SanGA: A Self-adaptive Network-aware Approach to Service Composition

Adrian Klein, *Member, IEEE,* Fuyuki Ishikawa, *Member, IEEE,* and Shinichi Honiden, *Member, IEEE*

Abstract—Service-Oriented Computing enables the composition of loosely coupled services provided with varying Quality of Service (QoS) levels. Selecting a near-optimal set of services for a composition in terms of QoS is crucial when many functionally equivalent services are available. As the number of distributed services, especially in the cloud, is rising rapidly, the impact of the network on the QoS keeps increasing. Despite this, current approaches do not differentiate between the QoS of services themselves and the network. Therefore, the computed latency differs from the actual latency, resulting in suboptimal QoS. Thus, we propose a network-aware approach that handles the QoS of services and the QoS of the network independently. First, we build a network model in order to estimate the network latency between arbitrary services and potential users. Our selection algorithm then leverages this model to find compositions with a low latency for a given execution policy. We employ a self-adaptive genetic algorithm which balances the optimization of latency and other QoS as needed and improves the convergence speed. In our evaluation, we show that our approach works under realistic network conditions, efficiently computing compositions with much lower latency and otherwise equivalent QoS compared to current approaches.

Index Terms—Web Services, Cloud, Network, QoS, Optimization, Service Composition.

1 Introduction

Service-Oriented Computing (SOC) is a paradigm for designing and developing software in the form of interoperable services. Each service is a software component that encapsulates a well-defined business functionality.

1.1 Service Composition

SOC enables the composition of these services in a loosely coupled way in order to achieve complex functionality by combining basic services. The main benefit of SOC is that it enables rapid and easy composition at low cost [23].

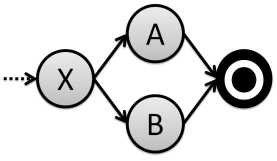


Fig. 1: Workflow

Such compositions result in workflows consisting of abstract tasks, like in Fig. 1, that can either be created manually or computed automatically by planning approaches [28]. For each abstract task, concrete services have to be chosen in order to execute the workflow.

- *A. Klein and S. Honiden are with the Department of Computer Science, The University of Tokyo, Japan. E-mail: adrian@nii.ac.jp.*
- *F. Ishikawa and S. Honiden are with the National Institute of Informatics, Tokyo, Japan. E-mail: {f-ishikawa,honiden}@nii.ac.jp.*

1.2 QoS-aware Service Composition

For service compositions, functional and non-functional requirements [22] have to be considered when choosing such concrete services. The latter are specified by Quality of Service (QoS) attributes (such as latency, price, or availability), and are especially important when many functionally equivalent services are available. The QoS of a composition is the aggregated QoS of the individual services according to its workflow patterns [10] (e.g. parallel patterns), assuming that each service specifies its own QoS in a Service Level Agreement (SLA). The user can then specify his QoS preferences in form of a utility function and QoS constraints for the composition as a whole, e.g. a user might prefer fast services, but only if they are within his available budget. Choosing concrete services that maximize the user's utility while satisfying his constraints has been shown to be a NP-hard optimization problem [24]. Thus, while the problem can be solved optimally with Integer (Linear) Programming (IP) [32], usually heuristic algorithms like genetic algorithms are used to find near-optimal solutions in polynomial time [5].

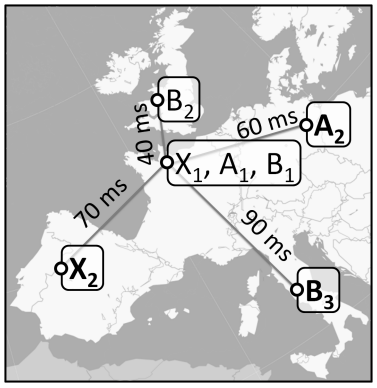
1.3 Service Composition in Distributed Settings

With the advent of Cloud Computing and Software as a Service (SaaS), it is expected that more and more web services will be offered all over the world [4]. This has two main impacts on the requirements of service compositions.

1.3.1 Network-Awareness

First, the impact of the network on the QoS of the overall service composition increases with the degree

of distribution of the services. Despite this, current approaches do not differentiate between the QoS of services themselves and the QoS of the network. The common consensus is that the provider of a service has to include the network latency in the response time he publishes in his SLA. This is not a trivial requirement, since latency varies a lot depending on the user's location [33]; e.g. users in Frankfurt, New York, or Tokyo may have quite different experiences.



(a) Deployed Services

Serv.	Exec. Time
X_1	100 ms
X_2	**90 ms**
A_1	100 ms
A_2	**95 ms**
B_1	100 ms
B_2	90 ms
B_3	**70 ms**

(b) Service SLAs

Fig. 2: Distributed Deployment

For example, consider the abstract workflow depicted in Fig. 1, and the corresponding concrete services (X_1, X_2, A_1, etc.), in which X_1 executes task X, etc., in the times conforming to Fig. 2b. We can see the deployed services and the network delays between the different deployment locations in Fig. 2a. In such a scenario, current approaches would select X_2, A_2 and B_3, because their QoS are optimal, resulting in a total execution time of 185 ms. Now, for user in France the round trip times for these services would add 295 ms to that time. In comparison, executing X_1, A_1 and B_1 would just take 200 ms and only incur a minimal delay because of round trip times. On the other hand, if providers added the maximum delay for any user to the execution time in their respective SLAs, this would guarantee a certain maximum response time to all users, but it would discourage users from selecting local providers and instead favor providers with the most homogeneous delays for all users (e.g. providers in the center of Fig. 1 in France).

1.3.2 Scalability

Secondly, as the number of services increases, the scalability of current approaches becomes crucial. That is, as the number of services grows, more functionally equivalent services become available for each abstract task, and this causes the complexity of the problem to increase exponentially. One can argue that the number of services offered by different providers with the same functionality might not grow indefinitely. Usually a small number of big providers and a fair number of medium-sized providers would be enough to saturate the market. The crucial point is that in traditional service composition, most providers specify just one SLA for the service being offered. Only in some cases up to a handful of SLAs may be specified; for example, a provider might offer platinum, gold and silver SLAs to cater to different categories of users with different QoS levels [27].

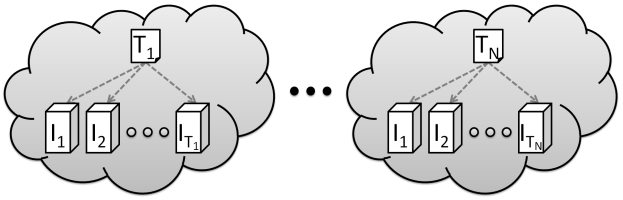


Fig. 3: Deployed Services by Different Providers

In contrast to this, in a network-aware approach each provider has to provide many SLAs, as in Fig. 3. Given an abstract task T a provider that offers a service T_i for T might have to supply different SLAs for each of his instances $I_1, \ldots, I_{T_i}$. Each instance might run on a physical or virtual machine with different characteristics (CPU, memory, etc.). Also, these instances might be executed at completely different locations in order to offer the service in different countries. As we want to differentiate between each instance, we get many more choices for each individual task. For example, whereas previous approaches might assume 50 different providers for each task [31], and, thus, consider 50 choices per task, we might easily have to consider 2500 choices, if we assume that each provider deploys 50 instances of his service on average.

1.4 Contributions

Thus, we propose a new self-adaptive approach, SanGA, for network-aware service composition which significantly improves our previous approach [16]:

1. Network Model. We adopt a generic network model that can be fed in a scalable way by state-of-the-art algorithms from the network research community. We enhance this model by adding scalable facilities to find services which are close to certain network locations or network paths. This allows us to estimate the network latency between arbitrary network locations of services or users and to find services that will result in a low latency for given communication patterns.

2. Network-aware QoS Computation. We specify a realistic QoS model that allows us to compute network QoS, such as latency and transfer rate. Our network-aware QoS computation can handle input-dependent QoS, as well. (For example, a video compression service could specify that the service's execution time depends on the amount of input data supplied.)

3. Self-adaptive Network-aware Algorithm. Our selection algorithm is based on a genetic algorithm.

In addition to standard operators, we use tailored operators for mutation, and crossover which leverage our network model. Our self-adaptive algorithm adjusts the use of its operators according to their performance at runtime for a given problem, which e.g. might require a higher or a lower ratio of latency optimization.

In our evaluations, we show that, unlike standard approaches, we achieve a near-optimal latency for our service compositions, while being able to scale with the problem size. Furthermore, we show that our approach works under realistic network conditions and that its self-adaptivity allows it to optimize latency without sacrificing other QoS.

In comparison to [16] we improve 1. by using a k-d tree [3], a popular Nearest Neighbour method, which scales better than our previous locality-sensitive hashing scheme. Most importantly, we refine our algorithm in 3. and make it self-adaptive, allowing it to optimize other QoS besides latency. Furthermore, we added the mentioned evaluations under realistic network conditions.

The structure of this paper is as follows. Section 2 defines our approach. Section 3 evaluates the benefits of our approach. Section 4 compares our approach with related work. Section 5 concludes the paper.

2 Approach

In this section we define our approach. We present our proposed network model first and then describe our network QoS computation method that makes use of this model. After that, we define our proposed network selection algorithm that makes use of the network model, as well.

2.1 Network Model

First, we summarize related network research, before introducing the two parts of our network model: a general network coordinate system that forms the basis of our network-aware approach and a k-d tree implementation that allows us to find services that are close to certain network locations or network paths.

2.1.1 Background

In the related field of network research, it has been established to use coordinate systems to predict network distances (i.e. latencies between any two points) [21]. We will refer to concrete approaches as network coordinate (NC) systems, following [8]. Two state-of-the-art algorithms, Vivaldi [8] and Phoenix [7], reliably build such NC systems in a scalable fashion. Our approach does not depend on a specific NC system; it uses a generic network model based on two-dimensional coordinates that can be fed by NC systems, as in [8], that are based on Euclidean distance models which have been shown to surpass other type of models since [21].

2.1.2 Network Coordinate System

In our cloud scenario, we face the challenge of dealing with a huge number of services, N, for which we have to compute the latency between any of them. Probing all pairwise link distances would require $\mathcal{O}(N^2)$ measurements, giving us exact values, but it would obviously not scale. Also, while services deployed in the cloud might exist for some time (allowing us to cache their latencies), users could frequently show up at new locations, and, thus, they would have to ping all existing services before they could use our approach; this would be prohibitive. Therefore, we use existing NC systems which just require $\mathcal{O}(N)$ measurements in total and $\mathcal{O}(1)$ measurements for adding a new network location.

We decided to base our generic NC system on algorithms that are based on the Euclidean distance model, like Vivaldi [8] which employs a three-dimensional model, because Euclidean models have been proven to work well for this purpose, as we mentioned before. Our approach can use any such network model with any number of coordinates, as long as it can be projected onto a two-dimensional space. While we use the model directly to estimate latencies, we use the two-dimensional space in the heuristic of our mutate operator, as it works sufficiently well and allows for flexibility in changing the actual network model. Accordingly, each network location (including services and users) is projected into a two-dimensional space, as in Fig. 4, and the latency between two locations can be estimated by their Euclidean distance.

2.1.3 K-D Tree

In order for our network-aware selection algorithm to work efficiently, it is not enough to compute the latencies between arbitrary network locations. Additionally, we need to be able to find services that are close to certain network locations or network paths, so that we can efficiently search for service compositions with lower latency. We realize this functionality by implementing a k-d tree which works on top of the two-dimensional space projected from our NC system.

A k-d tree, as in Fig. 5 allows us to run an efficient nearest neighbour (NN) search in $\mathcal{O}(\log n)$. We use

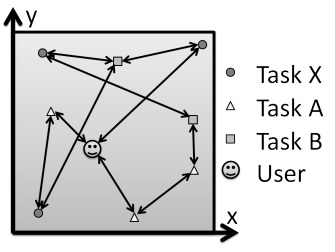


Fig. 4: Network Model

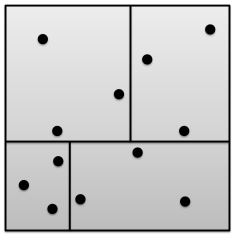


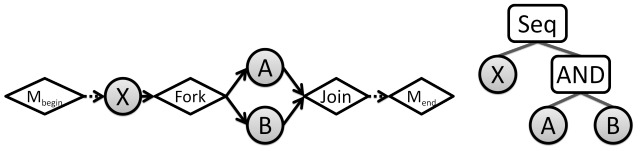
Fig. 5: Example of a K-D Tree

this NN search to find a fixed amount of nearest neighbours for our mutate operator.

While this is different to the complete search of our previous approach [16] through a locality-sensitive hashing scheme, our k-d tree implementation scales better and allows us to reduce the computation time of our mutate operator, which is beneficial for our self-adaptive algorithm, as we will explain later.

2.2 Network-aware QoS Computation

Our network-aware QoS computation consists of two phases. The first phase simulates the execution of the workflow in order to evaluate input-dependent QoS and network QoS. The second phase aggregates these QoS over the workflow structure.



(a) Graph (b) Tree

Fig. 6: Different Representations of a Workflow

1. Simulated Execution. We assume that a workflow is either given as a directed graph as in Fig. 6a or is converted into one (e.g. from a tree as in Fig. 6b), before we simulate the execution of the workflow according to Alg. 1.

2. QoS Aggregation. In the second phase, we take the obtained QoS for each node and aggregate it in a hierarchical manner (over the tree representation as in Fig. 6b) according to the commonly used aggregation rules from [10], [31] that take into account workflow patterns, like parallel(*AND*) or alternative(*OR*) executions and loops. Just for the runtime of the workflow,

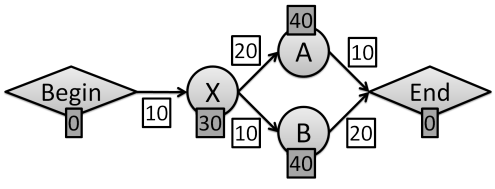


Fig. 7: QoS of a Workflow

Algorithm 1: simulateExecution(graph)

```
foreach vertex v ∈ graph do
    v.execStart = 0
    v.requiredIncoming = |v.incoming|
end
while ∃ v ∈ graph . v.requiredIncoming = 0 do
    pick any v ∈ graph . v.requiredIncoming = 0
    evaluate QoS of v
    v.executionEnd = v.execStart + v.qos.runtime
    foreach w ∈ v.outgoing do
        net = getNetworkQoS(v, w)
        duration = v.resultSize / net.transferRate
        transEnd = v.execEnd + net.delay + trans
        w.execStart = max{w.execStart, transEnd}
        w.requiredIncoming -= 1
    end
    v.requiredIncoming = −1
end
```

we keep the computation from the first phase, because we cannot compute it with a hierarchical aggregation.

3. Example. If we annotate the services of our previous workflow example with execution durations, and their network links with network delays, as in Fig. 7, our algorithm will produce the values of Fig. 8 for the execution times of the nodes ($start/end$) in five steps[1]. This simple example shows that hierarchical QoS aggregation alone would not work, because A and B would be aggregated together first. This makes it impossible to compute the correct network QoS, because the maximum of the delay of the incoming and outgoing nodes of (A,B) each would be aggregated as 20, adding up to 40. However, both paths that go through the incoming and outgoing nodes of (A,B) have an actual cumulative delay of 30.

#	Begin	X	A	B	End
0	0 / ?	0 / ?	0 / ?	0 / ?	0 / ?
1	**0 / 0**	10 / 0	0 / ?	0 / ?	0 / ?
2	0 / 0	**10 / 40**	60 / ?	50 / ?	0 / ?
3	0 / 0	10 / 40	**60 / 100**	50 / ?	110 / ?
4	0 / 0	10 / 40	60 / 100	**50 / 90**	110 / ?
5	0 / 0	10 / 40	60 / 100	50 / 90	**110 / 110**

Fig. 8: Simulated QoS

2.3 SanGA

Our selection algorithm, SanGA, is based on a genetic algorithm, which can solve the service composition problem in polynomial time. First, we give a basic overview of genetic algorithms. Then, we introduce the customizations of our algorithm that are tailored to the problem of network-aware service composition.

[1]the computational steps are denoted in the # column

2.3.1 Genetic Algorithm

In a genetic algorithm (GA), possible solutions are encoded with genomes which correspond to the possible choices available in the problem. For a service composition, a genome, as in Fig. 9, contains one variable for each task of the workflow, with the possible values being the respective concrete services that can fulfill the task.

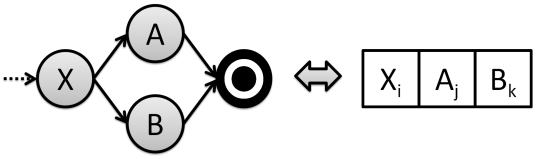


Fig. 9: Workflow with corresponding Genome

Given a fitness function that evaluates how good a possible solution (individual) is, the GA iteratively finds a near-optimal solution as follows. First, an initial population is generated. Then, in every iteration, individuals are selected, changed either by mutation or crossover operators, and inserted into the new population for the next iteration. This procedure is repeated until a convergence criteria is met, which checks if the fitness of the population has reached a satisfactory level, or if the fitness does not improve anymore.

2.3.2 Initial Population

The initial population is usually generated completely randomly. While it is desirable to keep some randomness for the GA to work properly, adding individuals that are expected to be better than average has beneficial effects on both the speed of convergence and the final solution quality. While we previously [16] used a heuristic on 25% of the initial population, this proved only to be efficient when optimizing latency. Thus, in SanGA we place more emphasis on randomness and generate six percent of the initial population by running a very simple Hill-Climbing algorithm, a gradient method which searches possible solutions in a local neighbourhood via steepest ascent [6]; this ensures that the generation of the initial population takes less than ten percent of the total runtime.

2.3.3 Selection

We select individuals that are to be reproduced via mutation and crossover through a roulette-wheel selection. The probability of an individual with fitness f_i to be chosen out of a population of size N is:

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$$

Thus, it is a fitness-proportionate selection ensuring that better individuals have a higher chance of being chosen. There is no need to customize the selection operator to our problem, as it is already reflected in the fitness function. We improved the runtime of the roulette-wheel selection compared to our previous approach [16] through an implementation using stochastic acceptance according to [19], allowing for a complexity of $\mathcal{O}(1)$.

2.3.4 Elitism

Elitism keeps the top-k individuals seen so far in order to achieve convergence in a reasonable time. We keep the most fit percent of our population in each generation.

2.3.5 Operators

There exist two kind of operators, mutation and crossover operators. For each, we use both, generic operators and operators which are tailored to our network-aware selection problem.

1. Mutation Operators. The purpose of mutation operators is to randomly change individuals slightly in order to improve their fitness and to escape local optima. For this part, we partly use the standard uniform mutation operator which changes each element of the genome with equal probability, and on average changes one element, as shown in Fig. 10.

Fig. 10: Uniform Mutation

In addition, we use our own *NetMutation* operator, which does a combination of mutation and a small local search. First, parts of the genome are selected randomly. For each part, we try to exchange the chosen service with randomly chosen nearest neighbours (in our k-d tree) of the network location between the preceding and following service (graph-wise). The best replacement is kept.

2. Crossover Operators. A crossover operator combines two individuals (parents) to create new individuals (offspring) that can draw from the good points of both parents. A standard single-point crossover operator is depicted in Fig. 11; a single point of the genome is chosen, and the new individuals are recombined from both parents around that point.

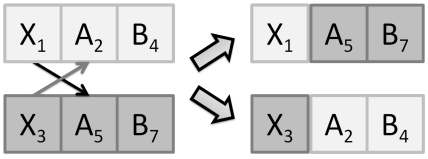


Fig. 11: Single-Point Crossover

In an analogous way, two-point crossover, three-point crossover, etc. can be implemented until one has a uniform crossover operator in which every part of the genome of the parents is randomly distributed

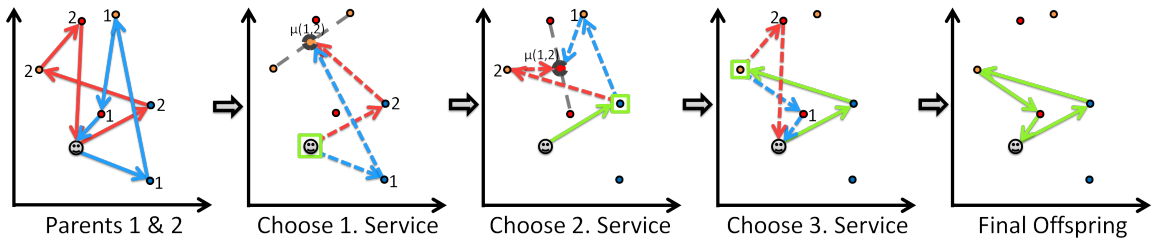


Fig. 12: Our custom *NetCrossover* Operator

among the offspring. Along with an uniform crossover, we use our own *NetCrossover* operator which, for each part of the genome, chooses a value from its parents with a certain probability as follows. We build one final offspring, task by task, by choosing between the parents par_1 and par_2. Let d_{prev_i} be the distance to the location of the previous service (or the user at the start), and d_{next_i} be the distance to the location of the following service (or the user at the end). If the following service is not chosen yet, we take the average of the locations of both possible following services from the two parents. The probability that we choose a parent par_i for the current task then is:

$$p_i = 1 - \frac{d_{prev_i} + d_{next_i}}{\sum_{j=1}^{2} \left(d_{prev_j} + d_{next_j}\right)}$$

The effect is that the offspring is a kind of smoothened version of the parents with regard to the network path. This quickly minimizes the latency of our offspring. Note that when the parents are spread in a similar fashion network-wise, our *NetCrossover* comes very close to standard uniform crossover.

In addition we also employ an *AttributeCrossover* operator which can be set to optimize a single QoS. It will then for each task pick the service of the parent optimizing that particular QoS. We use this operator to optimize other QoS besides latency.

2.3.6 Self-Adaptivity

In our previous approach [16], we defined less operators and fixed the probability for applying each of them according to our experiences. Instead of this naïve approach, which tries to pick good probabilities for one specific setting, here we opted for a self-adaptive approach which adjusts the operators' probabilities on-the-fly according to their effectiveness.

Algorithm 2: initialOpProbs(muOps, crOps)

```
foreach op m ∈ muOps do
    mu.probability = 0.5/mutationOps.size()
end
foreach op c ∈ crOps do
    cr.probability = 0.5/crossoverOps.size()
end
```

We will first describe how we track the operators' effectiveness, before we specify step by step how we determine good probabilities for them.

We used several individual ideas from approaches for adapting operator probabilities, such as PM, AP, and PPM (see [12]). Note that we had to add our own ideas and change the way all these individual bits were combined, fitting them to our problem, before we were able to achieve good practical results at all.

1. Operator Statistics. In order to judge the effectiveness of an operator, each time an operator is applied we record the "improvement" ratio of the offspring's fitness and its parents' (average) fitness. As we are maximizing the fitness, an improvement ratio greater than one tells us that a operator is useful. For each operator we record a fixed number of its most recent improvement ratios and maintain the average and maximum value of these ratios, as well.

2. Initial Probabilities. We assign the initial probabilities, according to Alg. 2, as follows. First, we split the overall probability equally across mutation and crossover. Then, we assign each operator an equal share of the overall mutation or crossover probability, respectively. Compared to splitting the probability equally among all operators, as it is common, this ensures that neither mutation nor crossover is over-represented in the beginning.

3. Adaptive Probabilities. As the improvement ratios of our operators might only span a small range, we have to amplify their values in order to effectively distinguish between operators. We do this through the following amp function, which can be customized by suitable $(\epsilon, \gamma_1, \gamma_2)$.

$$amp(r) = \begin{cases} (1+\epsilon)^{(1+r)^{\gamma_1}} & \text{if } \exists\ i \text{ with } f_i > 0 \\ (1+r)^{\gamma_2} & \text{otherwise} \end{cases}$$

If there is any solution (candidate) currently satisfying the user's constraint we have to discriminate more, as the mentioned range should be quite small. On the contrary, when many solutions violate the constraints, less discrimination is sufficient, as these solutions can have a negative fitness (through penalties), increasing the range of fitness values.

$$effs(ops) = \frac{\sum_{o \in ops} o.prob \cdot amp(o.maxRatio)}{\sum_{o \in ops} o.prob}$$

Given help function $effs$, first, we determine the overall effectiveness of all mutate and crossover operators, respectively, and distribute the probability between mutate and crossover, as in Alg. 3.

Algorithm 3: distributeTotalProbs(muOps, crOps)

```
muProb = effs(muOps) / (effs(muOps)+effs(crOps))
crProb = 1-muProb
distributeTypeProbs(muOps, muProb)
distributeTypeProbs(crOps, crProb)
```

Then, we distribute the probabilities within mutate and crossover operators according to Alg. 4.

Algorithm 4: distributeTypeProbs(ops, prob)

```
useful = {o ∈ ops | o.maxRatio > 1.00001}
distributeProbs(|useful| > 0 ? useful : ops, prob)
```

First, we check if there are "useful" operators which can actually produce offspring with a greater fitness than their parents. If there are such operators, we only distribute the probability among them, else we distribute it among all operators of the same kind.

Algorithm 5: distributeProbs(ops, prob)

```
distribute(ops, 0.8 · prob, f : o ↦ o.maxRatio)
distribute(ops, 0.2 · prob, f : o ↦ o.avgRatio)
```

In Alg. 5 we distribute the probabilities among the chosen operators according to the maximum and the average of their improvement ratio. We distribute most of it according to the maximum ratio, as it best shows the potential of a genetic operator; for instance, mutate might not always produce good results, but once in a while we can get big improvements.

$$eff(ops, op, f : op \mapsto val) = \frac{amp(f(op))}{\sum_{o \in ops} amp(f(op))}$$

Finally, we do the actual distribution given either the maximum or average improvement ratios. Given above help function eff to compute the effectiveness of an operator regarding the given ratio, we distribute the probability according to Alg. 6.

Algorithm 6: distributeProbs(ops, prob, f)

```
opMax = max{op ∈ ops | f(op)}
opMax.probability += 0.8 · prob
foreach op op ∈ ops do
    op.probability += eff(ops, op, f) · 0.2 · prob
end
```

Most of the probability goes to the operator with the highest ratio and the rest is distributed proportionally according to their effectiveness. Finally, the operators' probability values are adopted with a learning rate in each iteration of the GA.

3 Evaluation

In this section we evaluate our approach. First, we describe the setup of our evaluation and the algorithms we want to compare. Then, we evaluate their optimality and scalability when optimizing the latency or multiple QoS. Finally, we analyze the self-adaptivity of our approach.

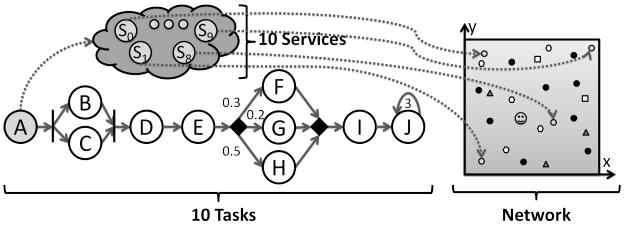


Fig. 13: Example Workflow of Size 10

3.1 Setup

The evaluation was run on a machine with 32 AMD Opteron cores with 2.4 GHz. All algorithm instances were evaluated in separated threads and granted a single exclusive core, while memory was shared and less than 1 GB per instance was needed.

Network Generation. We used a PlanetLab Trace Data Set[2] with real-world latency data. We filtered out some nodes which were not well-connected in the dataset and ended up with the latency data of about 110 real nodes from the PlanetLab[3] project. We then built a network model from this data, considering only the latency of each node to 32 other randomly chosen nodes. Finally, based on these real nodes, we randomly created similar nodes resulting in 100,000 unique network locations.

Workflow Generation. We generated workflows randomly. For each task, we inserted a parallel control structure with a chance of 5%, having 2, 3 or 4 branches and each branch having 1, 2 or 3 tasks, with corresponding probabilities of $\frac{1}{2}$, $\frac{1}{3}$, and $\frac{1}{6}$, respectively. Also, we randomly chose a number of network locations for its services. Figure 13 depicts an example of a generated workflow of length 10.

Workflow Size. We generated workflows with sizes between 10 and 100 (in steps of 10); these sizes are comparable to those of other recent approaches with 10 [2], up to 50 [31] or 100 [1]. The standard workflow size was **50** unless noted otherwise.

Service Instances per Task. We varied the number of services (instances) available per task between 500 and 5000 (in steps of 500), which is considerably more than what most of the previous studies have used (50 [31], 250 [25] or 350 [18]). As mentioned, the reason that we consider such large numbers is that previous studies just considered services per task, while we want to differentiate between instances of the same

[2]http://ridge.cs.umn.edu/pltraces.html
[3]https://www.planet-lab.org/

service (that are deployed in different network locations). Therefore, we think it is also more accurate to explicitly refer to them as service instances per task from now on. The standard number of service instances per task was **500** unless noted otherwise.

QoS. The QoS attributes of each service were generated independently from each other, according to a uniform distribution drawn from the range of $[0, 1]$. The network latency between two services or between a user and a service was computed from the built network model, which estimated the real latencies with an average error of less than 15 ms per node of our chosen dataset.

3.2 Algorithms

We chose several algorithms and compared their optimality and scalability.

Random. A simple random algorithm that picks all services at random from the available choices. It shows the expected value of a randomly chosen solution and provides a baseline that more sophisticated algorithms should be able to outperform easily.

Dijkstra=. An optimal algorithm for the shortest-path problem, operating on the real latency values. It can be used to find the service composition with the lowest latency, against which we can conveniently compare our algorithm. However, Dijkstra cannot be used with QoS constraints or with multiple QoS.

Dijkstra˜. Same algorithm, but operating on the network model, not the real latency values.

GA. A standard GA using uniform crossover and mutation with a population size of 100. The standard QoS model (without Network QoS) is used; each service provides its execution time plus its expected (maximum) latency. This is the current standard approach for the service composition problem.

GA*. The same settings as *GA*, except that we provide *GA** with our network model that allows it to accurately predict the network latency. *GA** represents a naïve adoption of the current standard approach to service composition in distributed settings.

NetGA. Our previous network-aware approach introduced in [16]. The size of the population is 100.

SanGA. Our new self-adaptive and network-aware approach as described in Sect. 2. The size of the population is 100.

GA* 50, NetGA 50 and SanGA 50. *GA**, our previous and our current approach with a smaller population size of only 50.

Convergence. The convergence criteria is identical for all GAs: In principal, if the fitness of the best individual does not improve by at least one percent over the last 40 iterations, the algorithm terminates. To allow for some flexibility, we continue while there still is some improvement of the average fitness of the population from iteration to iteration. This convergence criteria allows all GAs to find good solutions.

3.3 Latency Optimization

First, we evaluate all algorithms in a setting where they only optimize latency, allowing us to get a detailed comparison with our previous approach [16]. Thus, the achieved utility corresponds to the optimality of the latency. We introduce a linear scaling for the latency such that 100% corresponds to the optimal latency computed by **Dijkstra=**, which knows the real network latencies, and such that 50% corresponds to the expected latency if services are chosen randomly. Note that every data point in Fig. 14d aggregates 32 test cases, and, in the other graphs in Fig. 14, every data point aggregates 128 test cases.

3.3.1 Validity of Network Model

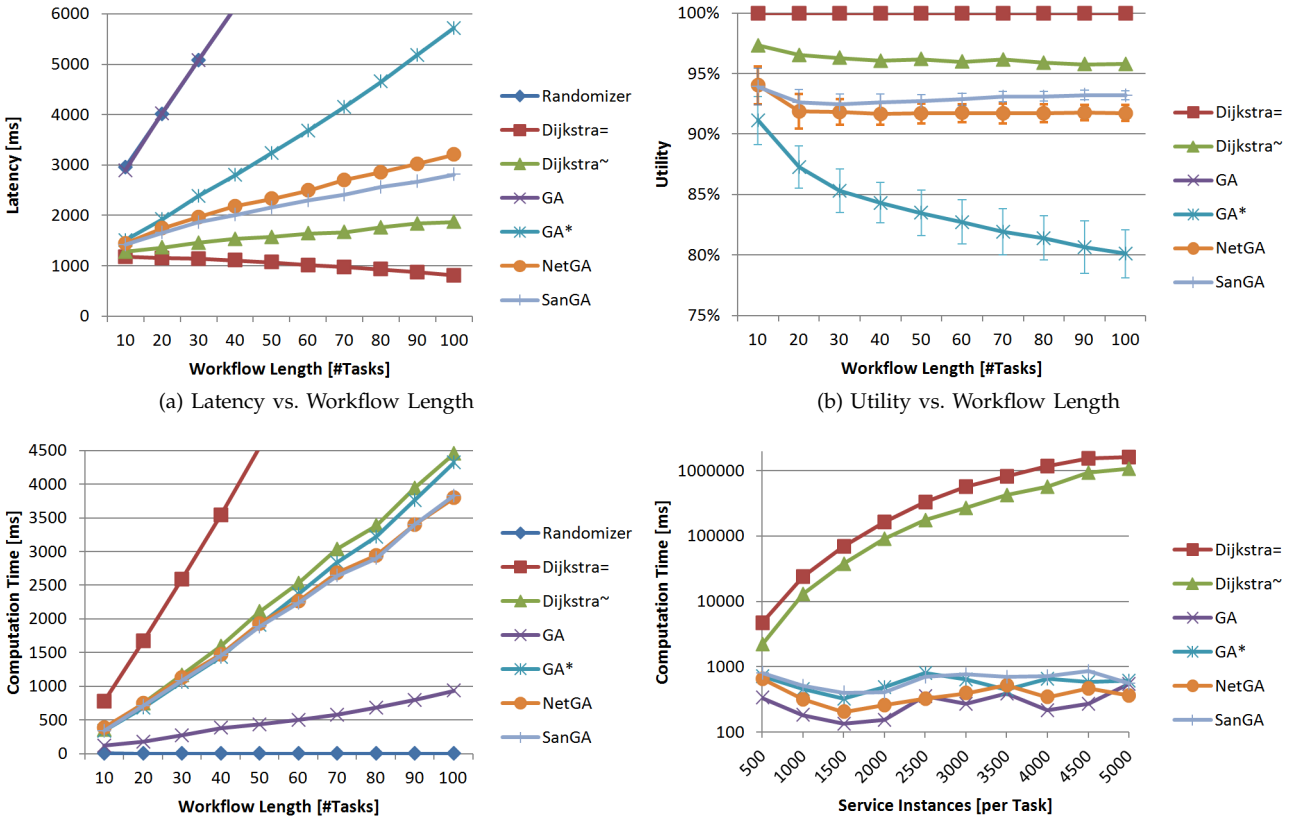
We want to show briefly that common network models, of which we implemented Vivaldi [8], work reasonably well for the real-world dataset we obtained. From our experiments in Fig. 14a, we can observe that the latency computed by **Dijkstra˜** comes quite close to **Dijkstra=** for workflow sizes ≤ 50 and that their difference keeps increasing in a slow linear fashion with the workflow size. In terms of optimality, **Dijkstra˜** stays within five percent of the optimal latency for the workflow sizes we evaluated. We think this is a quite reasonable approximation which could probably be improved even further by a more sophisticated implementation of the network model - which is out of the scope of our paper though.

Note that all other algorithms only work with the network model, not with the real latencies. Therefore, they cannot optimize towards a better latency than **Dijkstra˜**. Also note that **GA** achieves the same latency as the **Random** algorithm. Thus, we can conclude that it is mandatory to use the network model for an accurate latency prediction, instead of using worst-case latencies as it is current practice.

3.3.2 Optimality

Figure 14a shows the latency of the service compositions obtained. We can see that both **NetGA** and **SanGA** outperform the standard **GA*** algorithm, with **SanGA** having a slight edge over **NetGA**. For instance, for workflow size 100, **SanGA** finds compositions with ≈ 1.5 times the latency of compositions found by **Dijkstra˜**, resulting in an additional latency of ≈ 1 second; in comparison, **GA*** only finds compositions with ≈ 3 times the latency, resulting in ≈ 4 additional seconds of latency.

Figure 14b plots the utility of the obtained service compositions. Additionally, the standard deviation has been added as error bars, to show how much the algorithms deviate on average. As we can see, both **NetGA** and **SanGA** achieve an optimality of 91% and 93%, respectively, independent of the workflow size and with a very small standard deviation. On the contrary, the optimality of **GA*** decreases with increasing



(a) Latency vs. Workflow Length

(b) Utility vs. Workflow Length

(c) Computation Time vs. Workflow Length

(d) Computation Time vs. Service Instances

Fig. 14: Latency Optimization

workflow size. Thus, the improvement of **GA*** over a random algorithm diminishes quite drastically with increasing workflow size, and for even bigger workflows using **GA*** might not a be a desirable option if one wishes to obtain a good latency.

3.3.3 *Scalability*

The scalability of approaches in terms of computation time is important both in regard to an increasing workflow size and in regard to an increasing number of service instances per task. As we can see in Fig. 14c, all approaches scale roughly linear in regard to the workflow size. **NetGA** and **SanGA** take slightly less time than **GA*** and **Dijkstra~** to compute solutions; **Dijkstra=** takes roughly double the time of **Dijkstra~** and **GA** is quite faster than the other GAs, but is not relevant, as the latency of its solutions is not better than that of the **Random** algorithm.

Figure 14d has a logarithmic scale and shows that while the GAs' computation time is independent of the number of service instances per task, the computation time of both Dijkstra algorithms increases quadratically, quickly resulting in computation times of several minutes, even for a workflow of size 50. Thus, only GAs can provide scalability, providing solutions in real-time, in a matter of seconds.

3.3.4 *Smaller Population Size*

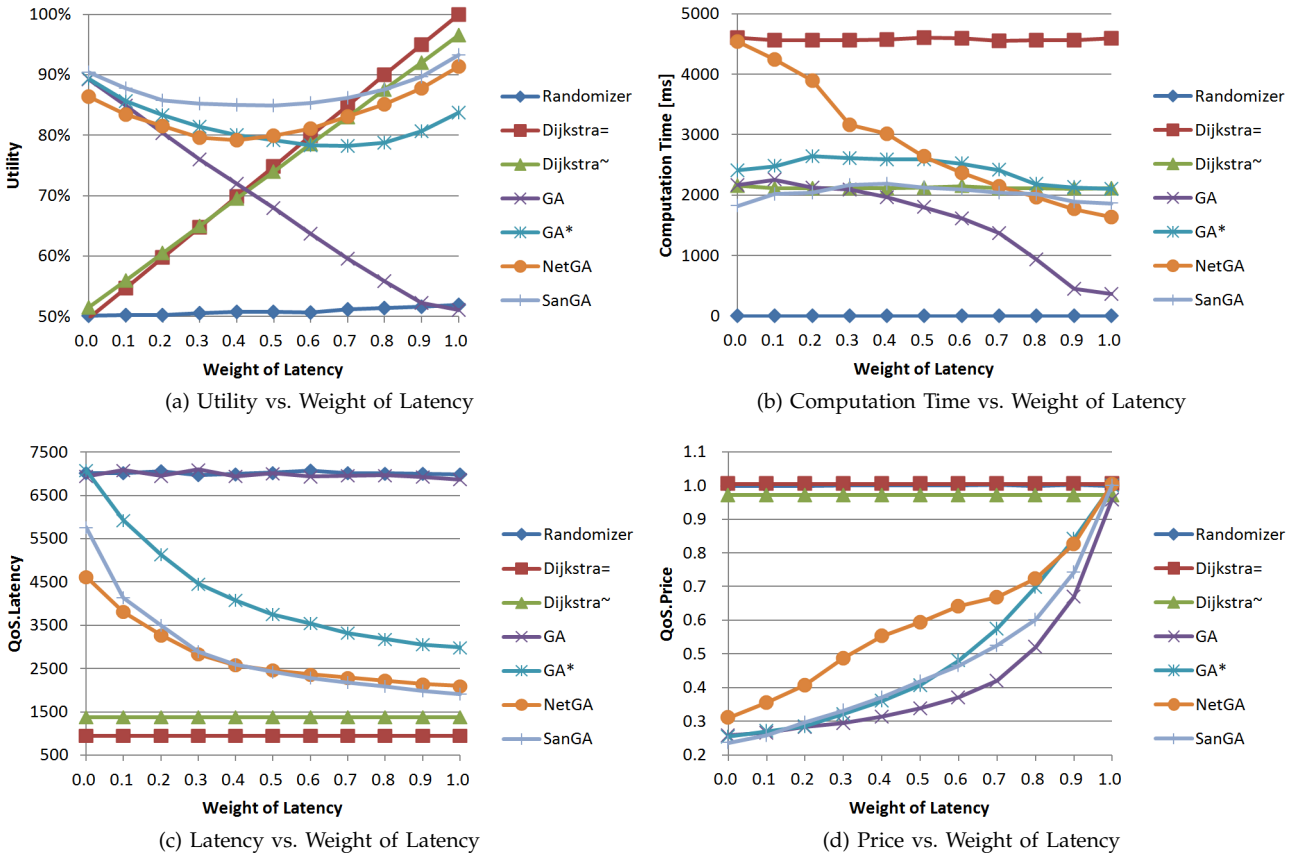
The GAs with a smaller population size of 50, namely **GA* 50**, **NetGA 50** and **SanGA 50**, are not plotted in Fig. 14, because they would clutter the previous graphs. Instead we give their relative performance in regard to their corresponding algorithms with a population size of 100 in Fig. 15. In general, we can say that they achieve almost the same utility ($\geq$ 96%) while only taking about half of the computation time. Thus, when slightly less optimal solutions are acceptable, using a GA with a reduced population size is a good trade-off. Note that this trade-off is best for **SanGA 50**.

Algorithm	Utility [%]	Comp. time [%]
GA* 50	[96.3 – 98.3]	[54 – 64]
NetGA 50	[97.4 – 98.4]	[62 – 69]
SanGA 50	[98.6 – 99.2]	[49 – 59]

Fig. 15: GAs with smaller Population Size (Latency Optimization)

3.4 QoS Optimization

Next, we evaluate the optimization of multiple QoS which we did not do previously [16]. Our approach works for any number and any kind of QoS which can be quantified; the utility is computed as a common



(a) Utility vs. Weight of Latency

(b) Computation Time vs. Weight of Latency

(c) Latency vs. Weight of Latency

(d) Price vs. Weight of Latency

Fig. 16: QoS Optimization

weighted sum, adding up the normalized values of all QoS multiplied by weights. For the sake of simplicity, we just focus on evaluating two QoS, latency and price; though any other QoS could be used instead, as our results mainly depend on the weight of latency. The latency values are scaled as in the previous evaluation, and for price we directly map 100% to the lowest and 0% to the highest possible price. Note that except for the extreme cases when just optimizing price, or just optimizing latency, there might not exist a single solution which achieves 100% utility. Also note that every data point of the graphs in Fig. 16 aggregates 256 test cases.

3.4.1 *Optimality*

Figure 16a confirms our open hypothesis from [16]: as we can see, when comparing **GA*** and our previous approach **NetGA**, **GA*** overtakes **NetGA**'s utility once the weight of latency drops below 0.5, or, in other words, 50% of the utility. In contrast to that, our new self-adaptive approach **SanGA** always achieves more utility than the standard **GA***, making it superior when optimizing an kind of QoS, not just latency.

On a side note, we can see that **GA** is not able to optimize the latency and drops to **Random**'s value of utility with increasing weight of latency; the two Dijkstra algorithms on the other hand drop to **Random**'s level of utility with decreasing weight.

3.4.2 *Scalability*

Figure 16b confirms that the scalability does not deviate from the results in Sect. 3.3, except for **GA** and **NetGA**. As latency becomes less important, the performance gap between **GA** and **GA*** becomes smaller, because **GA** can only optimize QoS other than latency. In **NetGA**'s case, the higher costs of its latency-specific operators increase its computation time for small weights of latency; normally, these costs would be offset by a faster convergence, but this cannot be achieved when optimizing QoS other than latency. All other GA's computation times seem to be relatively independent from the weight of latency.

3.4.3 *Smaller Population Size*

As in the previous evaluation, we just give the relative performance of the GAs with a smaller population size of 50, namely **GA* 50**, **NetGA 50** and **SanGA 50** (in regard to their corresponding algorithms with a population size of 100). Similarly, we can say that they achieve almost the same utility ($\geq$ 94%) while only taking about half of the computation time. **SanGA 50** shows the best trade-off again.

Algorithm	Utility [%]	Comp. time [%]
GA* 50	[95.4 – 96.8]	[45 – 52]
NetGA 50	[94.4 – 98.6]	[40 – 51]
SanGA 50	[96.3 – 99.1]	[46 – 53]

Fig. 17: GAs with smaller Population Size (QoS Optimization)

3.4.4 QoS

We can get even more specific insights by analyzing the individual QoS of the service compositions found by the algorithms for different weights of latency.

Latency. Figure 16c shows the latency of the compositions. A good pattern for an algorithm would be to have a random latency for weight 0, and better latency with increasing weight, until for weight 1 a near-optimal latency is reached. **GA**, **Random** and the Dijkstra algorithms do not exhibit this pattern, always returning an equally bad or good latency for all weights. On the other hand, **GA***, **NetGA**, and **SanGA** exhibit this pattern to varying degrees. **SanGA** beats **NetGA**, especially for small weights, also outperforming **GA*** for all weights. For weights close to 0 though, it is hard to evaluate if **SanGA** hits the perfect trade-off. We would expect almost random latencies as for **GA***, but **SanGA** still seems to optimize the latency somewhat compared to **Random**. This shows the limit of our approach's self-adaptivity; it is possible to reduce the latency optimization, but not to eliminate it completely.

Price. Figure 16d shows the price of the compositions. Similarly, a good pattern for an algorithm would be to find very cheap compositions for weight 0, and find compositions with an increasing price until for weight 1 a price similar to the **Random** algorithm is found. As we argued before, **GA** can only optimize the price in our evaluation, so **GA**'s pattern favors a cheap price too much. In regard to price, **GA*** exhibits a good pattern which can be seen as a standard, as the other GA's do not employ any particularly effective operators for optimizing price. Except for very big weights ≥ 0.8, our previous approach **NetGA** fails to come close to **GA***, showing that is not efficient at optimizing QoS other than latency. In contrast to that, our self-adaptive algorithm **SanGA** closely follows **GA*** for weights $0.2 - 0.6$, and even manages to surpass it for other weights. This shows that our new approach is successful in adapting itself to optimize other QoS, such as price.

3.4.5 Self-Adaptivity

Finally, we analyze the self-adaptivity of our **SanGA** algorithm in detail. First, we summarize the genetic operators defined in Sect. 2.3.6 which we employed in our evaluations. Then, we discuss how the probabilities of these operators were adapted by our **SanGA** algorithm and how effective our operators where in different evaluation settings.

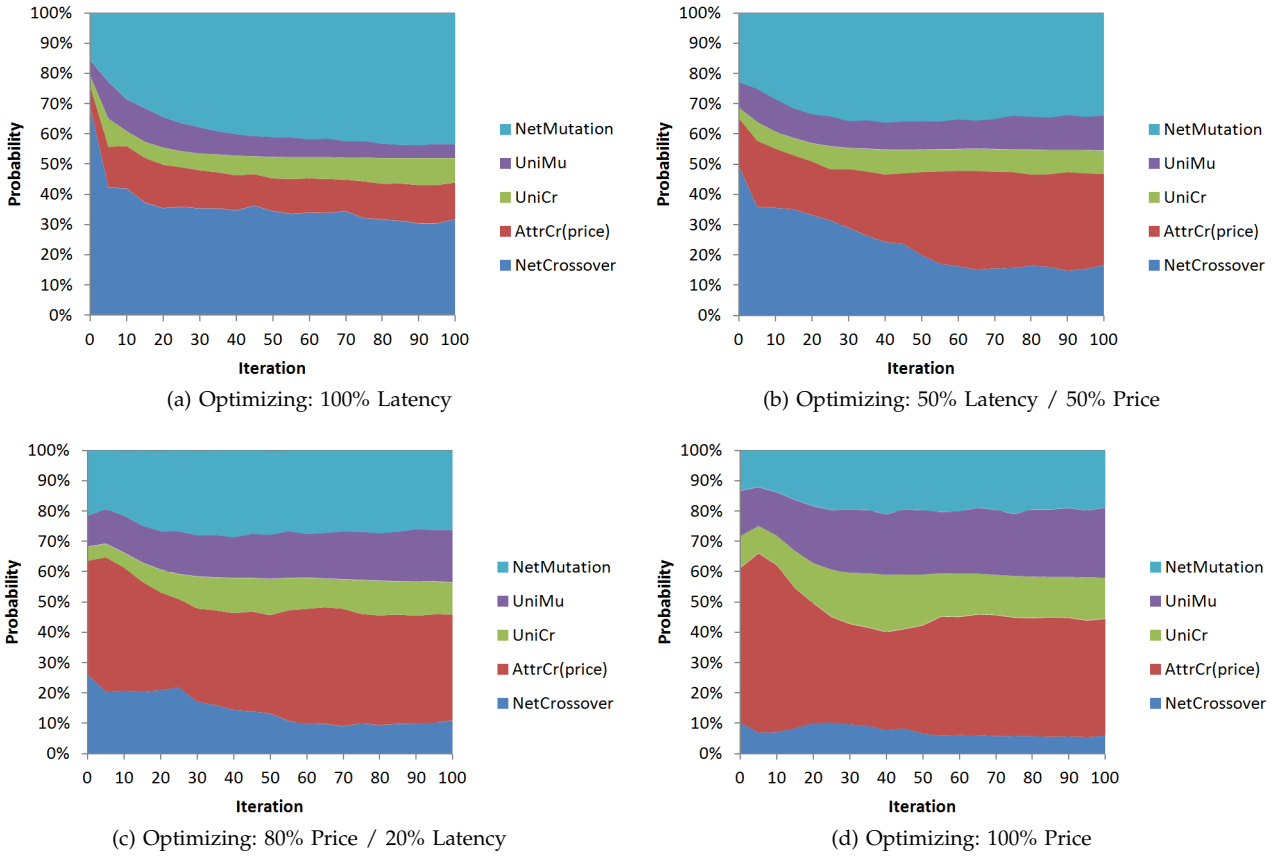
Operators. As **mutation** operators we employ a uniform mutation operator (*UniMu*) and our *NetMutation* operator, both modifying a single variable of the genome. We also tried variations of these operators changing two or four variables at once, but we could not observe a significant benefit. As **crossover** operators we employ a uniform crossover (*UniCr*), a specialized *NetCrossover*, and an *Attribute Crossover* for price, "*AttrCr(price)*". The purpose of *AttrCr(price)* is to balance *NetCrossover* when optimizing the price, as *UniCr* does not dominate *NetCrossover* as much; using *AttrCr(price)* does not change the overall trends observed in our evaluations though. Furthermore, in order for our adaptive algorithm to be effective, we tried to make sure that all operators took up a comparable computation time. In particular, implementing *NetMutation* with a k-d tree and further limiting its local search was helpful in not favoring it too much over *UniMu* (which is still faster, of course).

Aggressiveness. By tuning the parameters of the amp function from Sect. 2.3.6, the adaptation becomes more or less aggressive. If it is not aggressive enough, all operators will end up with very similar probabilities which is not good. If it is too aggressive on the other hand, there will be high fluctuations of the probabilities. Thus, we found that in our experiments $(0.4, 4, 6)$ worked best for amp's $(\epsilon, \gamma_1, \gamma_2)$ parameters.

Probabilities. We adapt the probabilities of those operators during the execution of **SanGA** according to the rules given in Sect. 2.3.6. Figure 18 shows how the probabilities of the operators changed during the first 100 iterations for different weights of latencies. The probabilities shown were averaged over all of the runs of **SanGA** during the evaluations from Fig. 16. Overall, the share of mutation operators starts at about 20% in the first iterations, increases to slightly more than 50% within about 30 iterations, and keeps this value from then. A similar but slightly stronger pattern can be seen for other weights of latency, as well. This stems from the fact that, in the first iterations of a genetic algorithm, crossover operators are more effective than mutation operators. Being able to make use of this fact is one of the benefits of our self-adaptive algorithm.

When optimizing just the latency, we get the distributions of Fig. 18a. As expected, *NetMutation* dominates *UniMu*, and *NetCrossover* dominates *UniCr* and *AttrCr(price)*. That *AttrCr(price)* does not get dominated more, shows the mentioned limit to our self-adaptivity, as we cannot be too aggressive in order to avoid wrong adaptations.

Once we start to optimize latency and price equally, we see a different pattern in Fig. 18b. *NetMutation* dominates *UniMu* less, but is still dominant, showing that *NetMutation* is quite effective in improving the utility of solutions. Overall, *NetCrossover* and *At-*



(a) Optimizing: 100% Latency

(b) Optimizing: 50% Latency / 50% Price

(c) Optimizing: 80% Price / 20% Latency

(d) Optimizing: 100% Price

Fig. 18: Operator Probabilities during the first 100 Iterations

trCr(price) both divide the main share of the crossover operators quite evenly. *UniCr* gets a more consistent, slightly higher share, but it is not as effective as the other crossover operators.

Once we focus on optimizing the price more, we can see *NetCrossover*'s share decreasing significantly in Fig. 18c. While *NetMutation*'s share also keeps decreasing, it is still bigger than *UniMu*'s; apparently *NetMutation* seems to be a quite effective operator even under such conditions.

When we just optimize the price, Fig. 18d shows that *UniMu* gets a slightly bigger share than *NetMutation*. As we said before, this indicates that *NetMutation* also seems to work well as a general mutation operator. Alternatively, it could be of value to investigate if using *UniMu* more often would be more efficient; in that case our criteria for measuring the effectiveness of operators would probably need to be tweaked or extended. Among the crossover operators, on the other hand, *NetCrossover* has been marginalized by the other operators, and *AttrCr(price)* dominates *UniCr* by a margin of 4:1. This shows *NetCrossover* is only effective for optimizing the latency. Furthermore, QoS-specific operators, like *AttrCr(price)*, seem to beat generic operators. Therefore, being able to use many QoS-specific operators in a self-adaptive fashion seems more effective than using generic operators.

4 Related Work

In this section, we survey related work from the following four categories, sorted from generally related to more specifically related material.

4.1 QoS-aware Service Composition

The foundation for our research is in [32]. That paper introduces the QoS-aware composition problem (CP) by formalizing and solving it with (Linear) Integer Programming (IP). Since then, genetic algorithms (GAs) [5], [9], as well as different heuristic approaches [31], [1], [18], [14], [15] have all been shown to be more scalable than IP. While the scalability of these approaches seems satisfactory in terms of complexity, there still is room for improvements by taking more advantage of the specific nature of the CP. Similar to [25], we do this by customizing the optimization operators of GAs. While they adjust the operators for the general CP, focusing more on constraints and penalties, we customize them for our network-aware CP.

All these approaches share the same definition of the CP, which ignores the QoS of the network connecting the services. Except for IP which requires a linear function to compute the utility of a workflow, most approaches can be easily augmented with our

network-aware QoS computation and some also with our network-aware operators.

The recent skyline approach [2] that prunes QoS-wise dominated services, would have to be modified to prune such services per network location though, greatly limiting the benefit of its pruning.

4.2 Advanced QoS

The approaches described above simply aggregate static QoS values defined in SLAs. A QoS evaluation depending on the execution time is described in [17]. In a similar way, our algorithm computes the starting time of the execution of each service, so it can be used to compute time-dependent QoS, as well. SLAs with conditionally defined QoS are described in [13]; these can be considered to be a special case of input-dependent QoS, and, thus, they can be handled by our approach as well.

Constraints whereby certain services have to be executed by the same provider are given in [20]. Placing such constraints on critical services could also reduce network delays and transfer times. However, doing so would require significant effort to introduce heuristic constraints and there is no guarantee that this would lead to a near-optimal solution.

4.3 Network QoS

Many studies, such as [29], [11], deal with point-to-point network QoS, but do not consider services and compositions from SOC. One of the few examples that considers this is [30], which looks at service compositions in cloud computing. The difference between this study and our approach is that instead of a normal composition problem, a scheduling problem is solved where services can be deployed on virtual machines at will. However, it is not clear if that approach can handle the computation of input-dependent QoS and network transfer times, because the authors did not provide a QoS algorithm.

4.4 Workflow Scheduling

In the related field of workflow scheduling, a workflow is mapped to heterogeneous resources (CPUs, virtual machines, etc.), and information about the network is sometimes considered, as well. The idea is to achieve a (near-)optimal scheduling minimizing the execution time; this is often done by using greedy heuristic approaches, like HEFT [26]. The reason such greedy algorithms seem to suffice is that only one QoS property (response time) is optimized, and that no QoS constraints have to be adhered to, greatly simplifying the problem. Thus, while the setting is similar to ours, the complexity of the problem is quite different, because we optimize multiple QoS properties under given QoS constraints. In addition, algorithms, like HEFT [26], often work like a customized Dijkstra, which does not scale well in our problem setting, as our evaluation shows.

5 Conclusion

In this paper we described our self-adaptive network-aware approach to service composition, SanGA. Our approach applies a realistic network-model, accurately estimates the network QoS, and employs a self-adaptive genetic algorithm which is very efficient at optimizing latency, while being able to adapt to optimize any kind of other QoS, as well. In our evaluations we showed that our network-model can work effectively for estimating real-world network latencies, and that our algorithm beats current approaches when optimizing the latency of compositions, while not losing out in regard to other QoS. Thus, our SanGA approach should be used whenever services are distributed over a network, for instance in cloud settings. Additionally, we show that a self-adaptive approach with QoS-specific optimization operators can be more effective than a generic approach. Furthermore, our evaluations show that we can achieve this with the same or even slightly less computational time; the overhead incurred by realizing the self-adaptivity is offset by quicker improvements and faster convergence.

As future work, we would like to conduct some experiments in a large-scale distributed setting to get further empirical validation of our approach. Also, extending the self-adaptivity and experimenting with additional QoS-specific operators in the context of service compositions could broaden our insights gained in regard to the benefits of self-adaptivity in our SOC domain. Additionally, we could like to integrate our approach with more complex QoS models, e.g. probabilistic ones, some of which we have explored in our previous research.

6 Acknowledgments

We would like to thank Florian Wagner for the countless discussions and the detailed feedback that helped us improve our approach. Also we would like to thank Michael Nett and Michael Houle for kindly offering us one of their high-spec machines for running our extensive evaluations. Adrian Klein is supported by a Research Fellowship for Young Scientists from the Japan Society for the Promotion of Science.

References

[1] M. Alrifai and T. Risse. Combining Global Optimization with Local Selection for Efficient QoS-aware Service Composition. In *WWW '09: Proceedings of the 18th International Conference on World Wide Web*, pages 881–890, 2009.

[2] M. Alrifai, D. Skoutas, and T. Risse. Selecting Skyline Services for QoS-based Web Service Composition. In *WWW '10: Proceedings of the 19th International Conference on World Wide Web*, pages 11–20, 2010.

[3] J. L. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Commun. ACM*, 18(9):509–517, Sep. 1975.

[4] K. Candan, W.-S. Li, T. Phan, and M. Zhou. Frontiers in Information and Software as Services. In *ICDE, IEEE 25th International Conference on Data Engineering*, 2009.

[5] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*, pages 1069–1075, 2005.

[6] A.-L. Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *Compte Rendu des S'eances de L'Acad'emie des Sciences XXV*, S'erie A(25):536–538, Oct 1847.

[7] Y. Chen, X. Wang, C. Shi, E. K. Lua, X. Fu, B. Deng, and X. Li. Phoenix: A Weight-based Network Coordinate System Using Matrix Factorization. *IEEE Transactions on Network and Service Management*, pages 1–14, 2011.

[8] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a Decentralized Network Coordinate System. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 15–26. ACM, 2004.

[9] M. Jaeger and G. Mühl. QoS-Based Selection of Services: The Implementation of a Genetic Algorithm. In *KiVS 2007 Workshop: Service-Oriented Architectures und Service-Oriented Computing (SOA/SOC), Bern, Switzerland*, pages 359–370, 2007.

[10] M. Jaeger, G. Rojec-Goldmann, and G. Mühl. QoS Aggregation for Web Service Composition using Workflow Patterns. In *EDOC '04: Proceedings of the Eighth IEEE International Enterprise Distributed Object Computing Conference*, pages 149–159, 2004.

[11] J. Jin, J. Liang, and K. Nahrstedt. Large-scale QoS-Aware Service-Oriented Networking with a Clustering-Based Approach. In *Proceedings of 16th International Conference on Computer Communications and Networks*, pages 522–528, 2007.

[12] M. Kim, R. Ian, B. Mckay, and D. Kim. Evolutionary Operator Self-adaptation with Diverse Operators. *Order A Journal On The Theory Of Ordered Sets And Its Applications*, pages 230–241, 2012.

[13] A. Klein, F. Ishikawa, and B. Bauer. A Probabilistic Approach to Service Selection with Conditional Contracts and Usage Patterns. In *Service-Oriented Computing*, volume 5900 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2009.

[14] A. Klein, F. Ishikawa, and S. Honiden. Efficient QoS-Aware Service Composition with a Probabilistic Service Selection Policy. In *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*, pages 182–196. Springer, 2010.

[15] A. Klein, F. Ishikawa, and S. Honiden. Efficient Heuristic Approach with Improved Time Complexity for Qos-Aware Service Composition. In *Proceedings of the 2011 IEEE International Conference on Web Services*, ICWS '11, pages 436–443, Washington, DC, USA, 2011. IEEE Computer Society.

[16] A. Klein, F. Ishikawa, and S. Honiden. Towards Network-aware Service Composition in the Cloud. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 959–968, New York, NY, USA, 2012. ACM.

[17] B. Klöpper, F. Ishikawa, and S. Honiden. Service Composition with Pareto-Optimality of Time-Dependent QoS Attributes. In *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*, 2010.

[18] F. Lécué and N. Mehandjiev. Towards Scalability of Quality Driven Semantic Web Service Composition. In *IEEE International Conference on Web Services*, July 2009.

[19] A. Lipowski and D. Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6):2193–2196, Mar 2012.

[20] D. A. Menascé, E. Casalicchio, and V. Dubey. On Optimal Service Selection in Service Oriented Architectures. *Performance Evaluation*, 67(8), 2010. Special Issue on Software and Performance.

[21] T. S. E. Ng and H. Zhang. Predicting internet network distance with coordinates-based approaches. In *INFOCOM*, 2002.

[22] J. O'Sullivan, D. Edmond, and A. Ter Hofstede. What's in a Service? *Distributed and Parallel Databases*, pages 117–133, 2002.

[23] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, and B. J. Krämer. Service-Oriented Computing: A Research Roadmap. In *Service Oriented Computing (SOC)*, Dagstuhl Seminar Proceedings, 2006.

[24] D. Pisinger. *Algorithms for Knapsack Problems*. PhD thesis, University of Copenhagen, Dept. of Computer Science, 1995.

[25] F. Rosenberg, M. B. Müller, P. Leitner, A. Michlmayr, A. Bouguettaya, and S. Dustdar. Metaheuristic Optimization of Large-Scale QoS-aware Service Compositions. *IEEE, International Conference on Services Computing*, pages 97–104, 2010.

[26] H. Topcuoglu, S. Hariri, and M. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, 2002.

[27] H. Wada, J. Suzuki, and K. Oba. Queuing theoretic and evolutionary deployment optimization with probabilistic slas for service oriented clouds. In *World Conference on Services - I*, pages 661–669, 2009.

[28] F. Wagner, F. Ishikawa, and S. Honiden. QoS-Aware Automatic Service Composition by Applying Functional Clustering. *IEEE International Conference on Web Services*, pages 89–96, 2011.

[29] J. Xiao and R. Boutaba. QoS-Aware Service Composition in Large Scale Multi-Domain Networks. In *IM 2005, 9th IFIP/IEEE International Symposium on Integrated Network Management.*, pages 397–410, May 2005.

[30] Z. Ye, X. Zhou, and A. Bouguettaya. Genetic Algorithm Based QoS-Aware Service Compositions in Cloud Computing. In *Database Systems for Advanced Applications*, pages 321–334, 2011.

[31] T. Yu, Y. Zhang, and K.-J. Lin. Efficient Algorithms for Web Services Selection with End-to-End QoS Constraints. *ACM Transactions on the Web*, 1(1), 2007.

[32] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality Driven Web Services Composition. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 411–421, 2003.

[33] Z. Zheng, Y. Zhang, and M. R. Lyu. Distributed QoS Evaluation for Real-World Web Services. pages 83–90, 2010.

Adrian Klein received his B.Sc. degree in Computer Science in 2008 from the University of Augsburg. In 2009, he received his M.Sc. degree with honours in Software Engineering from the TU Munich, the LMU Munich, and the University of Augsburg. He is currently working towards his Ph.D. degree in Computer Science at The University of Tokyo under the supervision of Shinichi Honiden and Fuyuki Ishikawa. His research interests in regard to service compositions include genetic algorithms, probabilistic QoS, efficient QoS optimization, and network-aware QoS optimization for large-scale distributed settings.

Fuyuki Ishikawa received his Ph.D. degree in Information Science and Technology from The University of Tokyo, Japan, in 2007. He is an associate professor at the National Institute of Informatics (NII), Japan. He is also a visiting associate professor at The University of Electro-Communications. His research interests include Services Computing, Cloud Computing and Software Engineering.

Shinichi Honiden received his Ph.D degree in Electrical Engineering from Waseda University, Tokyo, Japan, in 1986. From 1978 to 2000 he was working at Toshiba Corporation. Since April 2000, he has been a professor and a director of the Information Systems Architecture Research Division at the National Institute of Informatics (NII), Japan. Since 2012 he is a Deputy Director General at NII. He is also a professor in the Graduate School of Information Science and Technology at The University of Tokyo. His research interests include agent technology, pervasive computing, and software engineering.