# Efficient Heuristic Approach with Improved Time Complexity for QoS-aware Service Composition

Adrian Klein
*University of Tokyo*
*Tokyo, Japan*
*adrian@nii.ac.jp*

Fuyuki Ishikawa
*GRACE Center,*
*National Institute of Informatics*
*Tokyo, Japan*
*f-ishikawa@nii.ac.jp*

Shinichi Honiden
*University of Tokyo and*
*National Institute of Informatics*
*Tokyo, Japan*
*honiden@nii.ac.jp*

***Abstract*—Service-Oriented Architecture enables the composition of loosely coupled services provided with varying Quality of Service (QoS) levels. Given a composition, finding the set of services that optimizes some QoS attributes under given QoS constraints has been shown to be NP-hard. Therefore, heuristic algorithms are widely used, finding acceptable solutions in polynomial time. Still the time complexity of such algorithms can be prohibitive for real-time use, especially if the algorithms are required to run until they find near-optimal solutions. Thus, we propose a heuristic approach based on Hill-Climbing that makes effective use of an initial bias computed with Linear Programming, and works on a reduced search space. In our evaluation, we show that our approach finds near-optimal solutions and achieves a low time complexity.**

***Keywords*-Service-oriented Architecture, Service Composition, Quality of Service, QoS, Heuristic Algorithm, Initial Bias, Hill-Climbing, Linear Programming**

## I. Introduction

Service-Oriented Architecture (SOA) enables the composition of services in a loosely coupled way [1]. The value of SOA is achieved by enabling rapid and easy service composition with low costs.

For service compositions functional and nonfunctional requirements [2] have to be considered. The latter are specified by Quality of Service (QoS) attributes and are especially important when many functionally equivalent services are available. Such a composition should be optimal in regards to the user's QoS preferences and constraints. This composition problem (CP) has been shown to be NP-hard [3]. Therefore, an optimal solution cannot be found in polynomial time.

In order to achieve feasible solutions for the CP in acceptable time, heuristic algorithms are used, with some of the state-of-the-art algorithms being genetic algorithms [4] and Hill-Climbing [5]. The caveat of such heuristic algorithms is that while they can find feasible solutions quickly, they might not find solutions close to the optimal solution of the CP, and/or they might take a long time to find a solution. As a result, if real-time constraints are given, a good solution quality might have to be sacrificed for an acceptable performance. Accordingly, research is still ongoing today in order to improve heuristic algorithms in regards to performance and quality of the obtained solutions, as exemplified by recent works [6], [7].

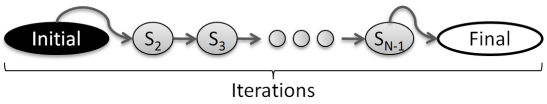



Figure 1. Heuristic Algorithm

Such heuristic algorithms usually improve a given initial solution iteratively until a satisfactory solution is obtained (Fig. 1). We will refer to the initial solution as initial bias (IB) as this is common use in optimization research. This IB is usually computed randomly, or in a very simple way to obtain a solution that is e.g. feasible or already satisfactory to some degree regarding QoS. Therefore, such an IB is usually quite different from the optimal solution. But it has been shown that an IB close to the optimal solution can positively influence the running time of a heuristic algorithm [8]. Also, in order to find near-optimal solutions, heuristic algorithms have to explore a huge search space that might grow more than linearly with the problem size, e.g. growing in a quadratic manner as in Hill-Climbing (see subsection III-D4).

Thus, we propose a heuristic approach that makes effective use of an initial bias (IB) and works on a reduced search space. First, we compute an IB that comes close to the optimal solution of the CP with an algorithm based on Linear Programming which has a low polynomial time complexity. Then, we feed this IB to our heuristic algorithm *HC** based on Hill-Climbing that explores a reduced search space. As a result, the time complexity is reduced, while still achieving near-optimal solutions. In summary, our two contributions are:

1) Efficient computation of a good IB.
2) Efficient heuristic algorithm with low time complexity achieving near-optimal solutions from a good IB.

Therefore, our paper shows that computing a good IB improves the quality of the solutions obtained by heuristic

algorithms, and furthermore allows us to reduce the size of the search space without sacrificing much of the quality of the solutions obtained, resulting in a low time complexity. This result is not limited to the specific heuristic algorithm we choose, but holds regardless of the specific heuristic algorithm; however, depending on the algorithm, the impact of a good IB can be diminished, and the reduction of the size of the search space may not be trivial.

The structure of this paper is as follows: Section II provides an overview of related work. Section III explains our approach. Section IV reports our evaluation results. Finally, section V concludes the paper.

## II. Related Work

In this section we survey our related work that can be categorized into four different types which we will describe in the following.

*Foundation:* The foundation for our research is given in [9] where the QoS-aware composition problem (CP) is introduced. Common notions, which we also use, are given, and the problem is formalized and solved with Integer Programming, which is still a common way to obtain optimal solutions for the CP.

*General Heuristic Algorithms:* One of the earliest research to use a heuristic algorithm is [10] where a genetic algorithm (GA) is used to solve the CP. They showed that using a GA scales very well with the problem size in terms of performance, not like Integer Programming which performs exponentially with the problem size. In [11] they use GA and Hill-Climbing (HC), on which our heuristic algorithm is based. Their results show that the solutions of both GA and HC do not come close to the optimal solution. In contrast, we show that we not only can achieve a good performance, but also find near-optimal solutions.

*Heuristic Algorithms using specific IBs:* Various IBs have been used with heuristic algorithms. In [12] a feasible solution is used as an IB in a heuristic algorithm that works quite similar to HC. The feasible solution is found by a simple heuristic algorithm, which itself could take a substantial amount of time compared to the main heuristic algorithm. A solution with a low execution time is used in [13] as IB for their heuristic algorithm by locally selecting services with the lowest execution time for their respective task. The standard approach of using a random solution as IB is used in [14] where stochastic HC is used. We, on the other hand, efficiently compute a more sophisticated IB with Linear Programming, which enables us to find near-optimal solutions. Also, to the best of our knowledge, no related work exists where different IBs are compared for the CP, which means that the impact of different IBs on heuristic algorithms for the CP is not well observed.

*Optimization Research:* In general optimization research, the effect of the IB has been observed in [8] on the OneMax problem with the univariate marginal distribution algorithm and the HC algorithm. They showed that the closer the IB is to the optimal solution the less time is needed to find a near-optimal solution. This work encouraged us to follow our intuition that a good IB might also be beneficial for the CP.

In conclusion, while there is general research on using good IBs to aid heuristic algorithms, there is no research regarding this for the CP. Also, to the best of our knowledge, there is no research work that reduces the size of the search space of HC for the CP. Thus, we contribute the idea to use a good IB for the CP, an actual way to compute a particular good IB, and a way to reduce the size of the search space for HC which works well in combination with a good IB.

## III. Approach

In this section, we define our approach. First, we formalize the composition problem. Then, we define the requirements for the computation of the IB and for the heuristic algorithm that facilitate our approach. After that, we show how to compute our IB. Finally, we introduce our heuristic algorithm.

### A. Composition Problem

We give a very brief definition of the composition problem, that is in line with common notations in existing research like [9], [12]. In the composition problem we first need to properly define workflows (structured compositions of services). As in [12] we support the following common composition patterns: Seq (sequential execution), AND (parallel execution), XOR (alternative execution with certain probabilities $p_i$) and Loop (with maximum loop count $x$).


Figure 2. Composition patterns: Seq, AND, XOR, Loop

For any given workflow, we aggregate the QoS according to the composition patterns. As for loops, we unroll all of them to a sequence according to their maximum loop count, so that their QoS can be aggregated. Then, the QoS of the workflow gets aggregated according to the aggregation functions presented in [15]. Given $n$ aggregated QoS attributes $q_1, \ldots, q_n$, we normalize them like in [9], limiting their values to $[0, 1]$ in order to simply maximize the weighted sum of their normalized values $\tilde{q_1}, \ldots, \tilde{q_n}$:

$$\sum_{i=1}^{n} w_i \tilde{q}_i$$

Constraints $Q_i^{min}$/$Q_i^{max}$ are applied on the aggregated non-normalized QoS of the workflow:

$$q_i \geq Q_i^{min} \wedge q_i \leq Q_i^{max}$$

### B. Requirements

We solve the given composition problem with our approach. In order to assure that our approach works, we have to meet the following three requirements:

1) The computation of the IB must be significantly faster than the heuristic algorithm itself.
2) The IB computed should be significantly better than a random IB.
3) The heuristic algorithm should make use of a good IB.

The first requirement is necessary to enable an overall performance improvement. As our goal is to speed up the heuristic algorithm, spending a similar amount of time on computing the IB would negate any positive effect achieved by using such an IB.

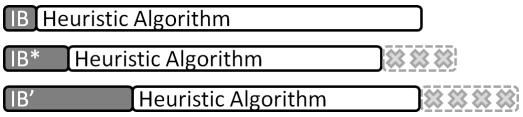



Figure 3. Effect of different IBs on the total Runtime

Thus, ideally the computation time is polynomial even in the worst-case, and is not significant compared to the runtime of the heuristic algorithm itself: e.g. like IB* in Fig. 3 that shortens the overall runtime, opposed to IB' that does not achieve any effect.

The second requirement means that while the computation of the IB must be efficient, it is worthless if the IB found is not good enough, e.g. not significantly better than a randomly chosen IB. Note that a good IB means an IB that is closer to the real optimum. Especially, because of the nature of heuristic algorithms this is actually quite a hard requirement: Heuristic algorithms have the tendency to improve on the initial IB quite quickly, using only a few iterations. Much iterations are spent to go from a good solution to a near-optimal solution, similar to the behaviour in Fig. 4 generated by ourselves. Therefore, if an IB is to improve performance significantly, it has to be a lot better than an average IB, not just a little bit better.

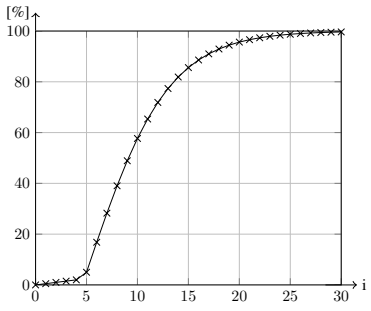



Figure 4. Converging Utility with increasing Iterations

The last requirement is that the heuristic algorithm should be able to make use of a good IB. If a good IB has no big impact on the final result, then our approach can not be effective even with a good IB. One obvious reason for such a behaviour is when in case of a genetic algorithm many random mutations are performed in the first iterations: This would mean that large parts of the IB are thrown away in the beginning so that the benefits of using the good IB are greatly diminished.

### C. Initial Bias

Now we present a way to compute a good IB that complies with the requirements given in the previous section. Our general idea is to solve a simplified version of the composition problem. If we do not simplify the problem too much, the IB computed will be very close to the optimal solution. At the same time we have to simplify the NP-hard problem enough so that in can be efficiently solved by a polynomial algorithm. In the following we will show how to achieve this balance of simplifying the problem just by the right amount.

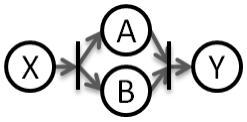



Figure 5. Sample Workflow

Thus, we apply the approach introduced in our previous research [16]. Given a workflow, like in Fig. 5, a Linear Programming (LP) problem is solved assigning many services probabilistically to each task. The LP problem is basically a relaxation of the original problem which can be formulated as an Integer Programming problem. A solution might give the following probabilities for task $X$: $90\%$ for service $S_1$ and $10\%$ for service $S_2$. In our previous research this result was meaningful in regards to multiple executions of the workflow: if the workflow is executed many times, in each execution $S_1$ or $S_2$ will be probabilistically chosen for $X$ according to the probabilities found before.

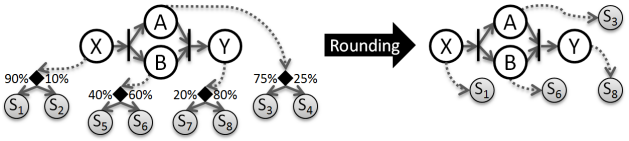



Figure 6. Rounding

But in this paper, we need a solution to the traditional composition composition problem presented before. Therefore, we round the solution obtained by [16] that assigns multiple services to each task to obtain a suitable IB that only assigns a single service to each task. For each task we select the service with the highest probability. E.g. task X would be assigned to service $S_1$, as $S_1$ has the highest probability for X in the initial LP solution (Fig. 6). We think

this IB is quite close to the optimal solution, because it is obtained by solving a slightly simplified version of the original problem. We will later show that this claim holds in our evaluation. Note that such an IB might slightly violate given QoS constraints, but those cases can be easily remedied by the heuristic algorithm. Furthermore, we have shown in [16] that the computation with Linear Programming is quite efficient and provides good scalability since efficient polynomial algorithms can be used to solve the problem. Thus, we think an IB computed with that algorithm fulfills both requirements (1) and (2).

### D. Heuristic Algorithm

After presenting our approach that computes a good IB, we now specify our heuristic algorithm. First, we describe Hill-Climbing (HC) on which we base our algorithm. Then, we specify the utility function $u$ we use to guide our algorithm. Afterwards, we introduce our algorithm *HC** that is based on HC and uses $u$. Finally, we analyze our algorithm *HC** regarding fulfillment of the requirements and regarding its time complexity.

*1) Base Algorithm:* We base our algorithm on Hill-Climbing (HC), as defined in [17], because it does achieve the requirements set before, as we will argue later. Assuming an utility function $u$ that should be maximized, Hill-Climbing starts from an initial candidate solution that is usually generated randomly. Then in each iteration all neighbour solutions of the current solution are explored, and a neighbour solution with a higher utility than the current solution is chosen. Note that a neighbour solution is a solution that only differs in a single task assignment from the current solution. The algorithm terminates when no further improvement of the current solution is possible. Regarding the choice of the neighbour solution in each step, we opted for the best improvement variant that always picks one of the solutions with the best utility.

*2) Utility Function:* The utility function $u$ used by HC influences if the algorithm can find feasible solutions at all, and how good the found solutions are. Therefore, we extend the standard utility function, previously described in subsection III-A, and define our utility function as follows:

$$u(x) = (\sum_{i=1}^{n} w_i \tilde{q}_i) - penalty(x)$$

To achieve that the algorithm finds feasible solutions, we subtract a penalty for the constraint violations from the weighted sum of the QoS, which makes up the standard utility function.

$$penalty(x) = \sum_{i=1}^{n} (isExceeded(i) \cdot exceed(i)^2)$$

The penalty is computed as the sum of the squares of the penalty factors that are computed by taking into consideration by how much the constraints are exceeded relatively to the constraint.

The $isExceeded$ function is a binary function that reflects if a constraint is violated on a particular QoS attribute:

$$isExceeded(i) = \begin{cases} 1 & \text{if constraint on } \tilde{q}_i \text{ is violated} \\ 0 & \text{otherwise} \end{cases}$$

Finally, the penalty factor is computed by taking into consideration the weight of the constrained QoS attribute and the amount by which the constraint is exceeded relatively to the value of the constraint:

$$exceed(i) = (1 + w_i)(1 + exceedAmount(\tilde{q}_i))$$

As the penalty factor is always greater than 1 and because the weighted sum of the QoS is $\in [0, 1]$, any infeasible solution (that violates some constraint) always has less utility than any feasible solution. This makes sure that the HC algorithm quickly finds a feasible solution, before looking for solutions with better QoS afterwards.

*3) Modified Algorithm:* We modify the HC algorithm in order to limit the size of the search space as follows:

**Algorithm 1:** HC*

```
Input: Limit L
Output: Solution s
s := initial candidate solution
i := 0
while i < √L do
    N := random sub ⊆ neighbourhoud of s
                with |sub| ≤ √L
    s' := random n ∈ N with max u(n)
    if u(s') ≤ u(s) then
        return s
    end
    s := s'
    i++
end
return s
```

Our modified algorithm *HC** limits the size of its search space according to the limit parameter $L$: The number of iterations is limited to $\sqrt{L}$, and the size of the neighbourhood explored in each iteration is limited to $\sqrt{L}$, as well. In the latter analysis we will show how this leads to a significant reduction of the time complexity. Note that, if we set $L$ to $\infty$, *HC** will perform identical to HC.

*4) Analysis:* In the following we analyze our algorithm *HC**. First, we show that *HC** fulfills the requirements we set before. Then, we give a semi-formal analysis of its time complexity.

*Requirements:* In order to see if our algorithm meets the requirements, we analyze its behaviour. The algorithm performs quite robustly and finds good solutions in general. In each iteration one task assignment of the solution is changed, so that the IB changes only gradually towards a local optimum. E.g. a solution that assigns services to the four tasks $T1$, $T2$, $T3$, and $T4$ of a given workflow to be optimized could change gradually during the first two iterations as shown in Fig. 7: In the first step the service $S07$ from the IB is replaced by $S05$, then in the second step $S09$ is replaced by $S11$, etc.

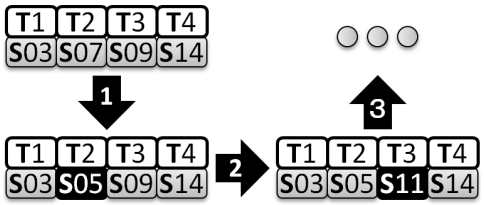



Figure 7. Iterative Changes during Execution of Hill-Climbing

Since the IB is only changed gradually, the algorithm heavily depends on its IB. The algorithm will finally find the local optimum which is reachable from the IB by taking only consecutive best improvement steps. How close the IB is to the local optimum determines how many iterations are needed for the algorithm to terminate, and how close the IB is to the global optimum determines how big the chances are that the algorithm finds a near-optimal or even optimal solution. This means that our requirement (3) is met, as the IB greatly influences the algorithm towards reaching a good solution in a short amount of time.

*Time Complexity:* Besides our functional requirements, we also want to achieve an improved time complexity for our algorithm *HC** (defined as algorithm 1 before). In the following we will refer to *HC** with a specific limit $L$ as *HC*/L*. We semi-formally show the time complexity of *HC*/*$\infty$ and *HC** with some specific limit that reduces the size of the search space significantly. As variables, we need the number of tasks a workflow can have, $T$, and the number of services that are available for each task $S$.

We first analyze *HC*/*$\infty$. The algorithm's work starts with the while loop in line 3. In each iteration one task of the workflow is assigned to a different service. Assuming a non-optimal initial candidate solution, each task's assignment should be changed a constant number of times at most, before the whole workflow's assignments cannot be improved anymore. Thus, the while loop is executed about $O(T)$ times. Then, in line 4, a neighbourhood is generated with size $O(T{\cdot}S)$, as for each tasks all services are tried to generate neighbour solutions that only differ in one task assignment. Finally, for each neighbour the workflow's utility is evaluated, which requires the aggregation of the QoS over all its tasks, thus, requiring $O(T)$. Overall, this makes $O(T^3 \cdot S)$ for *HC** with $L := \infty$. If we introduce a new variable $SF$ (scaling factor) that is the maximum of $T$ and $S$, we can simplify the result to $O(SF^4)$.

As for our heuristic approach, we choose *HC*/SF*: $\sqrt{SF}$ iterations are performed, a neighbourhood of size $\sqrt{SF}$ is explored in each iteration, and the assessing of each neighbour takes $O(SF)$. The resulting total time complexity is $O(\sqrt{SF}{\cdot}\sqrt{SF}{\cdot}SF) = O(SF^2)$. This is quite a significant reduction in time complexity by $SF^2$. In the latter evaluation we will show that this reduction holds and that the obtained solutions are still near-optimal.

## IV. Evaluation

In this section we evaluate our approach. First, we describe the setup of our evaluation. Then, we define several ways to compute an IB, including the way we proposed in subsection III-C, and show how close those different IBs are to the optimal solution. We continue with an evaluation of the traditional Hill-Climbing using the different IBs presented. Finally, we evaluate our heuristic algorithm *HC**.

### A. Setup

In the following we describe the common setup for all the following evaluations.

We generate 30 random workflows for each scaling factor $SF \in \{5, 10, 15, 20, 25, 30\}$ that determines both the size of the workflow (number of tasks) and the amount of services available for each task. Those workflows contain control structures introduced in subsection III-A which are randomly inserted into the workflows. Fig. 8 depicts an example of a generated workflow for $SF$ 10:

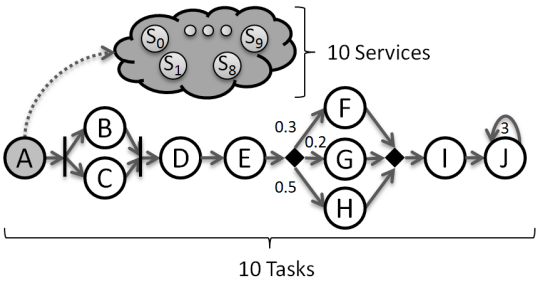



Figure 8. Generated Example Workflow

Each of those workflows is evaluated against our algorithms and IBs used with many different configurations of QoS weights and QoS constraints in order to get varied problems. Note that the weights are used for the utility function which contains a weighted sum of the QoS. The constraints are varied between loose (easily achievable) and tight (might not be achievable). Also note that we count only solvable problem instances, effectively resulting in more than 50 individual problem instances counted for each $SF$.

### B. Initial Bias

In order to evaluate if our approach for computing an IB with Linear Programming is effective, we evaluate it against other methods to compute IBs. We define some common and promising methods that adhere to our previous requirements in the following.

One of the most basic ways to compute an IB is to generate an IB by randomly choosing one service for each task with equal probability out of all the services available:

**Algorithm 2:** Random (R)

**Input**: Repetitions $n$
**do** $n$ times
  generate random solution $s$
**end**
pick solution $s$ with **max** $u(s)$

This does not make use of any of the given problem parameters, like QoS weights, constraints, etc. As it is quite cheap to randomly generate a solution, we can easily afford to repeat the process a fixed number of times, and to pick the best solution found. Thus, for our evaluation, we choose to repeat the generation 5 times, as it gave us good results, and still can be computed fast enough.

Another basic way is to make use of the given QoS weights by locally choosing the services which score best according to those weights, which are used in the weighted sum of the utility function $u$:

**Algorithm 3:** Greedy Weights (GW)

$s :=$ empty solution
**foreach** *task* $t \in s$ **do**
  set service $x$ on solution $s$ for task $t$ with **max** $u(x)$
**end**
pick solution $s$

This local heuristic takes into account neither the global QoS, nor the QoS constraints. As such, it might very well fail to fulfill the given constraints.

In contrast to this, a basic heuristic that does take the QoS constraints into account can be defined as follows:

**Algorithm 4:** Greedy Bounds (GB)

$s :=$ empty solution
**foreach** *task* $t \in s$ **do**
  set service $x$ on solution $s$ for task $t$
    with **max** distance to constraints of $s$ with $x$
**end**
pick solution $s$

This heuristic maximizes the margin towards the given QoS constraints regarding the workflow constructed so far in the algorithm. That means that the heuristic takes into account more than just local information, but it does this in a greedy manner.

Finally, we also give a definition of our algorithm that uses Linear Programming, as described in subsection III-C:

**Algorithm 5:** Linear Programming (LP)

$s :=$ empty solution
$lp :=$ compute solution with Linear Programming
**foreach** *task* $t \in s$ **do**
  pick service $x$ of solution $lp$ for task $t$
    with **max** probability
  set service $x$ on solution $s$ for task $t$
**end**
pick solution $s$

By picking the service with the highest probability for each task a kind of rounding is performed. This algorithm performs a global optimization on a simplified problem taking into account both QoS weights and constraints.

In the following and especially in the diagrams, we will refer to this four ways to compute an IB by their respective shortcuts R, GW, GB, and LP.

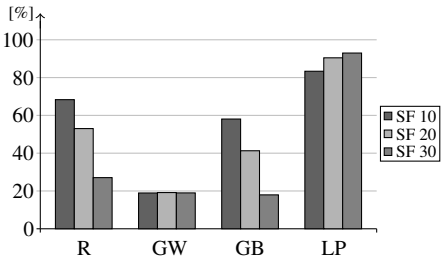
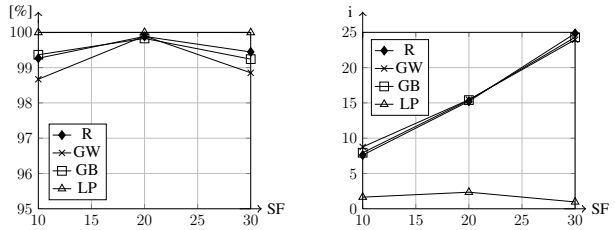



Figure 9. Average Utility of IBs for the $SF$s

As we already mentioned, according to [8], an IB closer to the optimal solution should lead to the benefits we want to achieve. Thus, we evaluate how close our chosen IBs are to the optimal solution for the $SF$s 10, 20, and 30. As we can see in Fig. 9, for $SF$ 10 both R and GB still find reasonably good IBs with over 50% utility, but for increasing $SF$s the utility decreases sharply, while GW always seems to find IBs with a utility of just about 20%. On the contrary LP achieves a utility of over 80% for $SF$ 10, and with increasing $SF$ this utility comes even closer to optimal solution, clearly exceeding 90%.

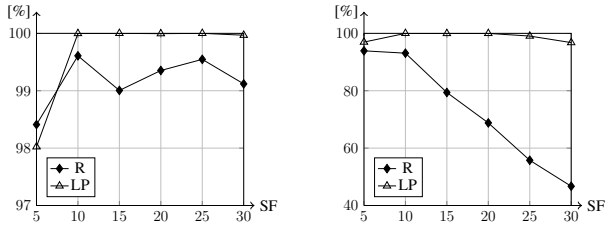
### C. Hill-Climbing

Given these four ways to compute IBs, we now evaluate traditional Hill-Climbing (HC) with those IBs for the $SF$s 10, 20 and 30. Fig. 10(a) shows that all IBs achieve quite good utility when used as an input for HC. Additionally, while GW and GB show no improvement over R, LP is slightly better for all $SF$s. In line with our analysis in subsection III-D4, the number of iterations increases linearly




(a) Utility (b) Iterations

Figure 10. HC with respective IBs and $SF$s




(a) HC (b) *HC*/SF*

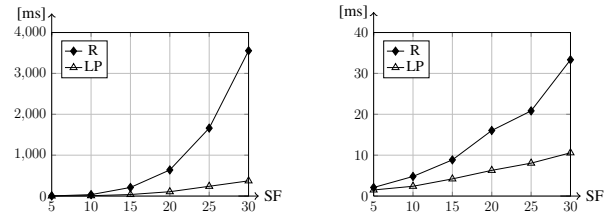
Figure 12. Average Utility of HC and *HC*/SF* for the $SF$s

with the $SF$ for R, GW and GB, as Fig. 10(b) shows. Only LP seems to need just a constant number of iterations. This means that using LP for HC seems to improve the time complexity by a factor $SF$ from $O(SF^4)$ to $O(SF^3)$.

We conclude about HC that using GW and GB does not provide any benefit over standard R, so R is a robust and good choice compared to IBs computed by simple heuristics. Regarding our LP, using it improves the time complexity, and achieves near-optimal solutions.

### D. *Our Algorithm* HC*/SF

Finally, we evaluate our algorithm *HC*/SF* (= *HC** with the limit set to the scaling factor $SF$). Regarding the IBs presented, we will only evaluate the standard R and our LP in the following, as the other IBs have resulted in no benefit over R. We will compare *HC*/SF*'s utility and time complexity against the standard HC algorithm with the $SF$s 5, 10, 15, 20, 25 and 30.




(a) HC (b) *HC*/SF*

Figure 11. Runtime of HC and *HC*/SF* for the $SF$s

The runtime depicted in Fig. 11 confirms the time complexity predicted: While HC with R performs with $O(SF^4)$, *HC*/SF* just needs $O(SF^2)$. As we already observed in the previous section, the number of iterations with LP becomes $O(1)$, which is a decrease from the $O(\sqrt{SF})$ iterations *HC*/SF* needs normally. So HC with LP seems to take $O(SF^3)$ time, as we already argued before, and *HC*/SF* with LP seems to take only $O(1 \cdot \sqrt{SF} \cdot SF) = O(SF^{1.5})$ time. This is quite a big improvement of $SF^{2.5}$ over the original HC algorithm using R, which clearly shows in the runtime improvements: for instance instead of over 3000ms for $SF$ 30, just about 10ms are needed to compute a solution with *HC*/SF*.

As we already saw in subsection IV-C, the average utility of solutions achieved by HC is quite similar with R and with LP, even if the utility with LP is slightly better (Fig. 12(a)). *HC*/SF*'s utility shows a quite different behaviour according to Fig. 12(b) though: The average solution utility of *HC*/SF* with R decreases sharply with an increasing $SF$, for instance already achieving less than $50\%$ for $SF$ 30. On the contrary, *HC*/SF* achieves near-optimal utility with LP independent from the $SF$, constantly achieving at least $96\%$, which comes quite close to the optimal solution.

We conclude that while *HC*/SF*'s runtime is good independent of the IB used, the utility of the solutions achieved with R is insufficient for practical use. This means that a reduction of the size of the search space for HC is only possible with a good IB. Thus, *HC*/SF* with our LP is the only viable combination as even while greatly reducing the size of the search space, the algorithm still manages to achieve near-optimal solutions.

## V. Conclusion

In this paper we introduced a heuristic approach that makes effective use of an initial bias (IB). We computed a good IB by using Linear Programming (LP), and used that IB as an initial candidate solution for our heuristic algorithm. As a heuristic algorithm we introduced *HC** which is based on Hill-Climbing (HC), but reduces the size of the search space explored. In our evaluation, we compared our IB computed with LP to using IBs computed randomly, or by greedily optimizing according to QoS preferences or QoS constraints. We showed that while the other IBs do not improve our heuristic algorithm significantly, our IB improves not only the utility of the solutions achieved, but also the runtime. Furthermore, we showed that only our IB enables us to use *HC** with a greatly reduced search space, while still finding near-optimal solutions. Thus, we showed that our approach produces near-optimal solutions in just a fraction of the time required for the standard HC algorithm. We also showed in our analysis and our evaluation that our approach has a much lower time complexity than the standard HC algorithm. Therefore, we think that using a good IB and reducing the size of the search space is crucial when using a heuristic algorithm.

While we strongly believe that our results can be generalized for many heuristic algorithms, the amount of improvement regarding both utility and time complexity will vary, of course. That is, why we think that it would be quite interesting to try our IB on a genetic algorithm, try to tweak the size of its search space, and then compare the results with our existing results. Especially for a genetic algorithm the tweaking of the algorithm could be crucial in order to profit as much as possible from a good IB, as there is mutation involved, which could diminish the effects of a good IB if used carelessly. Also with a genetic algorithm it would be possible to use even more information from the IB obtained by LP, if all the probabilities of the LP solution can be used in a meaningful way, instead of just rounding everything.

## VI. Acknowledgments

We would like to thank Florian Wagner for the fruitful discussions and the detailed feedback that helped us to improve our approach.

## References

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

[2] J. O'Sullivan, D. Edmond, and A. Ter Hofstede, "What's in a Service?" *Distributed and Parallel Databases*, vol. 12, no. 2–3, pp. 117–133, 2002.

[3] D. Pisinger, "Algorithms for Knapsack Problems," Ph.D. dissertation, University of Copenhagen, Dept. of Computer Science, 1995.

[4] C. Zhang, S. Su, and J. Chen, "DiGA: Population diversity handling genetic algorithm for QoS-aware web services selection," *Computer Communications*, vol. 30, no. 5, pp. 1082–1090, 2007.

[5] V. K. Dubey and D. a. Menascé, "Utility-Based Optimal Service Selection for Business Processes in Service Oriented Architectures," *2010 IEEE International Conference on Web Services*, pp. 542–550, Jul. 2010.

[6] 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*, 2009, pp. 881–890.

[7] 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*, 2010, pp. 11–20.

[8] M. Pelikan and K. Sastry, "Initial-population bias in the univariate estimation of distribution algorithm," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, no. 2009001. ACM Press, 2009, p. 429.

[9] 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*, 2003, pp. 411–421.

[10] 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*, 2005, pp. 1069–1075.

[11] 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*, 2007, pp. 359–370.

[12] 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*, vol. 1, no. 1, p. 6, 2007.

[13] D. A. Menascé, E. Casalicchio, and V. Dubey, "On optimal service selection in Service Oriented Architectures," *Performance Evaluation*, vol. 67, no. 8, pp. 659–675, 2010.

[14] F. Lecue and N. Mehandjiev, "Towards Scalability of Quality Driven Semantic Web Service Composition," in *ICWS '09: IEEE International Conference on Web Services*, 2009, pp. 469–476.

[15] 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*, 2004, pp. 149–159.

[16] A. Klein, F. Ishikawa, and S. Honiden, "Efficient QoS-Aware Service Composition with a Probabilistic Service Selection Policy," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, vol. 6470, 2010, pp. 182–196.

[17] H. H. Hoos and T. Stützle, *Stochastic Local Search : Foundations & Applications*. Morgan Kaufmann, 2005.