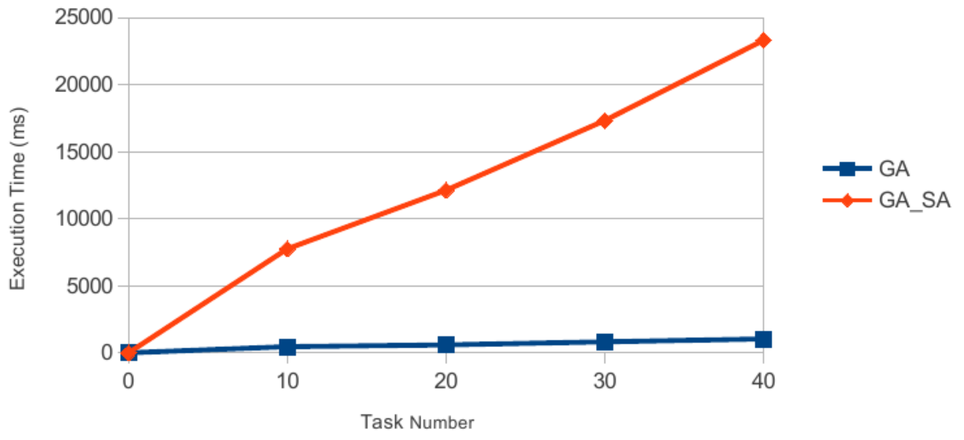


Figure 4.4: Execution Time vs. Number of Tasks Under Serial Workflow Model

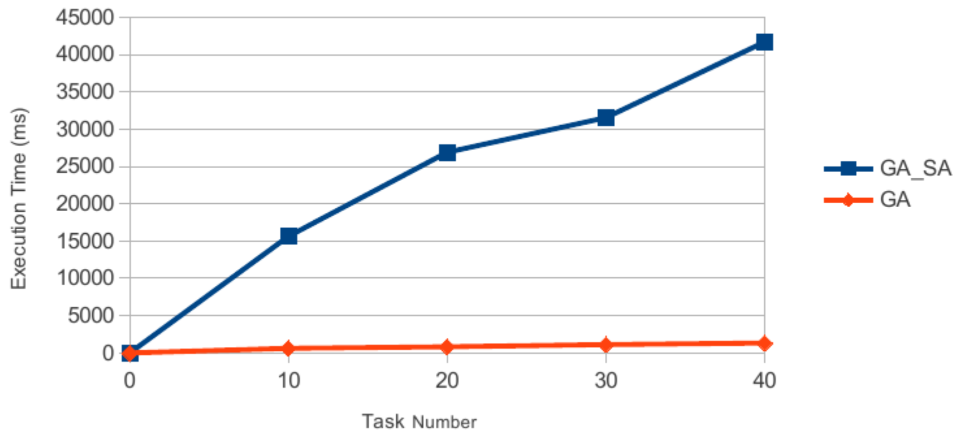


Figure 4.5: Execution Time vs. Task Number Under Serial and Parallel Workflow Model

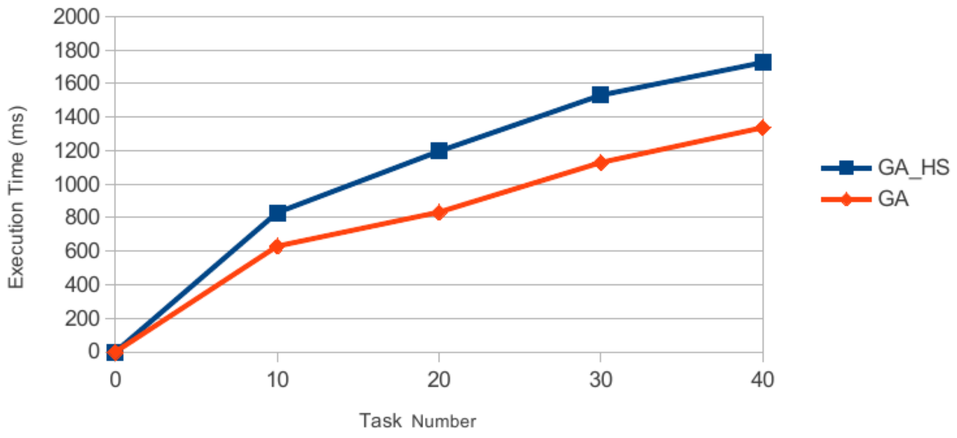


Figure 4.6: Execution Time vs. Task Number Under Serial and Parallel Workflow Model

However, simulated annealing improvement takes much longer time than that of other algorithms. This large gap between execution times make HS superior to SA improvement since HS and SA improvements provide almost the same results on the other test cases.

Similarly, in the serial and parallel execution time analysis, HS results are very close to pure GA implementation and SA improvement takes the highest execution time as seen in Figure 4.5 and Figure 4.6.

4.1.3 Execution Time vs. Generation Count

We also analyze the change of execution time as the generation count increases. In this test, we use the same execution plan and the same concrete service list for each test case. We used 15 abstract services and 10 concrete services for each task.

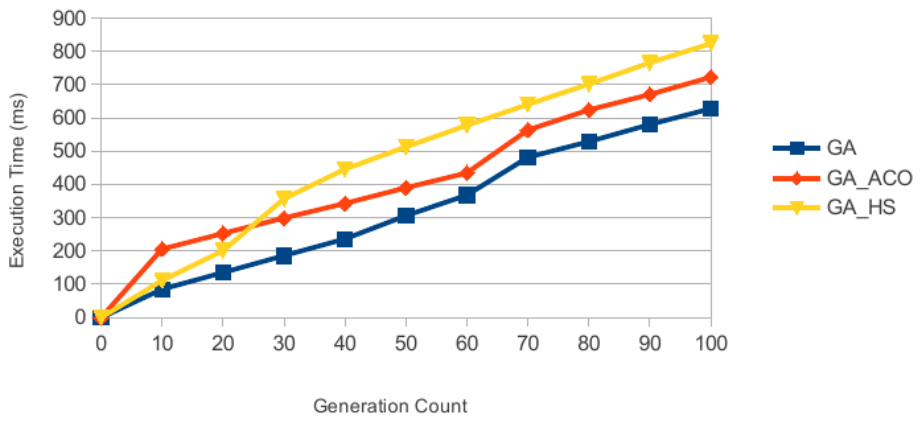


Figure 4.7: Execution Time vs. Generation Count Under Serial Workflow Model

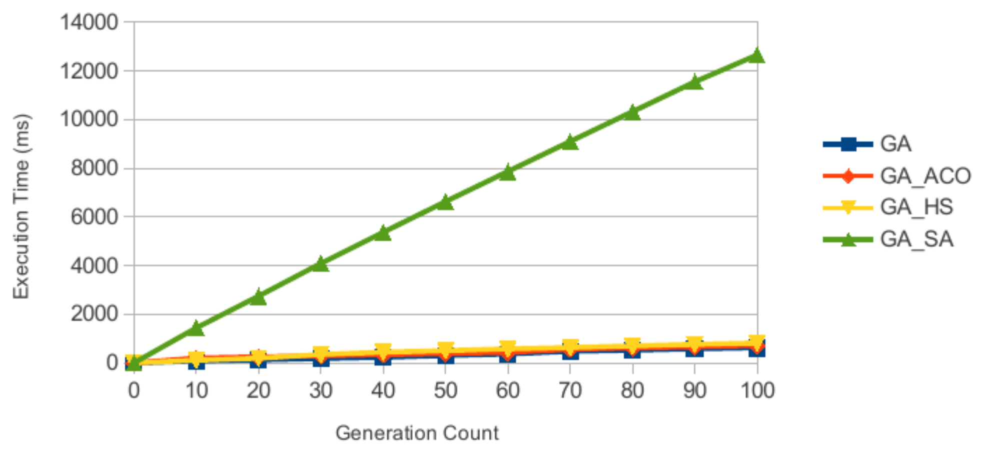


Figure 4.8: Execution Time vs. Generation Count Under Serial Workflow Model

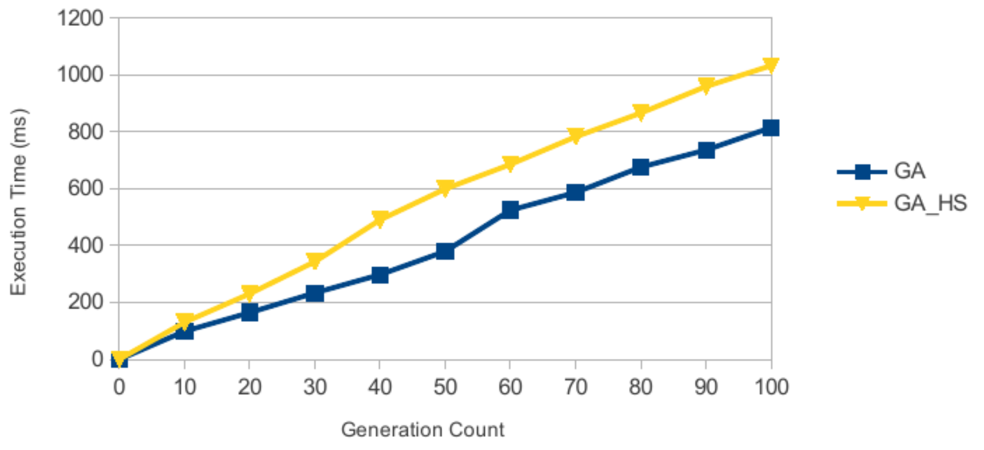


Figure 4.9: Execution Time vs. Generation Count Under Serial and Parallel Workflow Model

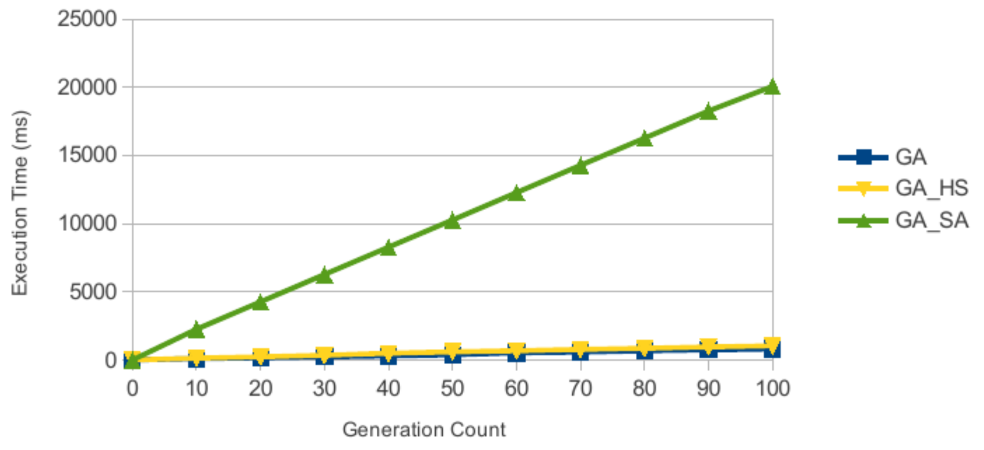


Figure 4.10: Execution Time vs. Generation Count Under Serial and Parallel Workflow Model

As seen in Figure 4.7, Figure 4.8 ,Figure 4.9 and Figure 4.10 the execution time and generation count are linearly dependent, as the generation number increases the execution times increase linearly. Note that HS improvement execution time is very close to pure GA whereas SA improvement's time is significantly higher than the other algorithms. The possible causes of fluctuation in the charts are garbage collection process in JAVA and the non-real time test environment.

4.1.4 Execution Time vs. Number of Concrete Services

In this experiment, we measure the execution time change while the concrete service number changes. We measured the execution time for 10, 20, 30 and 40 concrete services for each abstract service. Since we want to see only the service implementation number effect, we use the same execution plan in each of four cases.

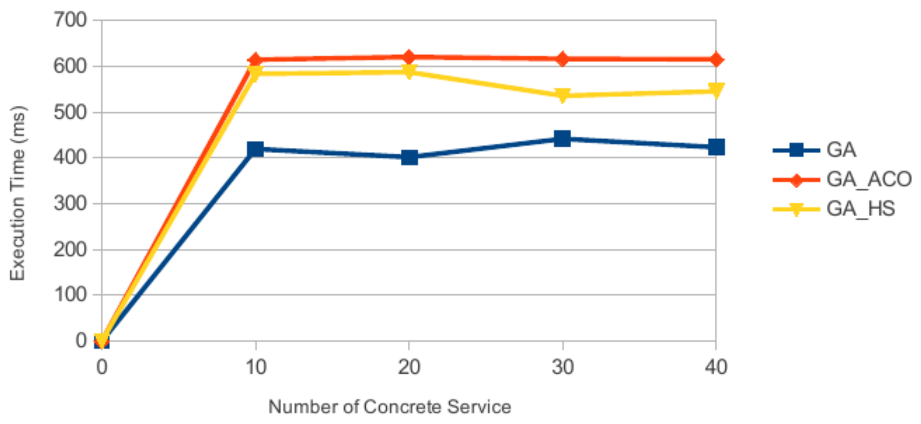


Figure 4.11: Execution Time vs. Number of Concrete Services Under Serial Workflow Model

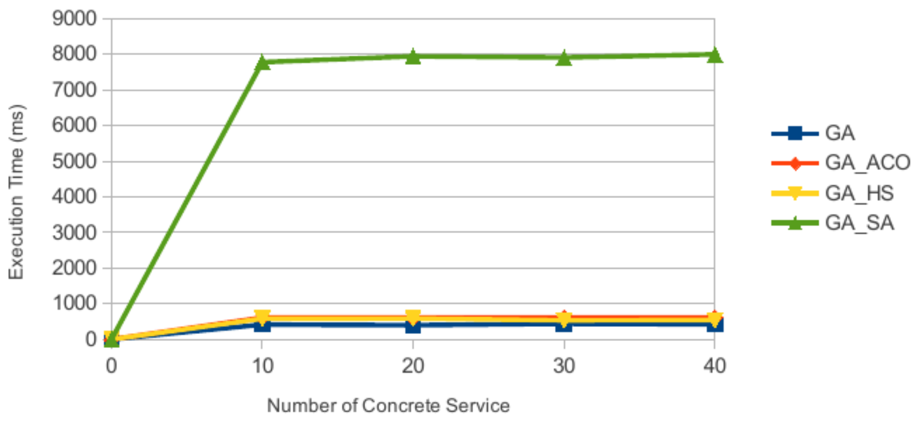


Figure 4.12: Execution Time vs. Number of Concrete Services Under Serial Workflow Model

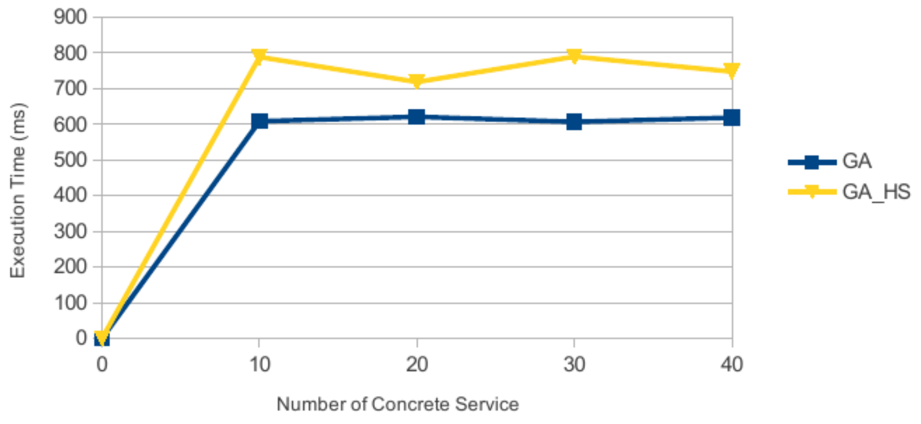


Figure 4.13: Execution Time vs. Number of Concrete Services Under Serial and Parallel Workflow Model

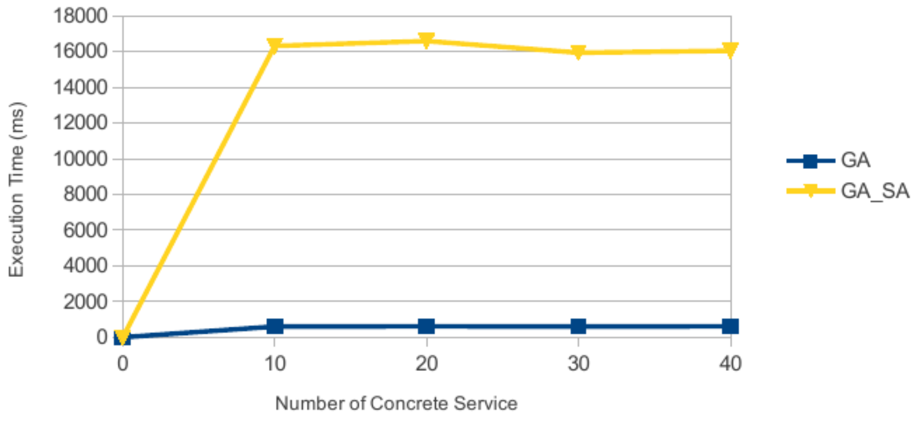


Figure 4.14: Execution Time vs. Number of Concrete Services Under Serial and Parallel Workflow Model

As seen Figures 4.11, 4.12, 4.13 and 4.14 the number of concrete services for each abstract service has no clear effect on the execution time.

Similar to the previous cases, SA improvement has higher execution time than the other algorithms, although this value is constant while the number of concrete services changes.

4.1.5 Best Fitness Value Test

In genetic algorithm, similar to the real evolution in the nature, the fitness of the whole population is more important than the fitness of single individual. In nature, this is the survival key of the species. Therefore, in GA implementations the success is measured from the average fitness values. On the other hand in our domain, overall QoS optimization of web service composition, giving the best result is more important from the last user point of view. However, in order to get the best result in genetic algorithm, the population must have many fit individuals because best individual can only be generated from other fit individuals.

In this experiment, we measured the best fitness values produced by the algorithms. For this evaluation, we present the chart result of our test program, qosOptimizer, in order to visualize the results. We used 10 abstract services in execution plan, and 40 concrete services for each abstract service. We use the same test input for all algorithms.

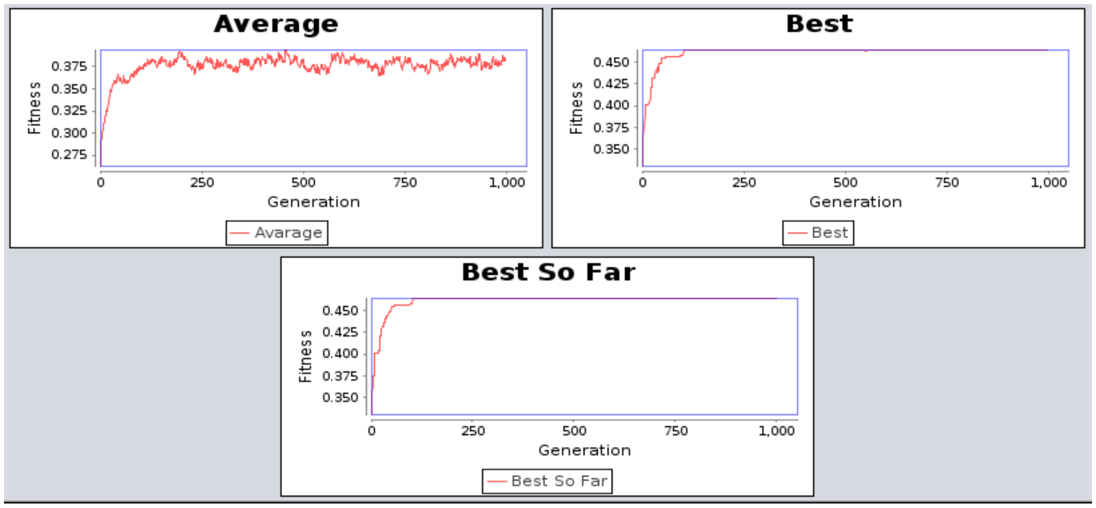


Figure 4.15: Genetic Algorithm Best Fitness

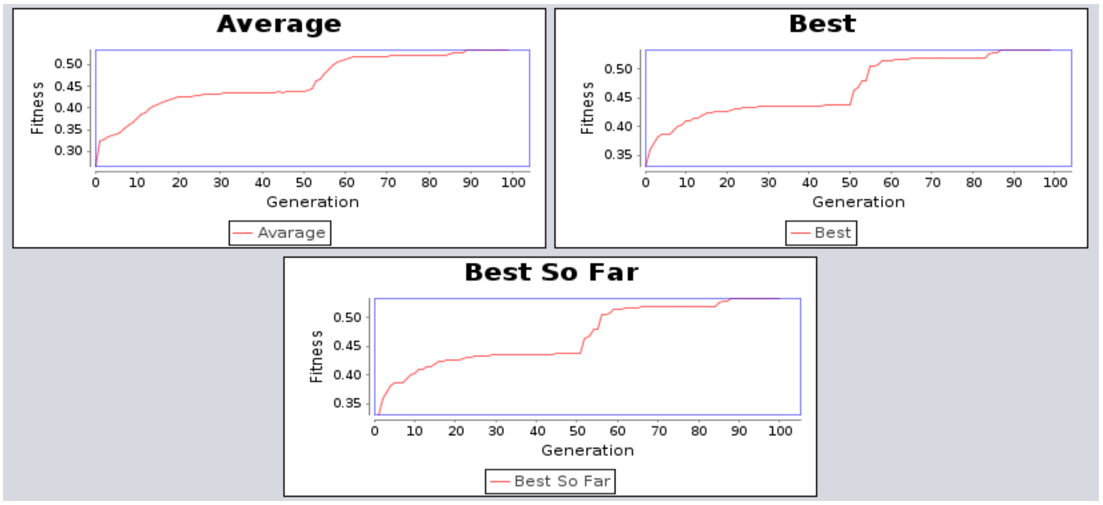


Figure 4.16: HS Improvement Best Fitness

In Figures 4.15, 4.16 and 4.17 there are three charts on the average fitness, best fitness and best fitness so far. In average fitness charts, we show the average fitness values per generation. In best fitness, we visualize the fitness value of the fittest individual per generation. In the best so far chart, we show the fitness value of the fittest individual up to that generation. Actually, the final value of the this chart is the result that we search for.

In Figure 4.15 it is seen that, the average fitness value is reached to 0.375 and the best fitness value is 0.450 when the pure GA is executed.

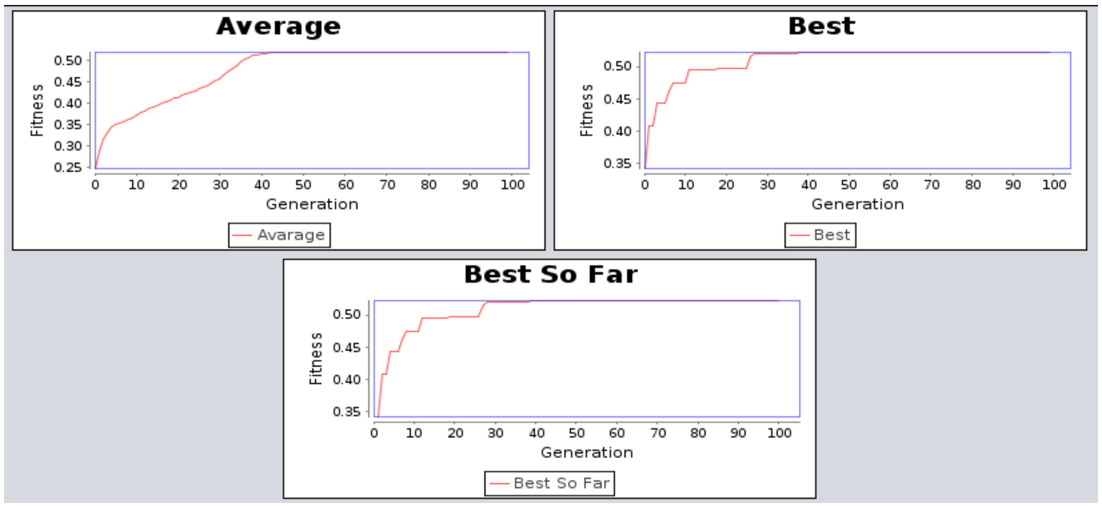


Figure 4.17: SA Improvement Best Fitness

On the other hand in Figures 4.16 and 4.17 it is shown that, the average fitness value is reached to 0.50 and the best fitness value is 0.50.

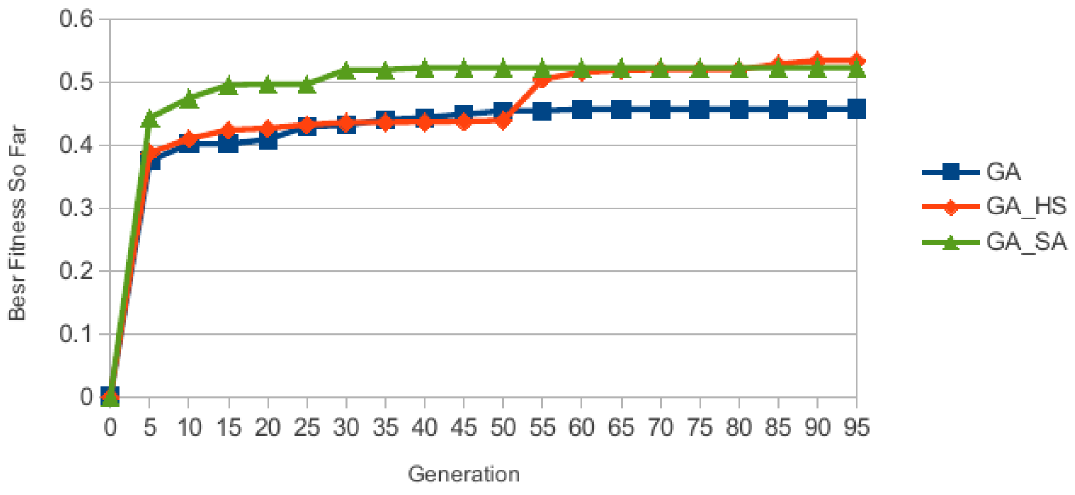


Figure 4.18: Best Fitness So Far

You can see the best fitness values for the first 100 generation in Figure 4.18. As seen HS improvement and SA improvement reach higher values than that of pure GA.

Table 4.3: Best Fitness Test

	GA	GA_HS	GA_SA
Maximum Average Fitness	0.39387	**0.53419**	0.51962
Maximum Best Fitness	0.46419	**0.53419**	0.52283
Maximum Best Fitness So Far	0.46419	**0.53419**	0.52283
Execution Time	5270	**3333**	38665

In this experiment, we increase the maximum generation number in pure GA, which determines the end of algorithm, in order to provide that genetic algorithm runs longer than the HS improvement. Although it runs 2 seconds more than HS improvement the result is far behind the HS improvement, which is given in Table 4.3. While the best fitness value of HS improvement is **0.53419**, best fitness value of pure GA **0.39387**. In this test, however, we increase the loop count of HS improvement to 2000 which makes the result more distinguishable. Although the best fitness value of SA improvement is also good, the run time is 38665 ms which is 8 times longer than that of the HS improvement.

4.1.6 Effect of Mutation Ratio

Since mutation operator is replaced with our smart mutation operators, we also measured the effect of change in mutation ratio on the average fitness value in pure GA. As seen in Figure 4.19, initially, as the mutation ratio increases, the average fitness increases. However, after we increase mutation ratio to 20%, the average fitness tends to decrease. This behavior can be interpreted as the hard to predict and destructive nature of mutation operator.

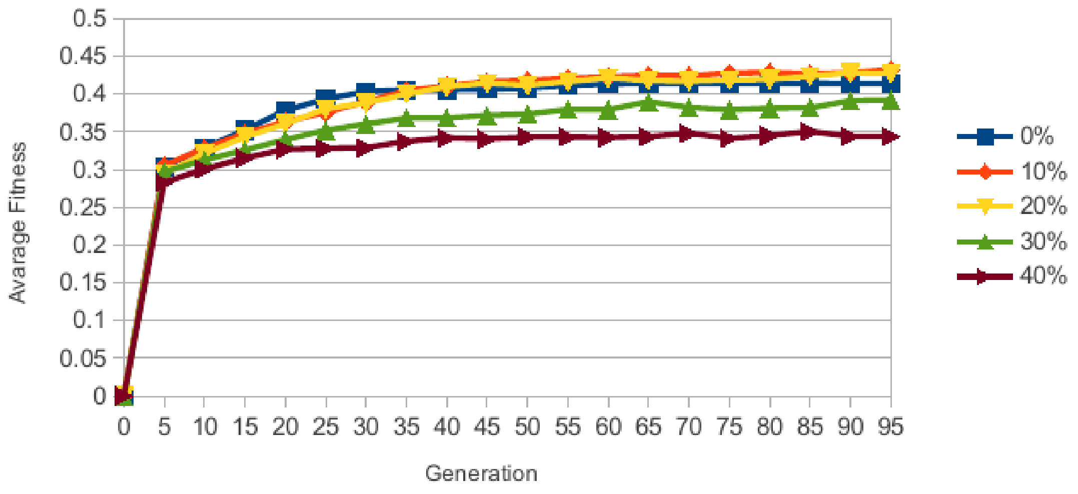


Figure 4.19: Average Fitness vs Generation Count Under the Change of Mutation Ratio in Pure GA

On the other hand, HS improvement is not destructive. As seen in Figure 4.20, HS improvement's effect gets higher as the iteration count increases. At higher generation counts, the net effect of iteration count becomes clear. In HS algorithm, at higher iterations more *pitch adjustment* (the mutation operator in HS domain) is applied. Since we control the effect of mutation on the population dynamically, it supports the population to higher fitness values especially in later generations which is described in detail in Proposed Methods section.

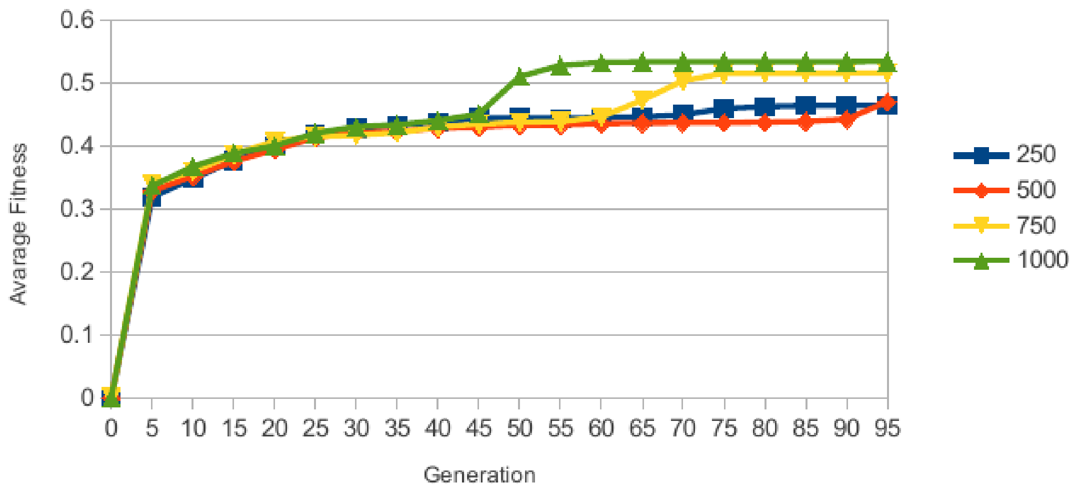


Figure 4.20: Iteration Count Harmony Search

Although average fitness of pure GA changes as the mutation ratio increases, HS improvement gets higher results than those of pure GA in all tested mutation ratios. You can see that, in Tables 4.4 and 4.5, HS improvement results in higher maximum average fitness values than those of pure GA. This final test shows that HS improvement is superior to the pure GA with various mutation ratios.

Table 4.4: Pure Genetic Algorithm

Mutation Ratio	Max. Average Fitness
%0	0.414448
%10	0.433151
%20	0.429535
%30	0.394951
%40	0.352531

Table 4.5: Harmony Search Improvement

Iteration Count	Max. Average Fitness
250	0.464190
500	0.485764
750	0.516976
1000	0.534086

CHAPTER 5

CONCLUSION

Using existing web services in order to compose new web services is very popular. With the increasing web technologies and demand on the services, many services become available. Among these services, many of them can be used interchangeably enabling them to be defined as *abstract services* regarding the input, output and the execution logic of them. Moreover, there can be a lot of concrete services for each these abstract services carrying out the same job. Although these services can be defined as equivalent in the sense of the job they realize, they may have different quality of service values that affect overall quality of the composed service.

There are two main issues in this orchestration process of the web services. The first one is the determining the execution plan which consists of the aggregation structure of the services in the composition. In this execution plan, aggregation is done with the abstract services since there can be many candidate services for each node in the composition. The second one is that, selecting concrete services for the each abstract service in the execution plan such that overall QoS of the composed service is maximized. In this thesis we try to propose solutions for this second problem.

In this thesis, there are two improved genetic algorithm proposed in order to optimize the overall QoS value of the given execution plan. Since the problem is NP-hard, we see that optimization techniques are very common in the literature, which is one of the reasons that we choose also using genetic algorithm. However, the methods in the literature use pure genetic algorithm with pure mutation operator. Instead of pure mutation operator, we propose other heuristics with modifications as *smart mutation operator*. Original mutation operator does not care the mutated individual has higher fitness or not. Moreover, mutation tends to generate individuals with lower fitness values, which makes the population has lower average fitness values if it is applied aggressively. However, if mutation operates such that it produces always better individuals then there appears the risk of getting stuck in the local optima. Therefore, we propose smart mutation operators such that, in the earlier generations it does not care the mutated individuals fitness value. In later generations, however, it only accepts the mutations only if they have higher fitness values.

The first proposed algorithm is *simulated annealing improved genetic algorithm* which uses

SA as mutation operator. As seen in Chapter 4, this algorithm makes the population to have higher fitness values than the original GA and ACO improved GA which is the only hybrid genetic algorithm for this problem in the literature domain. Although SA improvement's execution time is longer than the pure GA, pure GA cannot reach the average fitness value that the SA improvement reaches.

The second proposed algorithm is *harmony search improved genetic algorithm* which uses HS as smart mutation operator. Similar to SA enabled GA, this algorithm is also superior to pure GA and ACO enabled GA. Moreover, HS improvement execution time overhead is nearly 100 ms in our most compelling tests. Also, the convergence rate of HS improvement is much better than the pure GA. It reaches to the stable fitness value in earlier generations.

Another advantage of SA and HS improvement over pure GA is that, the fittest individual of our proposed algorithms are much better than the best of pure GA. Moreover, ACO enabled GA, the only hybrid GA in our literature search, can only operate on only serial execution plans. On the other hand, SA and HS improvement are applicable in all aggregations.

In order to test the algorithm implementations and compare them with each other, along with our algorithms, pure GA and ACO enabled one are also implemented. Moreover, we develop a test application ,*QoSOptimizer*, which is used in all test cases. Since we develop the algorithms in OSGI framework, they are bundled as independent libraries which can be easily used in other client applications. This modular design helped us to implement and test the implementations easily, since it separates all implementations form each other and enables them to be used without other implementations.

We use weighted fitness function design which makes the core algorithm single objective optimization. As a future work, multiple objective optimization can be realized within our proposed algorithms. Also, parallel computing techniques can be used in order to run multiple sessions in order to increase the chance to reach global optima which can be defined also as a web service.

REFERENCES

[1] ibm.com, "Understanding WSDL in a UDDI registry". Retrieved at April 2, 2011, from `http://www.ibm.com/developerworks/webservices/library/ws-wsdl/`.

[2] Osgi alliance, "The OSGI Architecture". Retrieved at May 12, 2012, from `http://www.osgi.org/Technology/WhatIsOSGi`.

[3] tutorialspoint.com, "UDDI Data Model". Retrieved at April 2, 2011, from `http://www.tutorialspoint.com/uddi/uddi_data_model.htm`.

[4] uddi.org, "Introduction to UDDI:Important Features and Functional Concepts". Retrieved at April 2, 2011, from `http://uddi.org/pubs/uddi-tech-wp.pdf`.

[5] *Artificial Intelligence A Modern Approach.* Prentice Hall, Pearson Education Inc., 2003.

[6] E. P. A. Aamodt. Case-based reasoning: Foundational issues, methodological variations, and system approaches. In *CAICom - Artificial Intelligence Communications, IOS Press*, pages 39–59, 1994.

[7] L. Ai and M. Tang. A penalty-based genetic algorithm for qos-aware web service composition with inter-service dependencies and conflicts. *CIMCA,IAWTIC,ISE*, pages 738–743, 2008.

[8] E. Askaroglu. Aotomatic quality of service (qos) evaluation for domain specific web service discovery framework. *METU Thesis Journal*, pages 1–51, 2012.

[9] H. C. Chunming Gao, Meiling Cai. Qos-aware service composition based on tree-coded genetic algorithm. *31st Annual International Computer Software and Applications Conference*, 2007.

[10] M. Dorigo, M. Birattari, and T. Stutzle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[11] F. Glover. Tabu search fundamentals and uses. Retrieved at October 8, 2012, from `http://leeds-faculty.colorado.edu/glover/TS%20-%20Fundamentals&Uses.pdf`.

[12] J. S. K. O. Hiroshi Wada, Paskorn Champrasert. Multiobjective optimization of sla-aware service composition. *IEEE Ongress on Services*, pages 368–375, 2008.

[13] N. Jafarpour and M. Khayyambashi. A new approach for qos-aware web service composition based on harmony search algorithm. In *Web Systems Evolution (WSE), 2009 11th IEEE International Symposium on*, pages 75–78, 2009.

[14] H. Liu, F. Zhong, B. Ouyang, and J. Wu. An approach for qos-aware web service composition based on improved genetic algorithm. In *Web Information Systems and Mining (WISM), 2010 International Conference on*, volume 1, pages 123–128, 2010.

[15] H. S. Mahmood Allameh Amiri. Qos aware web service composition based on genetic algorithm. *International Symposium on Telecommunications*, pages 502–507, 2010.

[16] E. Maximilien and M. Singh. A framework and ontology for dynamic web services selection. *Internet Computing, IEEE*, 8(5):84–93, 2004.

[17] T. Rajendran and P. Balasubramanie. An optimal agent-based architecture for dynamic web service discovery with qos. In *Computing Communication and Networking Technologies (ICCCNT), 2010 International Conference on*, pages 1–7, 2010.

[18] A. Strunk. "QoS-Aware Service Composition:A Survey". *Eight IEEE European Conference on Web Services*, pages 67–73, 2010.

[19] S. K. C. D. G. M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671 –680, 1983.

[20] C. Worasucheep. A harmony search with adaptive pitch adjustment for continuous optimization. *International Journal of Hybrid Information Technology*, 4:13 –20, 2011.

[21] X. S. Yang. Harmony search as a metaheuristic algorithm. *Studies in Computational Intelligence*, 181:1 –14, 2009.

[22] X. Ye and R. Mounla. A hybrid approach to qos-aware service composition. In *Web Services, 2008. ICWS '08. IEEE International Conference on*, pages 62–69, 2008.

[23] Y. S. C. H. W. J. Y. K. S. P. M. Yong Yi, Fanjiang. Genetic algorithm for qos-aware dynamic web service composition. *Proceedings of the Ninth International Conference on Machine Learning and Cybernetics,Qingdao*, pages 1–2, 2010.

[24] C. Z. Yue Ma. Quick convergence of genetic algorithm for qos-driver web service selection. *Computer Networks*, 2:1093–1104, 2008.

[25] C. Zhang. Adaptive genetic algorithm for qos-aware service selection. *Workshops of International Conference on Advanced Information Networking and Applications*, pages 273–278, 2011.

[26] C. Zhang. Adaptive genetic algorithm for qos-aware service selection. *Workshops of International Conference on Advanced Information Networking and Applications*, pages 273–278, 2011.

[27] G. L. Z.W. Geem, J.H. Kim. A new heuristic optimization algorithm: Harmony search. *Simulations*, 76:60 –68, 2001.

APPENDIX A

CASE STUDIES

In this section, average fitness values vs. generation number, execution time vs. task number, execution time vs. generation count and execution time vs. concrete service number tables are given. These tables are results of case studies which are given in Chapter 4.

Table A.1: Average Fitness values vs. Generation Number in Serial Workflow Model

Generation Number	GA	GA_ACO	GA_HS	GA_SA
5	0.23734	0.26399	0.27859	0.30613
10	0.33604	0.28001	0.30741	0.33604
15	0.35810	0.28494	0.32976	0.35810
20	0.37268	0.29303	0.34605	0.37268
25	0.29952	0.29368	0.35751	0.38207
30	0.30973	0.30608	0.37174	0.38415
35	0.30930	0.31167	0.37970	0.38539
40	0.31863	0.31304	0.38060	0.38581
45	0.31890	0.31220	0.38318	0.38557
50	0.31920	0.32058	0.38429	0.38550
55	0.31658	0.31973	0.38572	0.38563
60	0.31878	0.32324	0.38575	0.38569
65	0.32113	0.32415	0.38572	0.38581
70	0.31808	0.32554	0.38564	0.38577
75	0.31679	0.32449	0.38570	0.38580
80	0.31274	0.32157	0.38571	0.38563
85	0.31774	0.32458	0.38577	0.38575
90	0.32327	0.32398	0.38579	0.38574
95	0.32128	0.32000	0.38581	0.38564

Table A.2: Average Fitness values vs. Generation Number in Serial and Parallel Workflow Model

Generation Number	GA	GA_HS	GA_SA
5	0.25301	0.27237	0.30727
10	0.27028	0.29989	0.33361
15	0.28337	0.32218	0.35154
20	0.29684	0.33840	0.36415
25	0.30347	0.35260	0.37447
30	0.30518	0.36248	0.37938
35	0.30978	0.36953	0.38085
40	0.31022	0.37558	0.38090
45	0.31463	0.37957	0.38074
50	0.31464	0.37948	0.38084
55	0.31923	0.37968	0.38087
60	0.32200	0.38037	0.38082
65	0.3212	0.38054	0.38084
70	0.31673	0.38066	0.38079
75	0.31296	0.38054	0.38080
80	0.31279	0.38046	0.38078
85	0.31227	0.38048	0.38074
90	0.31029	0.38055	0.38072
95	0.31387	0.38065	0.38080

Table A.3: Execution Time vs. Task Number in Serial Workflow Model

Task Number	GA	GA_ACO	GA_HS	GA_SA
10	465	509	477	7751
20	602	775	844	12128
30	832	1043	1120	17319
40	1031	1588	1327	23321

Table A.4: Execution Time vs. Task Number in Serial and Parallel Workflow Model

Task Number	GA	GA_HS	GA_SA
10	630	830	15634
20	833	1197	26928
30	1129	1531	31572
40	1337	1725	41714

Table A.5: Execution Time vs. Generation Count in Serial Workflow Model

Task Number	GA	GA_ACO	GA_HS	GA_SA
10	85	206	111	1443
20	135	253	201	2743
30	186	299	357	4092
40	236	343	446	5370
50	306	390	513	6635
60	368	435	578	7867
70	482	564	641	9102
80	529	624	702	10325
90	580	671	766	11553
100	628	723	824	12658

Table A.6: Execution Time vs. Generation Count in Serial and Parallel Workflow Model

Task Number	GA	GA_HS	GA_SA
10	98	130	2259
20	164	230	4257
30	233	343	6272
40	297	490	8267
50	379	599	10263
60	524	685	12269
70	586	782	14274
80	675	866	16270
90	735	959	18266
100	814	1032	20069

Table A.7: Execution Time vs. Concrete Service Number in Serial Workflow Model

Task Number	GA	GA_ACO	GA_HS	GA_SA
10	419	614	583	7779
20	401	620	587	7942
30	441	616	535	7906
40	423	615	545	7993

Table A.8: Execution Time vs. Concrete Service Number in Serial and Parallel Workflow Model

Task Number	GA	GA_HS	GA_SA
10	609	789	16329
20	621	719	16604
30	607	790	15958
40	619	748	16054

APPENDIX B

XML BASED SERVICE EXECUTION PLAN & IMPLEMENTATIONS

In this section the XML based service execution plan and corresponding XML based concrete service list are given as an example. These are the ones that are used in the case studies section, the average fitness per generation test.

Listing B.1: Simple Executuion Plan

```
<?xml version="1.0" encoding="UTF 8"?>
<executionPlan>
   <serial>
      <serial>
         <abstractService id="0" />
         <abstractService id="1" />
         <abstractService id="2" />
      </serial>
      <parallel>
         <serial>
            <abstractService id="3" />
            <abstractService id="4" />
            <abstractService id="5" />
            <abstractService id="6" />
            <abstractService id="7" />
         </serial>
         <serial>
            <abstractService id="8" />
            <parallel>
               <serial>
                  <abstractService id="9" />
                  <abstractService id="10" />
               </serial>
               <serial>
                  <abstractService id="11" />
                  <abstractService id="12" />
               </serial>
            </parallel>
            <abstractService id="13" />
         </serial>
      </parallel>
      <serial>
```

```
            <abstractService id="14" />
            <abstractService id="15" />
        </serial>
    </serial>
</executionPlan>
```

Listing B.2: Service Implementation

```
<?xml version="1.0" encoding="UTF 8"?>
<services id="" >
   <serviceSet id="0">
      <services  id="0" cost="1" reliability="0.1" throughput="1"
          responseTime="1"/>
      <services  id="1" cost="12" reliability="0.2" throughput="2"
          responseTime="2"/>
      <services  id="2" cost="13" reliability="0.3" throughput="3"
          responseTime="3"/>
      <services  id="3" cost="12" reliability="0.2" throughput="2"
          responseTime="2"/>
      <services  id="4" cost="4" reliability="0.9" throughput="9"
          responseTime="9"/>
   </serviceSet>
   <serviceSet id="1">
      <services  id="0" cost="8" reliability="0.3" throughput="3"
          responseTime="3"/>
      <services  id="1" cost="11" reliability="0.1" throughput="1"
          responseTime="1"/>
      <services  id="2" cost="1" reliability="0.6" throughput="6"
          responseTime="6"/>
      <services  id="3" cost="12" reliability="0.2" throughput="2"
          responseTime="2"/>
      <services  id="4" cost="3" reliability="0.8" throughput="8"
          responseTime="8"/>
      <services  id="5" cost="10" reliability="0.5" throughput="5"
          responseTime="5"/>
   </serviceSet>
   <serviceSet id="2">
      <services  id="0" cost="7" reliability="0.7" throughput="7"
          responseTime="7"/>
      <services  id="1" cost="9" reliability="0.4" throughput="4"
          responseTime="4"/>
      <services  id="2" cost="11" reliability="0.1" throughput="1"
          responseTime="1"/>
      <services  id="3" cost="11" reliability="0.1" throughput="1"
          responseTime="1"/>
      <services  id="4" cost="3" reliability="0.3" throughput="3"
          responseTime="3"/>
      <services  id="5"  cost="1" reliability="0.1" throughput="1"
          responseTime="1"/>
      <services  id="6" cost="7" reliability="0.7" throughput="7"
          responseTime="7"/>
   </serviceSet>
   <serviceSet id="3">
      <services id="0"  cost="6" reliability="0.6" throughput="6"
          responseTime="6"/>
      <services id="1"  cost="6" reliability="0.1" throughput="1"
          responseTime="1"/>
      <services id="2"  cost="12" reliability="0.7" throughput="7"
          responseTime="7"/>
```

```
        <services id="3"  cost="8" reliability="0.3" throughput="3"
            responseTime="3"/>
        <services id="4"  cost="12" reliability="0.7" throughput="7"
            responseTime="7"/>
        <services id="5"  cost="6" reliability="0.1" throughput="1"
            responseTime="1"/>
    </serviceSet>
    <serviceSet id="4">
        <services id="0"  cost="14" reliability="0.9" throughput="9"
            responseTime="9"/>
        <services id="1"  cost="9" reliability="0.4" throughput="4"
            responseTime="4"/>
        <services id="2"  cost="15" reliability="1" throughput="10"
            responseTime="10"/>
        <services id="3"  cost="10" reliability="1" throughput="10"
            responseTime="10"/>
        <services id="4"  cost="11" reliability="0.6" throughput="6"
            responseTime="6"/>
        <services id="5"  cost="2" reliability="0.2" throughput="2"
            responseTime="2"/>
        <services id="6"  cost="7" reliability="0.7" throughput="7"
            responseTime="7"/>
        <services id="7"  cost="11" reliability="0.6" throughput="6"
            responseTime="6"/>
    </serviceSet>
    <serviceSet id="5">
        <services id="0"  cost="3" reliability="0.8" throughput="8"
            responseTime="8"/>
        <services id="1"  cost="14" reliability="0.9" throughput="9"
            responseTime="9"/>
        <services id="2"  cost="5" reliability="0.5" throughput="5"
            responseTime="5"/>
        <services id="3"  cost="15" reliability="0.5" throughput="5"
            responseTime="5"/>
        <services id="4"  cost="12" reliability="0.7" throughput="7"
            responseTime="7"/>
        <services id="5"  cost="6" reliability="0.1" throughput="1"
            responseTime="1"/>
        <services id="6"  cost="13" reliability="0.8" throughput="8"
            responseTime="8"/>
    </serviceSet>
    <serviceSet id="6">
        <services id="0"  cost="4" reliability="0.4" throughput="4"
            responseTime="4"/>
        <services id="1"  cost="1" reliability="0.1" throughput="1"
            responseTime="1"/>
        <services id="2"  cost="8" reliability="0.8" throughput="8"
            responseTime="8"/>
        <services id="3"  cost="13" reliability="0.3" throughput="3"
            responseTime="3"/>
        <services id="4"  cost="14" reliability="0.9" throughput="9"
            responseTime="9"/>
    </serviceSet>
```

```
<serviceSet id="7">
   <services id="0"  cost="8" reliability="0.8" throughput="8"
       responseTime="8"/>
   <services id="1"  cost="14" reliability="0.9" throughput="9"
       responseTime="9"/>
   <services id="2"  cost="5" reliability="0.1" throughput="10"
       responseTime="10"/>
   <services id="3"  cost="1" reliability="0.6" throughput="6"
       responseTime="6"/>
   <services id="4"  cost="13" reliability="0.3" throughput="3"
       responseTime="3"/>
</serviceSet>
<serviceSet id="8">
   <services id="0"  cost="12" reliability="0.2" throughput="2"
       responseTime="2"/>
   <services id="1"  cost="12" reliability="0.7" throughput="7"
       responseTime="7"/>
   <services id="2"  cost="12" reliability="0.2" throughput="2"
       responseTime="2"/>
   <services id="3"  cost="8" reliability="0.8" throughput="8"
       responseTime="8"/>
   <services id="4"  cost="2" reliability="0.7" throughput="7"
       responseTime="7"/>
   <services id="5"  cost="15" reliability="0.1" throughput="10"
       responseTime="10"/>
</serviceSet>
<serviceSet id="9">
   <services id="0"  cost="4" reliability="0.4" throughput="4"
       responseTime="4"/>
   <services id="1"  cost="13" reliability="0.8" throughput="8"
       responseTime="8"/>
   <services id="2"  cost="11" reliability="0.1" throughput="1"
       responseTime="1"/>
   <services id="3"  cost="14" reliability="0.4" throughput="4"
       responseTime="4"/>
   <services id="4"  cost="1" reliability="0.6" throughput="6"
       responseTime="6"/>
   <services id="5"  cost="3" reliability="0.3" throughput="3"
       responseTime="3"/>
   <services id="6"  cost="1" reliability="0.1" throughput="1"
       responseTime="1"/>
</serviceSet>
<serviceSet id="10">
   <services id="0"  cost="7" reliability="0.7" throughput="7"
       responseTime="7"/>
   <services id="1"  cost="7" reliability="0.2" throughput="2"
       responseTime="2"/>
   <services id="2"  cost="3" reliability="0.3" throughput="3"
       responseTime="3"/>
   <services id="3"  cost="14" reliability="0.4" throughput="4"
       responseTime="4"/>
   <services id="4"  cost="8" reliability="0.8" throughput="8"
       responseTime="8"/>
```

```xml
    <services id="5"  cost="13" reliability="0.3" throughput="3"
        responseTime="3"/>
    <services id="6"  cost="4" reliability="0.4" throughput="4"
        responseTime="4"/>
    <services id="7"  cost="3" reliability="0.8" throughput="8"
        responseTime="8"/>
    <services id="8"  cost="12" reliability="0.7" throughput="7"
        responseTime="7"/>
  </serviceSet>
  <serviceSet id="11">
    <services id="0"  cost="1" reliability="0.6" throughput="6"
        responseTime="6"/>
    <services id="1"  cost="2" reliability="0.7" throughput="7"
        responseTime="7"/>
    <services id="2"  cost="11" reliability="0.6" throughput="6"
        responseTime="6"/>
    <services id="3"  cost="1" reliability="0.1" throughput="1"
        responseTime="1"/>
    <services id="4"  cost="12" reliability="0.2" throughput="2"
        responseTime="2"/>
    <services id="5"  cost="9" reliability="0.9" throughput="9"
        responseTime="9"/>
  </serviceSet>
  <serviceSet id="12">
    <services id="0"  cost="12" reliability="0.2" throughput="2"
        responseTime="2"/>
    <services id="1"  cost="10" reliability="0.5" throughput="5"
        responseTime="5"/>
    <services id="2"  cost="15" reliability="0.5" throughput="5"
        responseTime="5"/>
    <services id="3"  cost="11" reliability="0.1" throughput="1"
        responseTime="1"/>
    <services id="4"  cost="5" reliability="0.5" throughput="5"
        responseTime="5"/>
    <services id="5"  cost="12" reliability="0.7" throughput="7"
        responseTime="7"/>
    <services id="6"  cost="5" reliability="1" throughput="10"
        responseTime="10"/>
    <services id="7"  cost="5" reliability="1" throughput="10"
        responseTime="10"/>
  </serviceSet>
  <serviceSet id="13">
    <services id="0"  cost="13" reliability="0.8" throughput="8"
        responseTime="8"/>
    <services id="1"  cost="6" reliability="0.6" throughput="6"
        responseTime="6"/>
    <services id="2"  cost="15" reliability="0.5" throughput="5"
        responseTime="5"/>
    <services id="3"  cost="6" reliability="0.6" throughput="6"
        responseTime="6"/>
    <services id="4"  cost="3" reliability="0.8" throughput="8"
        responseTime="8"/>
  </serviceSet>
  <serviceSet id="14">
    <services id="0"  cost="2" reliability="0.2" throughput="2"
        responseTime="2"/>
    <services id="1"  cost="11" reliability="0.1" throughput="1"
        responseTime="1"/>
    <services id="2"  cost="8" reliability="0.3" throughput="3"
        responseTime="3"/>
    <services id="3"  cost="11" reliability="0.6" throughput="6"
        responseTime="6"/>
    <services id="4"  cost="3" reliability="0.3" throughput="3"
        responseTime="3"/>
  </serviceSet>
  <serviceSet id="15">
    <services id="0"  cost="4" reliability="0.9" throughput="9"
        responseTime="9"/>
    <services id="1"  cost="14" reliability="0.4" throughput="4"
        responseTime="4"/>
    <services id="2"  cost="14" reliability="0.9" throughput="9"
        responseTime="9"/>
    <services id="3"  cost="7" reliability="0.7" throughput="7"
        responseTime="7"/>
    <services id="4"  cost="15" reliability="1" throughput="10"
        responseTime="10"/>
    <services id="5"  cost="11" reliability="0.6" throughput="6"
        responseTime="6"/>
    <services id="6"  cost="5" reliability="0.5" throughput="5"
        responseTime="5"/>
  </serviceSet>
</services>
```