A Divide and Conquer Approach for Web Services Location Allocation Problem

Harshal Tupsamudre∗
harshalcoep@gmail.com
Saket Saurabh†
sk.saurabh@tcs.com
Arun Ramamurthy†
raramamurthy.arun@tcs.com
Mangesh Gharote†
mangesh.g@tcs.com
Sachin Lodha†
sachin.lodha@tcs.com

ABSTRACT
The appropriate choice of locations for the deployment of web services is of significant importance. The placement of a web service closer to user centers minimizes the response time, however deployment cost may increase. The placement becomes more challenging when multiple web services are involved. In this paper, we address the problem of placing multiple web services with the aim of simultaneously minimizing conflicting objectives of total deployment cost and network latency. We solve the location allocation problem for each web service independently and combine the resulting solutions using a novel merge algorithm. We demonstrate through extensive experiments and simulations that the proposed approach is not only computationally efficient but also produces good quality solutions. Further, the proposed merge algorithm is generic and could be easily adapted to tackle any bi-objective optimization problem that can be decomposed into non-overlapping sub-problems.

KEYWORDS
Web services; Multi-Objective Optimization; Location Allocation Problem; Divide and Conquer

1 INTRODUCTION
Service-oriented computing using web services has emerged as a new computing paradigm for developing software applications. These web services provide a distributed computing infrastructure for both intra- and cross-enterprise application integration and collaboration. As the number of functionally similar web services is of significant importance. The placement of a web service closer to user centers minimizes the response time, however deployment cost may increase. The placement becomes more challenging when multiple web services are involved. In this paper, we address the problem of placing multiple web services with the aim of simultaneously minimizing conflicting objectives of total deployment cost and network latency. We solve the location allocation problem for each web service independently and combine the resulting solutions using a novel merge algorithm. We demonstrate through extensive experiments and simulations that the proposed approach is not only computationally efficient but also produces good quality solutions. Further, the proposed merge algorithm is generic and could be easily adapted to tackle any bi-objective optimization problem that can be decomposed into non-overlapping sub-problems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

∗Work was done as a part of TCS Research and Innovation
†TCS Research and Innovation, Tata Consultancy Services, India

WSLAP is a variant of a facility location problem which is proved to be NP-hard [4] [5]. The search space of WSLAP is combinatorial. If there are, say, s web services and n candidate locations, and each service can be deployed at multiple locations, then the number of solutions is 2sn [3]. To find multiple Pareto-optimal solutions using exact approaches such as Integer Linear Programming (ILP), branch and bound algorithm etc., is challenging.

Therefore, researchers have explored various Multi-Objective Evolutionary Algorithms (MOEAs) such as MOPSO [6], NSGAII [7] and NSGA-II with a local search [8] in WSLAP. As MOEAs work with a population of solutions, they can produce a set of trade-off solutions in a few generations. Further, MOEAs scale much better than ILPs due to the nature of heuristic search. Recently, Tan et al. [3] proposed a Binary Multi-Objective Particle Swarm Optimization with Crowding Distance (BMOPSOCD) algorithm to solve WSLAP and showed that it produces better solutions than NSGAIL. Many a time, decomposing the problem into smaller sub-problems, solving each sub-problem independently and merging is an efficient way to solve the problem [9]. The main contribution of this work is a novel merge algorithm to solve the WSLAP using a Divide and Conquer (D&C) approach. Particularly, we solve the location allocation problem for each web service using a MOEA (NSGAII or BMOPSOCD) and efficiently merge the resulting non-dominated solutions of each web service to obtain solutions for WSLAP. The major benefits of D&C are:

• Problem size reduces significantly.
• Sub-problems being independent can be solved in parallel.
• Sub-problems that are similar (in terms of configuration and requirements) need to be solved only once.

Further, the approach is incremental as it enables web service providers to find deployment locations for a new web service and
perform allocation of user centers without having to solve the entire problem from scratch.

The merge algorithm takes non-dominated solutions of multiple sub-problems (web services) and produces the solution to the original problem (WSLAP). We conducted multiple experiments with different WSLAP instance sizes to demonstrate that the D&C approach is not only efficient but also produces better solutions in terms of quality as well as diversity than the state-of-the-art [3].

The organization of this paper is as follows. First, we briefly describe the multi-objective WSLAP model as given by Tan et al. [3]. Subsequently, we explain our D&C algorithm in detail. Later, we compare the efficacy of our D&C approach with the state-of-the-art MOEAs [3] on the original WSLAP model. Finally, we conclude the paper and discuss the future work.

2 WSLAP DESCRIPTION

The location allocation problem involving multiple web services [3] is described as follows. A set of s web services \( W = \{W_1, W_2, \ldots, W_s\} \) needs to be deployed at one or more of the n candidate locations \( A = \{A_1, A_2, \ldots, A_n\} \). The locations could be the data centers of the enterprise itself and/or one or more cloud providers. The cost incurred to deploy a web service \( W_i \) at a location \( A_j \) is given by \( c_{ij} \).

The deployment cost varies from one data center to another due to the difference in electricity price, real estate price, human labour cost etc. All web services are assumed to be independent of each other. If that is not the case, we can merge dependent services into a single web service.

There are \( m \) user centers \( U = \{U_1, U_2, \ldots, U_m\} \) that require access to each of the s web services. A user center represents a geographic region. It also allows us to estimate the latency between the user center and candidate locations. The service invocation frequency (demand) of a web service \( W_i \) from a user center \( U_k \) over a unit time interval is represented by \( f_{ki} \). The service frequencies are computed as the average number of invocations over a period of time, such as over a month. The network latency between a user center \( U_k \) and a location \( A_j \) is given by \( l_{kj} \). Each service has to be deployed in at least one location. Further, a service can be deployed at multiple locations to improve its response time. Note that the requirement of a user center pertaining to a given web service is catered by exactly one location. The matrices required for modelling the input in WSLAP are given in Table 1.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Entry</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( W_{s \times 1} )</td>
<td>( W_i )</td>
<td>( i^{th} ) web service</td>
</tr>
<tr>
<td>( A_{n \times 1} )</td>
<td>( A_j )</td>
<td>( j^{th} ) location</td>
</tr>
<tr>
<td>( U_{m \times 1} )</td>
<td>( U_k )</td>
<td>( k^{th} ) user center</td>
</tr>
<tr>
<td>( F_{s \times n} )</td>
<td>( F_{ki} )</td>
<td>Frequency invocation of service ( W_i ) at ( U_k )</td>
</tr>
<tr>
<td>( C_{s \times n} )</td>
<td>( C_{ij} )</td>
<td>Cost of deploying service ( W_i ) at ( A_j )</td>
</tr>
<tr>
<td>( L_{n \times n} )</td>
<td>( L_{kj} )</td>
<td>Latency between ( U_k ) and ( A_j )</td>
</tr>
<tr>
<td>( X_{s \times n} )</td>
<td>( X_{ij} )</td>
<td>Decision variable indicating whether ( W_i ) is deployed at location ( A_j )</td>
</tr>
</tbody>
</table>

The mathematical model of WSLAP is as follows:

\[
\text{minimize } f_1 = \sum_{i=1}^{s} \sum_{j=1}^{n} C_{ij} \cdot X_{ij} \quad (1)
\]

\[
\text{subject to: } \sum_{j=1}^{n} X_{ij} \geq 1 \quad \forall i \in \{1, \ldots, s\} \quad (3)
\]

\[
X_{ij} \in \{0, 1\} \quad \forall i \in \{1, \ldots, s\} \quad \forall j \in \{1, \ldots, n\} \quad (4)
\]

The objective of WSLAP is to locate a set of web services \( W \) among a set of candidate locations \( A \) such that the total deployment cost \( f_1 \) and the total response time \( f_2 \) are minimized. A service location matrix \( X \) of size \( s \times n \) is used to represent the entire location plan. The value of the decision variable \( X_{ij} \) is 1 if web service \( W_i \) is deployed at location \( A_j \); otherwise, it is 0. A user center is served from the nearest web service location among the deployed locations to minimize latency.

To minimize latency, each user center is served from the nearest web service location among the set of deployed locations. \( R_{ki} \) represents the minimal response time incurred when web service \( W_i \) is accessed by user center \( U_k \) and is calculated as follows.

\[
R_{ki} = \min \{l_{kj} | X_{ij} = 1 \text{ and } j \in \{1, \ldots, n\}\} \quad (5)
\]

As the dimensions of the service location matrix \( X \) is \( s \times n \) and each entry takes a binary value 0 or 1, the size of the search space for WSLAP is \( 2^{s \times n} \).

A solution of WSLAP is a tuple \((a, b)\), where \( a \) represents the total deployment cost \((f_1)\) and \( b \) represents the total latency \((f_2)\). A solution \( \theta_i = (a_i, b_i) \) is said to dominate solution \( \theta_j = (a_j, b_j) \), if \( \theta_i \) is at least as good as \( \theta_j \) in one objective and strictly better in the other objective.

- \( a_i \leq a_j \) and \( b_i \leq b_j \) or
- \( a_i \leq a_j \) and \( b_i < b_j \)

A solution \( \theta^* \) is referred to as Pareto-optimal if it is not dominated by any other solution. A multi-objective optimization problem typically has many Pareto-optimal (non-dominated) solutions.

**Locations**: In the original problem, a web service can be deployed in at most \( n \) locations. If the candidate locations are cloud regions, then the value of \( n \) is typically 50 (e.g., Microsoft Azure). If multiple cloud providers are considered, then the value of \( n \) easily exceeds 100. Solving WSLAP could generate many solutions where each service is deployed in a large number of locations. In practice, a web service provider might consider deploying its web service \( W_i \) in at most \( d_i \) locations \((d_i << n)\). Therefore, to accommodate this upper bound \( d_i \), we change the WSLAP constraint given in equation (3) as follows:

\[
1 \leq \sum_{j=1}^{n} X_{ij} \leq d_i \quad d_i \leq n \quad \forall i \in \{1, \ldots, s\} \quad (6)
\]

If we set \( d_i \) to be \( n \), then the problem becomes equivalent to the original WSLAP.

3 OUR APPROACH

We show that WSLAP with the aforementioned characteristics can be tackled efficiently using a D&C approach. The steps involved in our D&C approach is as follows:

1. Decompose WSLAP into \( s \) subproblems, one for each web service.
2. Solve each subproblem independently using MOEAs.
3. Merge solutions of all subproblems.

In the rest of the section, first, we prove that WSLAP can be decomposed into s sub-problems, where s is the number of unique web services, and each sub-problem can be solved independently. Later, we give a novel space and time efficient merge algorithm for combining non-dominated solutions of each sub-problem to obtain non-dominated solutions for the original WSLAP problem.

**Theorem 1.** The multi-objective service location allocation problem (WSLAP) comprising of s web services can be decomposed into s sub-problems, one corresponding to each web service.

We can re-write the first objective of WSLAP given in equations (1) as follows:

\[
    f_1 = \left( \sum_{j=1}^{n} C_{ij} \cdot X_{ij} \right) + \left( \sum_{j=1}^{n} C_{j2} \cdot X_{j2} \right) + \ldots + \left( \sum_{j=1}^{n} C_{xn} \cdot X_{xn} \right)
\]

where, \( f_1 \) denotes the deployment cost of service \( W_i \).

Similarly, we can re-write the second objective as follows:

\[
    f_3 = \left( \sum_{j=1}^{m} R_{kj} \cdot F_{kj} \right) + \left( \sum_{j=1}^{m} R_{k2} \cdot F_{k2} \right) + \ldots + \left( \sum_{j=1}^{m} R_{kn} \cdot F_{kn} \right)
\]

where, \( f_3 \) denotes the latency of service \( W_i \).

Grouping the corresponding terms in equations (9), (12) and (13), we can model and solve the location allocation problem for each web service independently.

**Corollary 1.** If a solution \((f_1, f_3)\) for web service \( W_i \) is not Pareto-optimal then the corresponding solution \((f_1, f_3)\) to the original problem WSLAP is also not Pareto-optimal.

By Theorem 1, the solution \((f_1, f_3)\) to WSLAP is computed using solution obtained for each web service. Therefore, we have:

\[
    f_1 = f_{11} + f_{12} + \ldots + f_{1i} + \ldots + f_{1s} \\
    f_3 = f_{31} + f_{32} + \ldots + f_{3i} + \ldots + f_{3s}
\]

Since the solution \((f_1, f_3)\) corresponding to web service \( W_i \) is not Pareto-optimal, there exists a better solution \((f_1^*, f_3^*)\) which dominates it. Assume that \( f_{1i} \leq f_{1} \) and \( f_{3i} < f_{3} \). Therefore, the new solution \((f_1^*, f_3^*)\) to the original WSLAP problem can be computed as follows:

\[
    f_1' = f_{11} + f_{12} + \ldots + f_{1i} + \ldots + f_{1s} \leq f_1 \\quad f_3' = f_{31} + f_{32} + \ldots + f_{3i} + \ldots + f_{3s} < f_3
\]

As \((f_1, f_3)\) is dominated by \((f_1^*, f_3^*)\), it cannot be Pareto-optimal. Note: Similar argument can be made for the case of \( f_{1i} < f_{1} \) and \( f_{3i} \leq f_{3} \).

**Corollary 2.** If the relative invocation frequency and the deployment cost of a web service \( W_i \) is the same as that of another web service \( W_{i'} \), then the solution for \( W_{i'} \) can be obtained using solutions for \( W_i \) and vice versa.

Consider the relative invocation frequency of service \( W_i \) for \( m \) user centers, that is:

\[
    F_i = \sum_{k=1}^{m} F_{ki} = \sum_{k=1}^{m} F_{k1} = \cdots = \sum_{k=1}^{m} F_{kn}
\]

Similarly, consider the relative invocation frequency of service \( W_{i'} \):

\[
    F_{i'} = \sum_{k=1}^{m} F_{k'i'} = \sum_{k=1}^{m} F_{k'i1} = \cdots = \sum_{k=1}^{m} F_{kn}
\]

We are given that the relative invocation frequency and cost of both services are the same. Hence, we have:

\[
    F_{ki} = \sum_{k=1}^{m} F_{k'i} = \sum_{k=1}^{m} F_{k'1} = \cdots = \sum_{k=1}^{m} F_{kn}
\]

Let \( \sum_{k=1}^{m} F_{ki} = z \cdot \sum_{k=1}^{m} F_{kg} \)

Therefore, we have:

\[
    F_{ki} = z \cdot F_{kg} \quad \forall k \in \{1, \ldots, m\}
\]

Let \( \sum_{k=1}^{m} F_{ki} = z \cdot \sum_{k=1}^{m} F_{kg} \)

Therefore, there exists a solution \((f_{1k}, f_{3k})/z\) for \( W_{i'} \) that dominates \((f_{1k}, f_{3k})\) hence, the contradiction.

Once a web service \( W_i \) is located, allocation of user centers is relatively easy. Each user center is allocated to the nearest web service location. Therefore, search space for location allocation of a web service is \( \sum_{i=1}^{d_{i}} \binom{n}{i} \). After obtaining non-dominated solutions to each web service, we need to combine them to find the non-dominated solutions for the original problem. Let the set of non-dominated solutions for each web service \( W_i \) be denoted by \( S_i \). Therefore, the total solution space of WSLAP is:

\[
    \text{search space} = \sum_{i=1}^{d_{i}} \binom{n}{i} + \sum_{i=1}^{s} |S_i|
\]
web service, find non-dominated solutions to each sub-problem (web service) and finally merge the solutions of all sub-problems to obtain solutions to the original problem (WSLAP). The major benefits of using D&C approach for WSLAP are as follows:

1. **Efficiency**: Each sub-problem is relatively smaller and independent, hence can be solved in parallel. The existing approaches proposed in the literature [6][7][3] do not exploit these characteristics and attempt to solve the problem in its entirety.

2. **Incremental**: If a new web service is added to the existing problem, then the previously proposed techniques [6][7][3] need to resolve the entire problem from scratch. With the D&C approach, we only need to find location-allocation for the new web service and merge the resulting solutions with the existing solutions. Thus, the proposed approach is incremental.

3. **Reuse**: In some applications, it is likely that the invocation frequency and deployment cost of few web services are the same. With the D&C approach, we only need to find solutions for one such web service (Corollary 2).

The pseudocode of our D&C approach is depicted in Algorithm 1. First, we obtain a set of non-dominated solutions $S_i$ for each web service $W_i$ using evolutionary algorithms described in the next section (lines 4-6). The resulting sets of non-dominated solutions $(S_1, S_2, \ldots, S_k)$ are then combined using the Merge procedure (line 7). The merge algorithm is iterative and starts by combining non-dominated solutions of the first two web services which are then combined with non-dominated solutions of the third web service and so on (lines 13-16). The Combine procedure consists of three main steps. First, it adds non-dominated solutions of two sub-problems $E_1$ and $E_2$ in all possible ways (lines 23-29). The resulting set of solutions $E = \{(a+c,b+d) | (a,b) \in E_1 \text{ and } (c,d) \in E_2\}$ is of size $|E_1| \cdot |E_2|$. Second, it sorts set $E$ by the first objective in ascending order. Third, it uses Kung’s et al. Front method [10] which is a recursive procedure to find non-dominated solutions from the input set $E$.

**Theorem 2.** The Merge procedure takes as input a list of non-dominated solution sets $S_1, S_2, \ldots, S_k$, where $S_i$ is a solution set for $i^{th}$ sub-problem, and computes the set of non-dominated solutions $S$ to the original problem.

We prove the correctness of Merge procedure using a loop invariant [11]. The loop invariant is at the start of $i^{th}$ iteration where set $S$ contains non-dominated solutions to the first $i-1$ sub-problems. **Initialization.** Before the loop starts, set $S$ is initialized to the set $S_1$ of non-dominated solutions for the first sub-problem (line 13). Hence, at the start of the loop, when the value of $i$ is 2, $S = S_1$. Therefore, the loop invariant initially holds.

**Maintenance.** Suppose that the invariant holds at the beginning of iteration $i$, that is, $S$ contains non-dominated solutions to the first $i-1$ sub-problems. The Combine procedure is called to merge solutions of the first $i-1$ sub-problems with solution set $S_i$ of the $i^{th}$ sub-problem (line 15). The procedure begins by adding non-dominated solutions in two sets $S$ and $S_j$ in all possible ways (lines 23-29) and creates a new set $E = \{(a+c,b+d) | (a,b) \in S \text{ and } (c,d) \in S_j\}$. Then, it sorts set $E$ by first objective in ascending order (line 30) and calls Kung’s et al. Front method [10] to find the set of non-dominated solutions $N$ in $E$ (line 31). The set $N$ is returned and assigned to $S$. The set $S$ now contains non-dominated solutions to the first $i$ sub-problems and, therefore, the invariant holds at the end of the iteration as well.

**Termination.** The loop terminates when $i = s + 1$. Thus, from the invariant, $S$ contain non-dominated solutions to all $s$ sub-problems and hence to the original problem.

We note that the Combine procedure given in Algorithm 1 requires $O(|E_1| \cdot |E_2|)$ space to store all possible solutions (lines 24-29) and $O(|E_1| \cdot |E_2| \cdot \log(|E_1| \cdot |E_2|))$ time for sorting all possible solutions (line 30). However, the number of non-dominated solutions could be much less than $|E_1| \cdot |E_2|$. In the next section, we give a space and time efficient method to combine non-dominated solutions of two sub-problems.

**3.1 Novel Combine Algorithm**

Consider two sub-problems $P_1$ and $P_2$. Suppose that $P_1$ has $r$ unique non-dominated solutions $E_1 = \{(a_1, b_1), \ldots, (a_r, b_r)\}$ and $P_2$ has $t$ unique non-dominated solutions $E_2 = \{(c_1, d_1), \ldots, (c_t, d_t)\}$, where $r \leq t$. We consider only unique solutions to avoid redundant computations. Further, assume that each solution set is sorted in ascending order by first objective $i.e., a_1 < a_2 < \ldots < a_r$ and $c_1 < c_2 < \ldots < c_t$. Adding solutions in sets $E_1$ and $E_2$ in all possible ways yields a solution set $E$ consisting of $r \cdot t$ tuples as shown in Figure 1.

**Theorem 3.** Consider a solution tuple $(a_i + c_j, b_i + d_j) \in E$. It cannot dominate or be dominated by tuples $(a_k + c_j, b_k + d_j)$, where $(k \leq i \text{ and } l \leq j)$ or $(k \geq i \text{ and } l \geq j)$.

Since all solutions in $E_1$ are unique, non-dominated and arranged in increasing order of the first objective $f_1$, they must be in decreasing order of the second objective $f_2$. Similar is the case with solutions in $E_2$.

Since $a_k < a_i$ and $b_j > b_i$ for $k < i$, and $c_l < c_j$ and $d_j > d_j$ for $l < j$.

---

**Algorithm 1 D&C Algorithm**

1. procedure DivideAndConquer
2. Input: Frequency matrix $F_{max}$, Cost matrix $C_{max}$ and Latency matrix $L_{max}$
3. Output: A set of non-dominated solutions $S$ to WSLAP
4. for $i = 1$ to $s$ do
5. $S_i = \text{Solve}(F[1 \cdot m][i], C[i], L)$
6. end for
7. Merge($S_1, S_2, \ldots, S_s$)
8. end procedure
9. procedure Merge
10. Input: A set of non-dominated solutions $S_i$ for each of the $s$ web services
11. Output: A set of non-dominated solutions $S$ to WSLAP
12. $S = S_1$
13. for $i = 2$ to $s$ do
14. $S = \text{Combine}(S, S_i)$
15. end for
16. return $S$
17. end procedure
18. procedure Combine
19. Input: Two sets of non-dominated solutions $E_1$ and $E_2$
20. Output: A set of non-dominated solutions in the combined set $E = \{(a + c, b + d) | (a,b) \in E_1 \text{ and } (c,d) \in E_2\}$
21. $E = \emptyset$
22. for $x = 1$ to $|E_1|$ do
23. for $y = 1$ to $|E_2|$ do
24. $s = (a_x + c_y, b_x + d_y)$
25. if $(a_x + c_y, b_x + d_y) \in E_1 \times E_2$ then
26. $E \leftarrow E \cup \{(a_x + c_y, b_x + d_y)\}$
27. end if
28. end for
29. end for
30. sort tuples in $E$ by first objective in ascending order
31. return $\text{Front}(E)$
32. end procedure

---

1349
Algorithm 2 which generates \( d_l \) or \( (a, b) \), the row index of recently removed tuple (from min-heap) for each \( f_l \) when \( \text{fla} \) is inspected. Similarly, to generate the solution in the row \( j \) and column \( i \) of the tuples dominated by \( a \) and \( b \), we need to check if the solution left to it (column \( i - 1 \)) is already removed from the min-heap (lines 17-19). For this, the \((i+1)\)th entry in \( \text{flag} \) array is inspected. Similarly, to generate the solution in the row \( j \) and column \( i \), we need to check if the solution above it (row \( j - 1 \)) is already removed from the min-heap (lines 20-23). For this, the \((i-1)\)th entry in \( \text{flag} \) array is inspected. This entire process repeats until all \( r \times t \) solutions are generated. In the end, the stack \( N \) contains the set of non-dominated solutions to the combined problem which is returned to the calling function (line 24).

Figure 1: Two sets \( E_1 \) and \( E_2 \) of non-dominated solutions containing \( r \) and \( t \) tuples respectively, and their combination \( E \) consisting of \( r \times t \) candidate solutions.

\[
l < j, \text{ we have} \\
ak + cl < a_i + c_j, b_k + dl > b_i + d_j \quad (15)
\]

The above inequality holds even when \( k = i \) (and \( l < j \)) or \( l = j \) (and \( k < i \)). Hence, \( (a_1 + c_j, b_1 + d_j) \) cannot dominate or be dominated by \( (a_k + c_i, b_k + d_k) \), where \( k \leq i \) and \( l \leq j \).

Similarly, \( a_i > a_j \) and \( b_k < b_i \) for \( k > i \), and \( c_i > c_j \) and \( d_k < d_j \) for \( l > j \). Hence, we have

\[
ak + c_i > a_i + c_j, b_k + d_i < b_i + d_j \quad (16)
\]

The above inequality holds even when \( k = i \) (and \( l > j \)) or \( l = j \) (and \( k > i \)). Hence, \( (a_1 + c_j, b_1 + d_j) \) cannot dominate or be dominated by \( (a_k + c_i, b_k + d_i) \), where \( k \geq i \) and \( l \geq j \). Thus, the proof.

Consider a solution \( (a_3 + c_2, b_3 + d_2) \in E \) which is highlighted in green in Figure 1c. The theorem implies that all solution tuples that lie to the top-left (shaded in cyan) of \((a_3 + c_2, b_3 + d_3)\) are better in the first objective but worse in the second objective. Similarly, all solution tuples that lie to the bottom right (shaded in yellow) are better in the second objective but worse in the first objective. Therefore, \((a_3 + c_2, b_3 + d_3)\) need not be compared with these solutions.

Based on this observation, we propose an iterative procedure CombineEff (Algorithm 2) which generates \( |E| = r \times t \) solution tuples strategically to avoid unnecessary comparisons and returns a set of non-dominated solutions \( N \subseteq E \). The strategy is to generate a solution tuple \((a_1 + c_j, b_1 + d_1)\) only if the left tuple \((a_{i-1} + c_j, b_{i-1} + d_1)\) and the top tuple \((a_1 + c_{j-1} + b_{1} + d_{j-1})\) have already been generated and compared with the existing non-dominated solutions. The new procedure uses two data structures, min-heap \( H \) and stack \( N \). The generated solutions are stored temporarily in the min-heap. The solutions that are confirmed to be non-dominated are removed from the min-heap and stored in the stack. The value of first objective \( f_1 \) is used as key of min-heap. If two tuples have the same key (\( f_1 \) value), then they are compared using the second objective \( f_2 \).

We imagine all possible solutions being arranged in a \( t \times r \) table (as shown in Figure 1c) and identify each solution tuple using a row number and column number. An integer array \( \text{flag} \) of size \( r \) stores the row index of recently removed tuple (from min-heap) for each column.

Algorithm 2 proceeds as follows. Initially, the stack \( N \) contains a dummy solution \((0, \infty)\) (lines 4-5). The first tuple \((a_1 + c_j, b_1 + d_1)\) is generated and added to the min-heap \( H \) since its \( f_1 \) value is the lowest (lines 6-7). At the start of each iteration, the tuple \( \text{min} = (f_1, f_2) \) is removed from the root node (line 10). Its \( f_2 \) value is compared with \( f_2 \) value of solution at the top of the stack \( N \). If the value is smaller, then the solution \( \text{min} \) is pushed in the stack (lines 11-13). Note that, solutions are removed from the heap in the increasing order of first objective \( f_1 \). If \( f_1 \) value of \( \text{min} \) is equal to \( f_1 \) value of solution at the top of the stack, then \( f_2 \) value of \( \text{min} \) must be less than the \( f_2 \) value of top solution.

Assume that the solution \( \text{min} \) belongs to row \( j \) and column \( i \) in the solution table of size \( t \times r \) (Figure 1c). The \( i^{th} \) entry in \( \text{flag} \) is updated with the row number \( j \) since it is the most recent solution in column \( i \) which is removed from the min-heap (line 16).

Theorem 4. The memory requirement of the CombineEff procedure is \( O(r) + |N| \), where \( r = |E_1| \) and \( N \subseteq E \) is the set of non-dominated solutions to the combined problem.

We claim that the size of the min-heap \( H \) at any point in time during execution never exceeds \( r \). Suppose that the candidate solutions \( E \) to the problem are arranged in \( t \times r \) table as depicted in (Figure 1c). If there are more than \( r \) tuples in min-heap, then by pigeonhole principle at least two tuples, say \( e_u \) and \( e_v \) must belong to the same column. Suppose that \( u < v \) and \( v \) indicates the row numbers of these tuples and let \( u < v \). Then by Theorem 3, \( e_u \) must be smaller than \( e_v \) with respect to the first objective \( f_1 \), i.e., \( e_u, f_1 < e_v, f_1 \) and \( e_u, f_1 \) represents the first objective value of the tuples \( e_u \) and \( e_v \), respectively. However, CombineEff does not generate \( e_u \) until \( e_u \) is removed from the min-heap. Therefore, tuples \( e_u \) and \( e_v \) cannot be present in the min-heap at the same time. Hence, the contradiction.

The size of the integer array \( \text{flag} \) is also \( O(r) \). Further, the stack \( N \) only stores the non-dominated solutions. Therefore, the total memory requirement of the CombineEff procedure is \( O(r) + |N| \), where \( N \subseteq E \) is the set of non-dominated solutions to the combined problem. As the \( for \) loop runs for \( r \times t \) iterations and the maximum amount of work done in each loop is \( O(\log(r)) \) (min-heap operations), the time complexity of the CombineEff procedure is \( O(r \times t \cdot \log(r)) \). Note: Algorithm 2 assumes that solutions in both sets \( E_1 \) and \( E_2 \) are sorted by the first objective. It generates all possible \( r \times t \) solutions in a strategic manner. The same algorithm works well (with minimum modifications) even if we sort solutions by the second objective.

Illustration: Consider two sets of non-dominated solutions \( E_1 \) and \( E_2 \) (sorted by the first objective) given in Table 2. The combined set of solutions \( E \) is also shown in Table 2. The non-dominated solutions are highlighted in bold. Table 3 illustrates the contents of
Algorithm 2 Efficient Algorithm to Combine Non-dominated Solutions of Two Sub-problems

1: procedure CombineEff
2: Input: Two sets of non-dominated solutions $E_1$ and $E_2$. Both sets are sorted by the first objective.
3: Output: A set of non-dominated solutions $N$ in the combined set $E = \{(a + c, b + d)\mid (a, b) \in E_1 \text{ and } (c, d) \in E_2\}$
4: initialize stack $N$
5: $N$.push((0, oo))
6: initialize min heap $H$ of size $r$
7: $H$.add((a1 + c1, b1 + d1))
8: $flag[1] = \text{int}[r]$
9: for $l = 1$ to $r$ do
10: $(f_1, f_2) = H$.remove() // min solution
11: if $f_2 < N.top()$, then
12: $N$.push($f_2$) // increments top and pushes element into stack
13: end if
14: $i = \text{col index of min tuple} (f_1, f_2)$
15: $j = \text{row index of min tuple} (f_1, f_2)$
16: $flag[i] = j$ //update entry in pos $i$ with the row index $j$ of min soln
17: if $i + 1 \leq r$ and $(j + 1 = 1 \text{ or } flag[i + 1] = j + 1)$ then
18: $H$.add($(a_{i+1} + c_j, b_{i+1} + d_j))$ // generate right tuple
19: end if
20: if $j + 1 \leq r$ and $(i + 1 = 1 \text{ or } flag[i + 1] = j + 1)$ then
21: $H$.add($(a_i + c_{j+1}, b_i + d_{j+1}))$ // generate bottom tuple
22: end if
23: end for
24: return solutions in stack $N$
25: end procedure

Table 2: Two sets $E_1$ and $E_2$ of non-dominated solutions each containing four tuples (sorted by the first objective), and their combination $E$ consisting of $4 \times 4 = 16$ candidate solutions. Non-dominated solutions are highlighted in bold.

<table>
<thead>
<tr>
<th>$E_1$</th>
<th>$E_2$</th>
<th>Combined $E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,4)</td>
<td>(2,10)</td>
<td>(3,14)</td>
</tr>
<tr>
<td>(5,3)</td>
<td>(4,7)</td>
<td>(5,11)</td>
</tr>
<tr>
<td>(6,2)</td>
<td>(5,6)</td>
<td>(6,10)</td>
</tr>
<tr>
<td>(13,1)</td>
<td>(7,5)</td>
<td>(8,9)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(12,8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(13,7)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(20,6)</td>
</tr>
</tbody>
</table>

heap $H$, stack $N$ and array $flag$ during the iterations of Algorithm 2. In the beginning, a dummy entry $(0, oo)$ is added to the stack (line 5). Further, the solution tuple $(1,4)$ from $E_1$ and the solution tuple $(2,10)$ from $E_2$ are combined to produce $(3,14)$ and added to the heap (line 7). Thus, at the start of the first iteration, that is, $l = 1$, heap $H$ contains $(3,14)$. Subsequently, it is removed from the heap and compared with the top element in stack $N$. As the second objective value of the top element is $\infty$, the tuple $(3,14)$ is added to the stack (lines 11-13) and the dummy entry $(0, oo)$ is removed. As the first entry in the first column is generated and removed from the heap, $flag[0]$ is set to 1. Subsequently, the solution tuple $(5,3)$ from $E_1$ and the solution tuple $(2,10)$ from $E_2$ are combined to produce $(7,13)$, and added to the heap (lines 17-19). Similarly, the solution tuple $(1,4)$ from $E_1$ and the solution tuple $(4,7)$ from $E_2$ are combined to obtain $(5,11)$, and added to the heap (lines 20-22). Thus, at the start of the second iteration, that is, $l = 2$, heap $H$ contains $(5,11)$ and $(7,13)$, and stack $N$ contains a non-dominated solution $(3,14)$. In this way, in each iteration, the algorithm removes a tuple from heap $H$, pushes it on stack $N$ if appropriate, and generates at most two new tuples and adds them to heap $H$.

Table 3: Execution of Algorithm 2 on solutions $E_1$ and $E_2$ (Table 2) of two sub-problems (sorted by the first objective).

<table>
<thead>
<tr>
<th>Iter $l$</th>
<th>Heap $H$</th>
<th>Stack $N$</th>
<th>$flag$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(3,14)</td>
<td>(0, oo)</td>
<td>[0,0,0,0]</td>
</tr>
<tr>
<td>2</td>
<td>(5,11)(7,13)</td>
<td>(3,14)</td>
<td>[1,0,0,0]</td>
</tr>
<tr>
<td>3</td>
<td>(6,10)(7,13)</td>
<td>(3,14)(5,11)</td>
<td>[2,0,0,0]</td>
</tr>
<tr>
<td>4</td>
<td>(7,13)(8,9)</td>
<td>(3,14)(5,11)(6,10)</td>
<td>[3,0,0,0]</td>
</tr>
<tr>
<td>5</td>
<td>(8,9)(9,10)(8,12)</td>
<td>(3,14)(5,11)(6,10)</td>
<td>[3,1,0,0]</td>
</tr>
<tr>
<td>6</td>
<td>(8,12)(9,10)</td>
<td>(3,14)(5,11)(6,10)(8,9)</td>
<td>[4,0,0,0]</td>
</tr>
<tr>
<td>7</td>
<td>(9,10)(15,11)</td>
<td>(3,14)(5,11)(6,10)(8,9)</td>
<td>[4,1,0,0]</td>
</tr>
<tr>
<td>8</td>
<td>(10,9)(15,11)(10,9)</td>
<td>(3,14)(5,11)(6,10)(8,9)</td>
<td>[4,2,0,0]</td>
</tr>
<tr>
<td>9</td>
<td>(10,9)(15,11)</td>
<td>(3,14)(5,11)(6,10)(8,9)</td>
<td>[4,2,0,0]</td>
</tr>
<tr>
<td>10</td>
<td>(11,8)(15,11)(12,8)</td>
<td>(3,14)(5,11)(6,10)(8,9)</td>
<td>[4,3,2,0]</td>
</tr>
<tr>
<td>11</td>
<td>(12,8)(15,11)</td>
<td>(3,14)...(8,9)(11,8)</td>
<td>[4,3,3,0]</td>
</tr>
<tr>
<td>12</td>
<td>(13,7)(15,11)</td>
<td>(3,14)...(8,9)(11,8)</td>
<td>[4,4,3,0]</td>
</tr>
<tr>
<td>13</td>
<td>(15,11)</td>
<td>(3,14)...(11,8)(13,7)</td>
<td>[4,4,4,0]</td>
</tr>
<tr>
<td>14</td>
<td>(17,8)</td>
<td>(3,14)...(11,8)(13,7)</td>
<td>[4,4,4,1]</td>
</tr>
<tr>
<td>15</td>
<td>(18,7)</td>
<td>(3,14)...(11,8)(13,7)</td>
<td>[4,4,4,2]</td>
</tr>
<tr>
<td>16</td>
<td>(20,6)</td>
<td>(3,14)...(11,8)(13,7)</td>
<td>[4,4,4,3]</td>
</tr>
<tr>
<td>17</td>
<td>(23,4)</td>
<td>(3,14)...(13,7)(20,6)</td>
<td>[4,4,4,4]</td>
</tr>
</tbody>
</table>

3.2 Evolutionary Algorithms

Classical optimization methods typically convert a multi-objective problem to a single objective problem by assigning appropriate weights to each objective. Such methods yield only one Pareto-optimal solution for the given weights. To obtain multiple solutions, they have to be applied many times. On the other hand, evolutionary algorithms are considered to be more effective in solving multi-objective optimization problems since they work with a population of solutions and produce a set of non-dominated solutions in a single simulation run. Several multi-evolutionary algorithms have been proposed in the literature. NSGAI is an extended genetic algorithm that uses the concept of elitism, fast non-dominated sorting and crowding distance (for diversity) to solve multi-objective problems [12]. Recently, a binary particle swarm optimizer BMOPSOCD was proposed to solve WSLAP [3] and was shown to generate better solutions than NSGAI. BMOPSOCD maintains a set of non-dominated solutions in the external archive and uses crowding distance to find diverse solutions. In this work, we make use of NSGAI and BMOPSOCD to find the non-dominated solutions for each web service in WSLAP.

4 EXPERIMENT

We compare the efficacy of the proposed D&C approach with multi-objective evolutionary algorithms (MOEAs) proposed in the literature [3, 6, 7] on test instances of different sizes. Specifically, we consider four algorithms: NSGAI, BMOPSOCD, D&C NSGAI and D&C BMOPSOCD. All algorithms were implemented in Python version 3.7 and the experiments were conducted on a Windows 10 machine with i7-8650U 2.11 GHz processor and 16GB of RAM. We follow the experimental methodology used by [3].
4.1 Test Instances
We conducted experiments with 14 different WSLAP instances described in [3]. The search space of an instance is computed using the number of services and candidate locations \(2^sn\) as shown in Table 4. Since the computation complexity of an instance also depends on the number of user centers, for each search space, the number of user centers is varied. The test instances employ real-world WSdream dataset in obtaining latency numbers. This dataset contains only latencies between candidate locations and user centers, and lacks deployment costs for candidate locations and invocation frequencies for web services. Therefore, to generate a complete test instance, we followed the approach suggested in [3]. We randomly generated the deployment costs for candidate locations according to a normal distribution with a mean of 100 and a standard deviation of 20. We also randomly generated the invocation frequencies for user centers from a uniform distribution between 1 and 120.

Table 4: Different WSLAP Instances Used in Experiments.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Services</th>
<th>Locations</th>
<th>User Centers</th>
<th>Search Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>5</td>
<td>10</td>
<td>2^100</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>2^100</td>
</tr>
<tr>
<td>3</td>
<td>50</td>
<td>15</td>
<td>20</td>
<td>2^750</td>
</tr>
<tr>
<td>4</td>
<td>50</td>
<td>15</td>
<td>40</td>
<td>2^750</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>25</td>
<td>20</td>
<td>2^1250</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>25</td>
<td>40</td>
<td>2^1250</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>15</td>
<td>20</td>
<td>2^1500</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
<td>15</td>
<td>40</td>
<td>2^1500</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
<td>25</td>
<td>20</td>
<td>2^2500</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>25</td>
<td>40</td>
<td>2^2500</td>
</tr>
<tr>
<td>11</td>
<td>200</td>
<td>25</td>
<td>40</td>
<td>2^5000</td>
</tr>
<tr>
<td>12</td>
<td>200</td>
<td>25</td>
<td>80</td>
<td>2^5000</td>
</tr>
<tr>
<td>13</td>
<td>200</td>
<td>40</td>
<td>40</td>
<td>2^5000</td>
</tr>
<tr>
<td>14</td>
<td>200</td>
<td>40</td>
<td>80</td>
<td>2^5000</td>
</tr>
</tbody>
</table>

4.2 Performance Metrics
We use HyperVolume (HV) [14, 15] and Inverted Generational Distance (IGD) [16] to evaluate the diversity and quality of solutions produced by four algorithms (NSGAI, BMOPSOCD, D&C NSGAII and D&C BMOPSOCD). HV is a measure that reflects the volume enclosed by a solution set and a reference point. A larger HV value indicates a better quality solution set. However, for calculating the HV value, we need a true Pareto front, and for our problem, the true Pareto front is unknown. Therefore, as discussed in [3], we computed an approximated Pareto front by combining all solutions produced by four algorithms and then applying a non-dominated sorting to obtain the final non-dominated set.

4.3 Parameter Settings
The parameter values used for each algorithm considered in the evaluation are shown in Table 5. We found the parameter values for NSGAI algorithm empirically, that is, we tried several values for parameters, and observed whether the solutions have converged (have similar fitness values between two consecutive generations). We used the same values of crossover probability (0.8) and mutation probability (0.2) for D&C NSGAII. The size of a chromosome used for NSGAII is \(s \cdot n\), where \(s\) is the number of web services and \(n\) is the number of candidate locations. Since D&C NSGAII, solves the location allocation for each web service independently, the size of a chromosome, in this case, is just \(n\). As the search space of NSGAII is larger \(2^sn\), we used a population size of 250. As the search space of D&C NSGAII is smaller \(2^n\), we used a smaller population size of 20. Note that, our study focuses on the effectiveness of D&C approach instead of selecting the best parameter set.

For BMOPSOCD algorithm, we used the parameter values as described in [3]. The value of static inertia weight \(w\) was set to 0.4 and the mutation probability \(\rho_m\) was set to 0.5. The parameters \(c_1\) and \(c_2\) were both set to 1. Hence, particle’s personal best and swarm’s global best had an equal influence on the swarm.

Table 5: Parameters Choice for the Four Algorithms.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>NSGAII</th>
<th>D&amp;C NSGAII</th>
<th>BMOPSOCD</th>
<th>D&amp;C BMOPSOCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>250</td>
<td>20</td>
<td>250</td>
<td>20</td>
</tr>
<tr>
<td>Chromosome size</td>
<td>(s \cdot n)</td>
<td>(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tournament size</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Crossover probability</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
</tr>
<tr>
<td>Mutation probability</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>Maximum generations</td>
<td>250</td>
<td>40</td>
<td>250</td>
<td>40</td>
</tr>
</tbody>
</table>

We normalized two objective functions (latency and cost) between 0 and 1. The point (1, 1) is the extreme point of objective values. We used (1, 1) as the reference point in calculating HV. For each experiment, the proposed algorithm was run ten times independently. The best results of all the runs were compared. To obtain the best result of ten runs, the results of all ten runs were combined and sorted by the non-dominated values.

4.4 Results
Figure 2a depicts the set of non-dominated solutions produced by the four algorithms for one of the instances (Instance 3). For better clarity, we have shown the closer view of the non-dominated solutions obtained for the instance in figure 2b. The figure clearly shows that D&C NSGAII and D&C BMOPSOCD are able to find better and diverse solutions than NSGAII and BMOPSOCD. Similar trends were observed for other instances. Although NSGA-II and
BMOPSOCD have longer tails (i.e., many solutions with cost > 0.5) than D&C NSGAI and D&C BMOPSOCD, the solutions present in the tails are dominated by the solutions of the D&C algorithms. Table 6 shows HV values and Table 7 shows IGD values (all values are rounded to two decimal places) calculated using non-dominated solutions obtained by each algorithm for all fourteen instances. A larger HV value indicates a better and diverse solution set. A lower IGD value indicates a better quality solution set. From the Tables 6 and 7, it is clear that solutions obtained using the D&C approach are much better than the ones obtained using the combined approach in terms of quality as well as diversity. Overall, D&C BMOPSOCD produced better solutions than the other three algorithms. Further, as observed in [3], BMOPSOCD produced better results than NSGAI. However, D&C NSGAI resulted in better solutions than BMOPSOCD.

Table 6: HV Values from the Four Algorithms

<table>
<thead>
<tr>
<th>Instance</th>
<th>NSGA II Mean</th>
<th>NSGA II Std</th>
<th>D&amp;C NSGA II Mean</th>
<th>D&amp;C NSGA II Std</th>
<th>BMOPSOCD Mean</th>
<th>BMOPSOCD Std</th>
<th>D&amp;C BMOPSOCD Mean</th>
<th>D&amp;C BMOPSOCD Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.72</td>
<td>0.03</td>
<td>0.82</td>
<td>0.01</td>
<td>0.78</td>
<td>0.01</td>
<td>0.90</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.68</td>
<td>0.07</td>
<td>0.92</td>
<td>0.00</td>
<td>0.89</td>
<td>0.01</td>
<td>0.96</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>0.55</td>
<td>0.09</td>
<td>0.95</td>
<td>0.00</td>
<td>0.89</td>
<td>0.01</td>
<td>0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0.56</td>
<td>0.09</td>
<td>0.95</td>
<td>0.00</td>
<td>0.90</td>
<td>0.01</td>
<td>0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0.54</td>
<td>0.10</td>
<td>0.97</td>
<td>0.00</td>
<td>0.93</td>
<td>0.01</td>
<td>0.98</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0.55</td>
<td>0.10</td>
<td>0.97</td>
<td>0.00</td>
<td>0.93</td>
<td>0.00</td>
<td>0.98</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>0.54</td>
<td>0.09</td>
<td>0.95</td>
<td>0.00</td>
<td>0.86</td>
<td>0.01</td>
<td>0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>0.55</td>
<td>0.09</td>
<td>0.95</td>
<td>0.00</td>
<td>0.87</td>
<td>0.01</td>
<td>0.97</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>0.54</td>
<td>0.10</td>
<td>0.97</td>
<td>0.00</td>
<td>0.90</td>
<td>0.01</td>
<td>0.98</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>0.54</td>
<td>0.10</td>
<td>0.96</td>
<td>0.00</td>
<td>0.91</td>
<td>0.01</td>
<td>0.98</td>
<td>0.00</td>
</tr>
<tr>
<td>11</td>
<td>0.51</td>
<td>0.00</td>
<td>0.96</td>
<td>0.00</td>
<td>0.88</td>
<td>0.01</td>
<td>0.98</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.51</td>
<td>0.00</td>
<td>0.96</td>
<td>0.00</td>
<td>0.89</td>
<td>0.01</td>
<td>0.98</td>
<td>0.00</td>
</tr>
<tr>
<td>13</td>
<td>0.51</td>
<td>0.00</td>
<td>0.95</td>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
<td>0.98</td>
<td>0.00</td>
</tr>
<tr>
<td>14</td>
<td>0.51</td>
<td>0.00</td>
<td>0.94</td>
<td>0.00</td>
<td>0.92</td>
<td>0.00</td>
<td>0.98</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 7: IGD Values from the Four Algorithms

<table>
<thead>
<tr>
<th>Instance</th>
<th>NSGA II Mean</th>
<th>NSGA II Std</th>
<th>D&amp;C NSGA II Mean</th>
<th>D&amp;C NSGA II Std</th>
<th>BMOPSOCD Mean</th>
<th>BMOPSOCD Std</th>
<th>D&amp;C BMOPSOCD Mean</th>
<th>D&amp;C BMOPSOCD Std</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.19</td>
<td>0.02</td>
<td>0.14</td>
<td>0.00</td>
<td>0.15</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.17</td>
<td>0.04</td>
<td>0.07</td>
<td>0.00</td>
<td>0.07</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>0.34</td>
<td>0.07</td>
<td>0.05</td>
<td>0.01</td>
<td>0.09</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>0.25</td>
<td>0.05</td>
<td>0.04</td>
<td>0.01</td>
<td>0.05</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>0.32</td>
<td>0.07</td>
<td>0.03</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>0.30</td>
<td>0.07</td>
<td>0.03</td>
<td>0.00</td>
<td>0.04</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>0.39</td>
<td>0.07</td>
<td>0.08</td>
<td>0.01</td>
<td>0.13</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>0.26</td>
<td>0.05</td>
<td>0.04</td>
<td>0.01</td>
<td>0.07</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>0.28</td>
<td>0.06</td>
<td>0.03</td>
<td>0.00</td>
<td>0.04</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>0.30</td>
<td>0.06</td>
<td>0.04</td>
<td>0.01</td>
<td>0.05</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>11</td>
<td>0.27</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
<td>0.05</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.26</td>
<td>0.01</td>
<td>0.04</td>
<td>0.00</td>
<td>0.05</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>13</td>
<td>0.30</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>14</td>
<td>0.28</td>
<td>0.01</td>
<td>0.04</td>
<td>0.01</td>
<td>0.03</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Figure 2: a) Pareto fronts obtained from the four algorithms for instance 3 (50,15,20). b) Closer view of the pareto fronts.

5 CONCLUSION AND FUTURE WORK

In this paper, we show that WSLAP can be solved effectively using a Divide and Conquer (D&C) approach; wherein, the location-allocation problem is solved independently for each web service. Further, we propose a novel merge algorithm to combine the solution from each service. We used NSGA-II and BMOPSOCD to compute the Pareto front for each service. We compared performance of both the algorithm with and without D&C approach. Our extensive experiments showed that the D&C approach is able to produce better quality (lower Inverted Generational Distance) and diverse solutions (higher HyperVolume) than solving the WSLAP as one problem. The computational time is also less for D&C approach.

Further, the computational time can be significantly reduced by exploiting the parallelization capability of the proposed algorithm to merge the non-dominated solutions of multiple services simultaneously. Specifically, we can generate solutions for each service in parallel, and, during merge, we could do pairwise merges in parallel going all the way up the complete binary tree. Since the proposed merge algorithm is generic, we plan to study the effectiveness of the algorithm for other bi-objective optimization problems.
REFERENCES