RI 10006 28 May 2010 Computer Science

IBM Research Report

ReComp: QoS-aware Recursive service Composition at minimum Cost

Vimmi Jaiswal
Independent

India

Amit Sharma
IIT Kharagpur
Kharagpur
India

Akshat Verma
IBM Research Division
IBM India Research Lab
4-Block C, ISID Campus, Vasant Kunj
New Delhi - 110070. India.

ReComp: QoS-aware Recursive service Composition at minimum Cost

Vimmi Jaiswal, Amit Sharma, and Akshat Verma
[1] Independent
[2] IIT Kharagpur
[3] IBM Research-India

Abstract. *In this work, we address the problem of selecting the best set of available services or web functionalities (single or composite) to provide a composite service at the minimum cost, while meeting QoS requirements. Our* Recursive composition *model captures the fact that the available service providers may include providers of single as well as composite services; an important feature that was not captured in earlier models. We show that* Recursive Composition *is an intrinsically harder problem to solve than other studied compositional models. We use insights about the structure of the* Recursive Composition *model to design an efficient algorithm BGF-D with provable guarantees on cost. Our approach is equally suitable for all platforms that compose existing functionalities including web services and web mashups. As an embodiment, we design and implement the ReComp architecture for Recursive Composition of web-services that implements the BGF-D algorithm. We present comprehensive theoretical and experimental evidence to establish the scalability and superiority of the proposed algorithm over existing approaches.*

1 Introduction

The world wide web has often been viewed as a vast repository of knowledge and services in the form of custom web sites or services. A lot of research has focussed on leveraging these services to create composite value-added functionality with a very small lead time. The first wave of integrating distinct functionalities available on the web came in the form of web services. The key idea driving web-services was standardization of interfaces, which allowed an end-user to automatically discover, select and use a web service of his or her interest. The standardization of interfaces also led to the trend of composite value-added services; where a service provider uses existing web-services to compose value-added services. Web services had become very popular in the early part of the decade, with many directory services like RemoteMethods [5], xMethods [7], WSIndex [6] providing a list of available services.

An alternate model for providing integrated services on the web has emerged recently in the form of web mashups. Web mashups do not burden the provider of single services to build a standard interface but build custom interfaces for each web site that is being integrated. This allows mashups to be build over existing web sites as well and many web-sites today can be viewed as mashups. For example, the site zoomtra [20] provides the cheapest airfare between any 2 indian cities by querying the sites of individual airlines. The only requirement to create a mashup is that the web sites being integrated (or mashed up) expose their API publicly. The popularity of web mashups have led to the the emergence of more structured frameworks providing a rich set of user-configurable modules. For example, Yahoo Pipes [4], one of the most popular platforms of this type has over 30,000 customized feed processing flows in a short time [17]. Further, tools are being developed for rapid deployment of mashups [15] without programming effort. Web service framework and collaborative mashups provide two different web paradigms for integrating web functionality. We use the term service composition to capture such integration of web services as well as mashup of web site functionalities.

Service composition may be perceived as a problem of (i) discovery of services that may provide one or more of the functionalities needed for a composite service (ii) selection of the appropriate services, if more than one options are available and (iii) binding of the individual services. One may note that the first and third problems differ for the two paradigms but the service selection problem is identical for both web mashups and composite web services. The focus of this work is the service selection problem and we use service selection and service composition interchangeably in the paper.

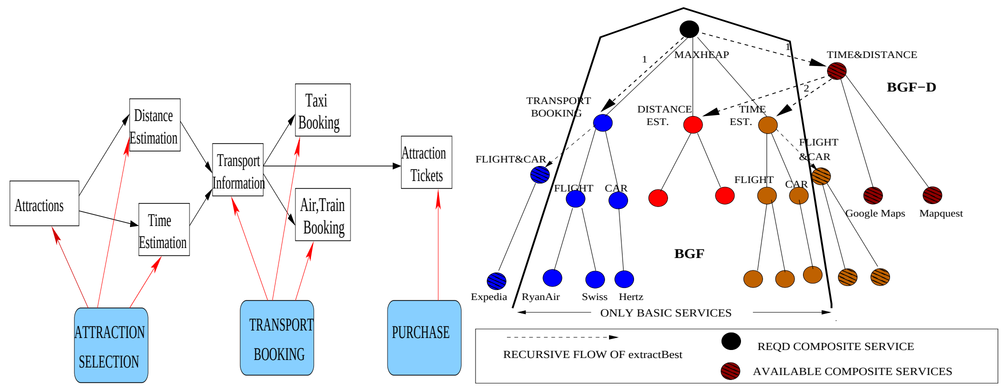
The emerging trend of service delivery or web site functionality with differentiated Quality of Service (QoS) adds an important facet to the service composition problem; a composite service provider may find more than one provider providing a functional requirement needed by the composite service. Hence, the composite service provider needs to select providers for individual requirements not only based on the functional properties of the service(s) provided by them, but on non-functional properties as well. Further, the composite service provider also needs to ensure that the composite service is provided at a competitive price while meeting any required QoS guarantees. Hence, a web provider can be characterized along two dimensions : (i) a cost charged by the provider and (ii) a QoS parameter guaranteed by the provider. A typical QoS parameter is average response time (e.g., delay for a stock quote service), which is also used in this work. However, our framework works for any linear QoS parameter.

The underlying service selection problem seen by the composite service provider is fairly complicated: competing providers may provide sub-services (or the functionalities) that the composite service needs at various [price,QoS] points. The presence of existing providers of composite services adds another dimension to the complexity. Service selection now needs to evaluate a composite service (providing a subset of the overall functionality required) against various combinations of basic or smaller composite services with respect to the cost-QoS tradeoff. We term the problem of recursive service selection from basic or smaller composite service as *Recursive Composition* and address this problem.

1.1 Travel Planner Service

We present the following service scenario example to elucidate *Recursive Composition*. Consider a value-added web mashup or service that provides complete travel planning using various individual services.

The Travel Planner service (Fig. 1(a)) consists of (i) Attraction Selection (ii) Transport Booking and (iii) Purchase, which use other existing services in



(a) (b)

Fig. 1. (a) Composite Travel Planner Service and (b) Recursive Flow

order to build the composite service. The Travel Planner service first invokes a provider of Tourist *Attractions* Service to create a list of nearby attractions and then invokes providers of Mapping Services to provide *Distance* and travel *Time Estimation* between various attractions. The service then computes a target set of attractions and identifies transport arrangements using providers of *Transport Information* service. It then makes the required *Taxi, Train* and *Flight Booking.*

There may be many providers at different cost-performance points for each service. For example, a planner looking for tours in Europe would find that the train tickets can be purchased through *TrenItalia* Ticket Service or *Eurail* Ticket Service. *Eurail* website has a better response time but the cost of the same ticket is higher than *TrenItalia.* Further, a single provider may provide more than one required service. For example, *Google Maps* provides both distance and travel time estimate whereas goitaly.about.com provides only distances. Similarly, *Expedia* service provides both flight and car booking. Further, there are many existing mashup services on the web for travel that query the airlines, train or hotel websites that the travel planner service can leverage. Hence, the *Travel Planner Service* would need to select a subset of various providers in a manner such that all the required basic services can be obtained. Further, the *Travel Planner Service* needs to create this service with a fast response time and still provide cost effective options.

Fig. 1(b) shows the inherent recursive nature of the problem. At each level of the composition tree, composite services may exist which compete directly with the combination of two or more services in the same level. The service parameters (cost and delay) of these dynamically composed sub-services is obtained recursively by traversing its children in the tree. In the example shown, for estimating the Time and Distance related to a trip, the planner makes a choice between a composite service like *Google Maps*, and two separate services for *Time Estimation* and *Distance Estimation.* These two separate services are combined recursively from the simpler services below it, and then compared with the composite service. Further, the recursion may not be restricted to only one level. In the example shown, the *Time Estimation* service further breaks up into estimation for road travel, and flight travel. Another important characteristic of web-based services is the time-varying price of services, necessitating a late binding (or dynamic composition). Existing composition frameworks are unable to capture this inherent recursive nature of web services, and new algorithms are needed to handle dynamic composition of web functionalities in the *Recursive Composition* framework.

1.2 Contribution

We present a cost-minimizing QoS-aware *Recursive Composition* framework that uses basic as well as already existing composite services to create new value-added composite services on a service delivery platform within a QoS delay bound. We show that *Recursive Composition* is a fundamentally much harder problem than already existing composition forumulations. By showing that *Recursive Composition* is at least as hard as the *Set Cover* problem, we establish that existing approximation schemes (e.g., Dynamic programming based FPTAS) for already studied composition problems [19, 18] are not applicable for the *Recursive Composition* problem. We use insights from real service settings to identify a *Disjoint Set* property amongst the providers. Through a careful use of this property, we design a fast recursive algorithm *BGF-D* with good theoretical properties. To evaluate the *BGF-D* algorithm, we implement it in a recursive web service framework called *ReComp* and conduct a comparative evaluation of *BGF-D* algorithm with other approaches under a wide variety of scenarios. *BGF-D* meets all the requirements of automated, dynamic 'service composition agents': simplicity, efficiency, deterministic behavior, and guarantees on the goodness of the solution. Our proposed algorithm is useful for composition of web services as well as to create web mashups.

1.3 Related Work

Many researches have investigated techniques that deal with modeling and standardization of interfaces for reliable discovery of web services [1, 10] and in mechanisms for automated run-time composition of services [9, 12]. Of late, there has been a lot of interest in frameworks for automatically creating mashups [4], even extending it to spreadsheet-based creation of mashups for rapid deployment [15]. Recently, the algorithmic aspects of service composition have also attracted attention for web services. In [19], Yu and Lin investigate service selection algorithms to maximize utility with a single QoS constraint extending it to include multiple QoS constraints in [18]. Similar composition algorithms have been proposed in [8, 21], albeit in a slightly different framework.

However, all these algorithms use a Service Composition model, where every service provider provides *only one* service. As we show, the *Recursive Composition* model that uses both basic as well as composite service providers is fundamentally harder than the existing Service Composition models, and requires design of new methodologies. In our model, one provider can provide more than elementary service, packaged together as a composite service. Further, the objective in existing formulations is utility-maximization and none of them handle cost-minimization, which is also a popular objective for many service providers. Finally, dynamic composition of services require a composition algorithm that is fast, simple to implement, and has deterministic behaviour with cost guarantees. Most of the earlier work is based on heuristics [18], Dynamic Programming [19] or Integer Programming [8, 21], and is not suitable for on-the-fly dynamic composition.

2 Model and Problem Formulation

In this paper, we use the term composite service to refer to both composite web services as well as a mashup, service provider to denote web sites as well as web service provider, and basic service to denote both individual web services as well as a single web functionality from a website.

Consider a web delivery platform with N existing service providers and a new service provider that wants to offer a new composite service CS. We denote the basic services required to compose CS by the set $S = \{u_1, ...u_M\}$. Each service provider P_i on the service delivery platform is characterized by its offering set os_i, where each entry in the offering set corresponds to one or more of these basic services u_j. For each provider P_i and a basic service $u_j \in os_i$, d_i^j denote the delay bound guaranteed by P_i for providing service u_j. It is important to note here that a basic service interaction may require multiple rounds of interactions and d_i^j captures the duration between the first request to the service and the final response from the basic service. Generally, providers provide a delay guarantee for services that they provide, and we can directly plug in these guarantees for the delay in our model. Further, c_i denotes the cost or the price of using the provider P_i. For services u_j not provided by provider P_i, the corresponding d_i^j is set to infinity. In practise, a single provider may provider a service at multiple [Cost,QoS] points and we capture it in our model by treating each [Cost,QoS] as a new provider.

The goal of the composition problem is to create the composite service CS by selecting a subset of the N providers such that the total cost of providing the composite service is minimized. Further, the composite service should have a delay no more than some delay bound D. Formally, the goal of the composition problem is to find a matching x_i^j from the basic services u_j to the service providers P_i, such that

$$min \sum_{i=1}^{N} \mathbf{x}_i c_i; s.t. \sum_{i=1}^{N} \sum_{j=1}^{M} d_i^j x_i^j \leq D, \sum_{i=1}^{N} x_i^j \geq 1 \quad \forall j, \mathbf{x}_i = \max_j x_i^j \quad \forall i \qquad (1)$$

An intuitive way to visualize the problem is as a flow problem on a graph with M legs (parts), where we want to route unit flow through the M legs. In each leg j, there may be up to N links (one corresponding to each service provider that provides the service) that can route the flow, and any solution to the problem routes the unit flow through one or more of these links for each leg. The total cost of the solution is the cost of distinct service providers selected in the M legs. The optimal solution to the problem is a set of flows that has the minimum cost incurred in the M legs, and the delay averaged over all the flows is bounded by D. In typical settings, the average delay seen by all the separate flows should be bounded by D. We capture it using the following additional integrality constraint $x_i^j \in \{0, 1\}$. We call this version of the problem as *Bounded Delay Composition* and is the focus of this work.

3 Characterizing Recursive Composition

We investigate the *Bounded Delay Recursive Composition* problem in this work. Henceforth, we will use the term *Recursive Composition* to refer to this version of the problem. Researchers have looked at a simplified version of the problem earlier, where the platform has no composite services and all the N service providers provide exactly one of the M basic services. We formulate this restricted scenario in an equivalent, but slightly different formulation from the one in [19], called *Independent Set Composition.*

Definition 1 *A service delivery platform with N service providers is said to satisfy the* Independent Set Composition *property if for any two service providers P_i, P_j, $i \neq j$, we have $(os_i \cap os_j = \phi) \vee (os_i = os_j)$.*

Earlier solutions to this problem have looked at Dynamic Programming based solutions for the problem and we first investigate if the Recursive Composition problem can also benefit from such techniques. We first characterize the *Independent Set Composition* problem. This model captures the composition problems addressed in [19, 21] by framing cost as inverse of utility. We show that the problem is related to a generalization of the knapsack problem, called the Exact k-item Knapsack Problem (EkKP) problem.

It is well known that the knapsack problem is NP-hard and the same properties hold for unbounded knapsack and its $EkKP$ variant [11]. It is also known that $EkKP$ is polynomially solvable in $O(N^k)$ by brute force, if k is a fixed constant [13]. We have proved the hardness of the composition problem by showing that the Exact k-item Unbounded Knapsack (EkKP) problem can be converted to an instance of the composition problem. Hence, any solution to the composition problem would imply a solution to the EkKP problem. For lack of space, we have omitted all proofs that are available in the appendix.

Theorem 1. *Independent Set Composition is NP-hard.*

We have shown that the *Independent Set Composition* problem is NP-hard. However, knapsack and many of its generalization are amenable to approximation techniques and have a Fully Polynomial Time Approximation Sceheme (FPTAS) algorithm [11]. We show that the general *Recursive Composition* problem is much harder, and approximation algorithms with constant factor approximation bounds do not exist. To establish that, we have proved (proof in appendix) that the Recursive Composition problem is at least as hard as the *Set Cover* problem.

Definition 2 *Set Cover Problem: Given a finite universe $U = \{e_1, e_2, ..., e_n\}$ of N members, $S = \{s_1, s_2, ..., s_m\}$, $\forall i s_i \subseteq U$, a collection of subset of U and a weight function $w : s \rightarrow \mathbb{R}^+$ that assigns a positive real weight to each subset of U, find the minimum weight subcollection of S whose union is U.*

Theorem 2. *The Recursive Composition Problem is at least as hard as the Weighted Set Cover Problem.*

Set Cover Problem belongs to a different class of problems than single dimension knapsack problems. Feige has shown [14] that the Set Cover problem can not be approximated in polynomial time of $(1 - o(1)) \log N$, unless NP has quasi-polynomial time algorithms. Raz and Safra [16] have established a bound of $c \log N$, under the stronger assumption of $P \neq NP$. The above results clearly indicate that a Dynamic programming based algorithm would not be able to find the optimal solution for the *Recursive Composition* problem. Hence, existing techniques proposed for the Composition Problem can not be applied or adapted for the *Recursive Composition* problem. Further, the strong results preclude even the existence of constant factor approximation algorithms for the general problem. Hence, we investigate new approaches for this *very hard* problem.

4 Recursive Composition

Recursive Composition is one of the hardest problems within the NP-Complete class. Hence, we look for structural properties in the problem to solve it.

4.1 Disjoint Set Composition

A composite service typically can be broken down into one or more sub-services, where each sub-service may again be a composite service. A typical example is a *Shopping application*, that uses *Browsing, Credit Card Processing*, and *Inventory Management* services. The *Credit Card Processing* sub-service may again be broken down into Credit Card Verification and Credit Card Charging services. Similarly, *Inventory Management* may consist of Supply Chain Management services, Item Alerts, and Accounting services and one may use basic services or a composite service for the complete *Inventory Management* sub-service. However, one may never find a provider providing services from both Inventory and Credit Card domains, say, a provider providing both *Credit Card Verification* and *Supply Chain Management* services. Similarly, in our *Travel Planner* Service example, a *Mapping* service may provide *Distance* and *Time Information.* A *Booking Service* may provide *Taxi* and *Flight Booking.* But it is not common for a provider to provide services that cover both *Booking* and *Mapping* domains. For example, the same provider may not provide *Taxi Booking* and *Time Estimation* services together.

The clear separation of domains amongst the various sub-services often reflects itself in a disjointness property amongst the service providers. This assumption draws its weight from the fact that it is necessary to have an expertise in a domain to provide quality services. Hence, when a provider expands the offered services, he/she would usually offer another service in the same domain. Encompassing one domain completely before venturing into other domains also helps a provider in engaging customers. We term this property as *Disjoint Set Composition*, defined next.

Definition 3 *A service delivery platform with N service providers is said to satisfy the* Disjoint Set Composition *property if for any two service providers P_i, P_j, $i \neq j$, $(os_i \cap os_j \neq \phi) \quad \Rightarrow \quad (os_i \subseteq os_j) \vee (os_j \subseteq os_i)$.*

The *Disjoint Set Composition* property simplifies the *Recursive Composition* problem. However, this problem has at least 2 dimensions (a provider has a delay and the number of services it covers) and existing Dynamic Programming approaches based on 1-dimension knapsack problem can still not be applied. Even though the problem is still much tougher than *Independent Set Composition*, we design a fast algorithm with provable guarantees.

4.2 *BGF* Algorithm

We now present *Best Goodness First* (*BGF*) algorithm for the *Disjoint Set Composition* problem. We note that *Independent Set Composition* problem is a special case of this problem and hence the algorithm is also applicable for the *Independent Set* problem. We first present the algorithm in the simpler context of *Independent Set Composition* and then extend it to the *Disjoint Set* problem.

Recall that in the *Independent Set Composition* problem, each provider provides exactly one service. We use P_i^j to denote the i^{th} provider of the service u_j, d_i^j to denote the delay guaranteed by P_i^j, and c_i^j to denote the cost charged by P_i^j. We also assume that d_i^j is less than the delay bound D and $\forall i, j, k\ (d_i^j < d_i^k) \rightarrow (c_i^j > c_i^k)$. Note that any providers that do not satisfy these conditions can not be selected as part of optimal solution, and hence can be discarded. BGF algorithm first constructs a low delay solution S_o by selecting the least delay provider P_o^j for each of the required service u_j. Note that the solution thus constructed may incur a large cost. We then swap service providers in S_o by providers who provide the service at a lower cost. The order of swapping is dictated by a metric that we call the goodness of a service provider and is defined relative to the currently selected provider P_{new}^j for any given service u_j. Formally, the goodness g_i^j of P_i^j is given by $\frac{c_{new}^j - c_i^j}{d_i^j - d_{new}^j}$.

The method repeatedly invokes an *extractBest* method that returns the best provider P_{i*}^{j*} (with the highest goodness) to be added to the current selection S_{new}. BGF then replaces the service provider selected for the service u_{j*} in S_{new} with P_{i*}^{j*} . Observe that the provider P_{i*}^{j*} has the property that if we swap the previously selected service provider P_{new}^{j*} with it for providing the service u_{j*}, then the decrease in cost per unit increase in delay is the maximum amongst all possible swaps. We also check that the swap does not violate the delay constraint. If the required swap g_{i*}^{j*} for a new provider P_{i*}^{j*} causes the delay bound to be violated, we stop and create a new solution S_1 by swapping P_{i*}^{j*} in the original solution S_o. If S_1 has a cost less than the solution S_{new} achieved before S_1 was taken into consideration, then we return S_1, otherwise we return S_{new} as the solution. We also return a solution S_2 obtained by accepting the swap that violates the delay bound. The algorithm is similar to the one described in Fig. 2.

We now characterize the running time of BGF for *Independent Set Composition.* It is easy to see that that the method may try out every provider for each service once before converging. Without loss of generality, assume that there are n providers per service. Hence, it implies that we may have an outerloop of Mn, with each iteration taking time equal to updating the Goodness data structure G and extracting the best element from it. Since each service is independent of other services, the goodness entry needs to be updated only for the providers that provide the service u_{j*} that has been selected in the current iteration. This provides the following bounds on the running time of BGF.

Lemma 3. *The running time for BGF is* $O(Mn(Update(G) + extractBest(G)))$.

Theorem 3. *If the Goodness metric G is implemented as a sorted array, BGF runs in time quadratic the size of the input mn, i.e.,* $O(M^2n^2)$.

Theorem 4. *If the Goodness metric G is implemented as a heap ordered tree of depth* 2 *with all providers as leaf nodes and each basic service as an intermediate node, the running time of BGF is given by* $O(Mn \max\{M, n\})$.

The heap-based implementation of BGF (Fig. 1(b)) provides further insights. Observe that all the providers for each service are part of one sub-tree with the current best goodness entry being the root of this sub-tree. Similarly, all the best goodness entries of each service are hierarchically combined to form another sub-tree and the procedure *extractBest* returns the root node of this sub-tree. One may note that any changes in one sub-tree does not effect other providers and this fundamental insight leads to an efficient running time of BGF. Another important observation is that independence is not necessary for fast convergence as long as one can ensure that changes in one heap do not lead to (a) a large number of updates and (b) an expensive *extractBest* procedure. The *Disjoint Set* property ensures both of these properties. We next leverage this important observation to show how *BGF* works for *Disjoint Set Composition*.

4.3 $BGF - D$ for Disjoint Sets

```
Algorithm BGF-D(ServiceSet SS)
 $DS$ = getDisjointServiceSets($SS$)
 $\forall s_j \in DS$, Compute $DS_o^j$ = minDelaySet($s_j$)
  $S_o = \{DS_o^1, ..., DS_o^k\}, \cup_{l=1}^{k} DS_o^l = SS$
 $\forall s_j \in DS, \forall P_i^j \in s_j$ Compute $G = \{g_1^1, \ldots, g_i^j, \ldots\}$
  $S_{new} = S_2 = S_o$
  While $(Delay(S_2) < D)$
     $g_{i*}^{j*} = extractBest(G)$,    $S_2 = Swap(S_{new}, P_{i*}^{j*})$
     If(Delay$(S_2) < D$)
        $S_{new} = S_2$,    $\forall i$ $Recompute$ $g_i^{j*}$,  $Update(G, g_i^{j*})$
     Else
        $S_1 = Swap(S_o, P_{i*}^{j*})$
  If $cost(S_1) < Cost(S_{new})$ return $S_1, S_2$
  Else      return $S_{new}, S_2$
```

Fig. 2. BGF-D Algorithm

We now present an extension of *BGF* for *Disjoint Set Composition* that we call *BGF-D*. The key difference between *Independent Set Composition* and *Disjoint Set Composition* is in the choice of good service providers (providers with high goodness) available for swapping with the current best solution. In *Disjoint Set Composition*, we have providers who provide multiple services. Hence, the goodness function needs to be defined for a set of services, instead of a single service. This requires us to define the best service provider (the one with the highest goodness) for each set of services that has an eligible service provider.

In order to achieve this, we enhance the heap-based implementation of the goodness data structure by creating additional nodes for each set of services that has at least one eligible service provider. Hence, instead of a tree with depth 3 for *Independent Sets*, we now have a tree with depth equal to the number of disjoint set levels. The *extractBest* method returns the disjoint set with the highest goodness value in this new extended data structure. The goodness value for any such disjoint service set DS is calculated in a recursive manner as maximum of the goodness value for a service provider P_i whose offering set os_i is DS and the goodness of any disjoint providers P_j, P_{j+1}, ..., P_k whose combined offering set is DS ($os_j \cup os_{j+1} ... \cup os_k = os_i$). The provider P_i, with maximum goodness for the services set is selected as before in BGF. For calculating the composition of constituent services that leads to the best goodness, we invoke *extractBest* recursively on the services set, using the delay of the current composition for the services set as the delay bound. At each level, we select the first composition to violate the calculated delay bound, thus giving us an incremental decrease in cost with respect to the current configuration.

To elaborate with an example, consider Fig. 1(b) with an already existing composite service that provides both *Time & Distance Estimation*. Hence, while trying to build the most cost-efficient solution, we create another node for *Time & Distance Estimation*. Further, the goodness value of this node is calculated as the maximum of the goodness of all providers of this composite service and the goodness of the best goodness values of the two disjoint subsets (*Time Estimation, Distance Estimation*), whose union is this set. The rest of the algorithm proceeds as BGF. First, we calculate the minimum delay composition for the given set of services. This is done by recursively computing the disjoint sets and selecting the minimum delay providers within each disjoint set. Then, at each step, we swap the existing solution with the highest goodness entry (DS), unless it violates the delay bound. We describe the algorithm in Fig. 2.

The *Recursive* computation of *extractBest* works because the goodness data structure G has no cycles. The non-cyclic nature is a direct implication of the *Disjoint Set* property, which imposes a hierarchical structure to the data structure. Hence, it is now easy to verify by inductive reasoning similar to the one used for BGF that the running time of BGF-D is $O(N*(Update(G)+ExtractBest(G))$, where N is the total number of available providers. However, BGF-D computes goodness values for each disjoint set of providers that can be potentially swapped by a substitute set. This could potentially lead us to an exponential number of subsets. We again leverage the disjointness property and show that the number of such subsets (Lemma 4) are of the same order as the number of services M. This ensures that the running time of *extractBest* for BGF-D is of the same order as BGF. The other component of running time is the time required to update the *Goodness* heap. However, the disjointedness proprty ensures that selection of a provider only impacts the nodes in that sub-tree. This ensures that the time for update is bounded by the number of providers that provide the service. Hence, the results about the running time of BGF also hold for BGF-D. We have proved the following results (proof in appendix) for BGF and BGF-D.

Lemma 4. *Number of disjoint sets with at least one service provider is* $O(M)$.

Lemma 5. *If the solution S achieved by BGF at any step has a delay D, then S is the minimum cost solution which has a delay less than or equal to D.*

Let C_{max} be the cost of the least delay solution (S_o). Using the bound on d_i^j, an averaging argument and Lemma 5, we bound the cost of BGF.

Theorem 5. *The Solution S_1 returned by BGF has a cost no more than $(C_{max}+C_O)/2$ where C_O is the cost of the optimal. For a delay bound D, BGF returns a solution (S_2) with delay less than $2D$ and cost less than the cost of the optimal solution for delay less than or equal to D. Further, the first solution returned by BGF has a delay less than D and a cost bounded by twice the cost of the optimal solution with delay less than D.*

Corollary 1. *If BGF is executed with delay $D/2$, its solution has delay bounded by D and cost bounded by cost of the optimal solution with delay D.*

5 Evaluation

5.1 Prototype Implementation

We have designed *ReComp*, a Recursive Service Composition architecture to evaluate the *BGF-D* algorithm. *ReComp* takes as input a composite service specification and its SLA and outputs a composite service description in the BPEL4WS language [1]. The composite service description can be executed by a workflow engine such as BPWS4J [3]. The composition flow in *ReComp* is started with *Discovery* of the services that either match the complete user given service specification or parts of it using UDDI queries. The central intelligence of *ReComp* lies in the *Composition Analytic* module, which determines the optimal basic services and their layout that best meets the composite service requirement. This module implements *BGF-D* and uses the service description and the SLAs of various discovered services to compute an optimal composition of the services that can meet the SLA for the composite specification at the lowest cost. An *Assembler* module then uses the interface definitions of individual web services in the WSDL specification to generate a composite service description using the BPEL4WS specification. This description is the final ouput generated by *ReComp* and is fed to a workflow engine for execution.

5.2 Experimental Settings

Recursive Composition is a much harder problem than existing composition models and earlier heuristics or Dynamic Programming-based solutions for service composition can not solve *Recursive Composition.* Since no existing algorithms can deal with recursive composition, we implemented the following competing algorithms in the *Composition Analytic* module and performed a comparative study with *BGF-D*. These algorithms cover the spectrum between techniques that are fast, and techniques the are close to the optimal solution.

Fractional Optimal Composition: This algorithm proceeds exactly as *BGF-D* until the last step, where we assign fractional providers to the selected service-set so as to completely consume the delay bound available. The cost obtained by *Fractional Optimal* is less than the *Optimal* and provides a lower bound to compare the effectiveness of any composition algorithm . We do not compare against the optimal as computing it takes exponential time.

Equal Delay Allocation: The *Equal Delay* method breaks down the delay bound for the composite service into equal delay bounds for the basic services. It then selects a provider for each service that is able to provide the service at the minimum cost within the delay bound for that basic service.

Proportional Delay Allocation: This algorithm is similar to Equal Delay with difference that the delay bounds for each basic service are assigned in proportion to the mean delay of that service. The mean delay of a service is defined as the average delay of the service across all providers that provide the service. One can also use an Integer Programming (IP) solver to solve the *Recursive Composition* problem. However, it would take an inordinately large amount of time and fare worse than Fractional Optimal Composition. As we show, *BGF-D* approaches fractional optimal in our experiments, implying that it returns a solution very close to the optimal.

The baseline setting of our experiments simulate the *Travel Planner Service*, once a set of *Attractions* have been selected. The service requires 5 basic services, namely *Distance Estimation, Time Estimation, Transport Information, Taxi Booking and Train Booking.* The *UDDI* registry does not contain all the services required for us to perform this study and hence, we simulated a *UDDI* registry that captures the required scenario. The relative weightage (or contribution) of each service to the cost and delay of the composite service was captured

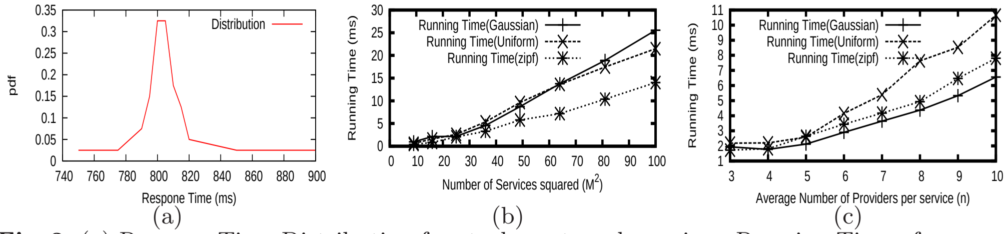


Fig. 3. (a) Respone Time Distribution for stock quote web services. Running Time of BGF-D with increase in (b) basic services and (c) providers.

by a weight metric (W). For each service, the weight distribution was computed using a Zipf distribution. For each service u_i, we generated 5 providers from a cost-delay curve, where cost of a service was inversely proportional to delay ($c_i d_i^j = W$, W is the weight of the service). The model is based on the well known knee curve between response time and the load factor of typical Markovian queueing model. In this model, the response time d_i is inversely proportional to $(1 - rho)$, where ρ is the utilization of the server. This combined with the fact that idleness $(1 - \rho)$ is directly proportional to the number of servers used (which determines cost), implies that c_i is inversely proportional to d_i^j. We also added 2 composite services for (i) *Distance* and *Time Estimation* and (ii) *Taxi* and *Flight Booking*. The providers of composite services (3 per composite service) give a discount of 5% over the sum of cost of independent providers.

We also studied the performance by varying all the parameters from the baseline. The most important metrics for a compositional algorithm are scalability and cost optimization. Hence, we change the number of basic services and the average number of providers per service. Further, we vary the overall weightage of the basic services by changing the weight metric ($W = C \times D$) of the service. In order to understand the structure of the weight function, we randomly sampled 50 services that provided stock quotations from the web service search engine seekda. We observed that the response time fits the Gaussian distribution (Fig. 3(a)). Hence, we selected Gaussian distribution as one candidate to generate the weight metric. In order to study the applicability of our study to other settings like mashups, we also used the zipf distribution to generate the weight values. Zipf distribution is commonly found in web traffic (e.g., most of the web requests are targetted to only a few sites, file size distribution embedded in web pages are skewed) and captures the relative weightage of various web sites in a mashup [2]. We also experiment with the Uniform Distribution to capture scenarios that are in between the mean-heavy Gaussian and the skewed Zipf. Another important characteristic that may effect the performance of algorithms is the granularity of providers along the weight function of a service. Hence, we change the variance of the provider characteristics around the mean weight function and study the algorithms.

5.3 Experimental Results

Scalability of ReComp We first study the time taken to compose the service by *ReComp* using the *BGF-D* methodology. Fig. 3(b) shows that the running time of our proposed methodology has a quadratic relationship with the number of basic services required by the composite service. This is in line with the worst case running time of $Mn \max\{M, n\}$ as proven in Theorem 4. The quadratic relationship holds for all distributions of the weight functions. However, we observe that the running time is lower when the weight of the services is obtained by *Zipf* distribution. One may note that even though the worst case running time of *BGF-D* is given by Theorem 4, the actual running time may be lower if the method hits the delay bound and can swap no more providers. For the zipf distribution, the large skew in weightage between different component services leads to much fewer providers for *BGF-D* to swap with the current solution as we increase the number of providers. Hence, we find that the algorithm converges faster with a high skew weight distribution like *Zipf*.

We next investigate the change in running time with the average number of providers available per service (n). Fig. 3(c) shows a surprising linear relationship between running time and the average number of providers per service as opposed to the quadratic relationship suggested by Theorem 4. To understand this behavior, we take a closer look at the theorem and observe that the worst case running time is obtained by assuming that all providers for each service may need to be swapped and we need n updates in each swap, before we find the best solution. However, in actual execution, *BGF-D* needed to swap only a constant number (2 or less in many cases) of providers per service. Hence, in real execution, we see a linear relationship of *BGF-D* with n.

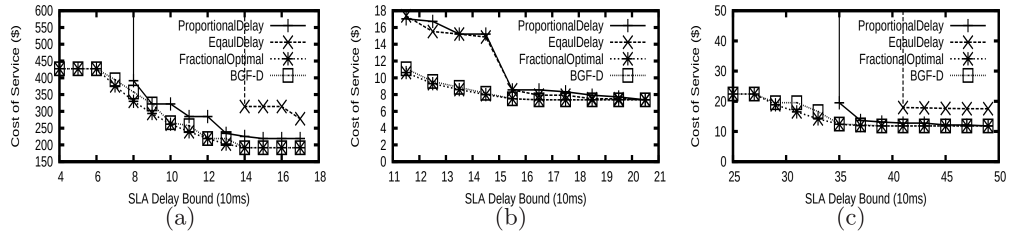


Fig. 4. Cost of service created by various methods with increase in Delay Bound for (a) zipf (b) Gaussian and (c) Uniform weight distribution. Baseline delay is 10 $\times 10ms$

Comparative Evaluation We now investigate the performance of *BGF-D* in comparison to the other methodologies. Fig. 4(a) studies the performance of various methodologies at the baseline experimental setting with change in the delay bound D of the composite service. Since the goal is cost minimization, all algorithms reduce cost as the delay bound is increased. We observe that *BGF-D* is able to find a solution with cost that is no more than 5% away from the fractional optimal solution. Note that fractional optimal is only a lower bound on the optimal solution. Hence, even in cases where *BGF-D* is costlier than fractional optimal, *BGF-D* solution may still be the optimal solution. We also observe that the solution provided by other competing methodologies are more than twice as expensive for the baseline delay. In fact, the *Equal Delay* solution is unable to find a solution for low delay bound. We find a similar observation to hold for Gaussian as well as Uniform distribution(Fig. 4(b),(c)). We also observe that *Equal Delay* is able to find solutions for *Gaussian* distribution for the complete range. This can be explained by the low variance used in the Gaussian distribution. However, for highly skewed weights, one may not be able to find any providers for a constant delay value as required by *Equal Delay.*

We also observe that *Proportional Delay*, *BGF-D* and *Fractional Optimal* converge together for high delays. A high delay bound implies that any combination of providers present an eligible solution. Hence, all methodologies eventually converge to the least cost providers for each basic service and delay of a provider is not relevant. Such high delay bounds are impractical for service scenarios with SLA guarantees and For practical delay bounds, a methodology like *BGF-D* intelligently optimizes the cost delay tradeoff to obtain a cost effective solution.

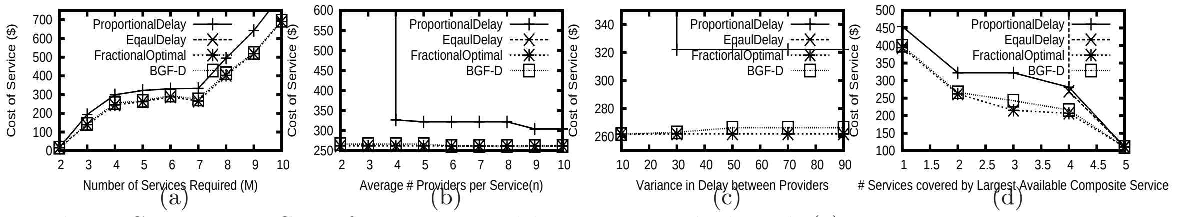


Fig. 5. Comparative Cost of service created by various methods with (a) increase in Required Services and (b) increase in Average Number of Providers per Service, (c) increase in Delay Variance amongst Providers of the Same Service and (d) coverage of available Composite Services

We study the cost of composite service with change in various service parameters in Fig. 5. The cost increases with increase in number of services for all methods (cost increases with increaese in services) but *BGF-D* outperforms *Proportional Delay* throughout the range (Fig. 5 (a)). With increase in number of providers (Fig. 5(b)), *BGF-D* approaches fractional optimal as it is more likely to find more providers near the fractional optimal solution. As delay variance increases, we observe (Fig. 5(c))that the difference between the fractional and the integral solution increases. This is because the chances of finding a provider near the fractional optimal solution decrease with increase in the variance of delay amongst the providers. Hence, the increase in cost because of rounding off a fractional solution to an integral solution increases with increase in provider variance. We also note that the algorithms maintain an almost similar cost as we increase the variance of delay. We observed that *Equal Delay* fairs very poorly in these experiments(Fig. 5). In fact, it fails to build up the required composite service whenever the delay bound is competitive. This is because a skewed distribution like Zipf violates the premise on which *Equal Delay* is based (that all services contribute equally to the delay). Existing Composite service providers provide a shortcut to obtaining a good solution if they provide the solution within the delay bound. Further, in our experimental methodology, we provide a lower cost to a composite provider to account for economy of scale. Hence, all methodologies are able to make use of existing composite providers to reduce their cost (Fig. 5(d)). Further, if a composite service can provide all services, all methodologies select the minimum cost provider for this composite service.

We observe that *Proportional* achieves solutions with good costs (within 25% of the cost of *BGF* for $M \leq 5$). We surmised that this may be because all services have the same Cost Vs Delay relationship (an inverse relationship) for the providers providing this service. Hence, we used a different cost vs delay curve ($a_1 c_i^j + a_2 d_i^j = a_3$) for half of the basic services. We observed (figure omitted for lack of space) that *Proportional* performed poorly because different services had different sensitivity to delay variations and a *Proportional* allocation was not able to compensate for them.

5.4 Conclusion

Our experiments comprehensively established *BGF-D* as an efficient algorithm significantly outperforming other approaches with performance close to optimal. The algorithm scales well with the number of services and providers and can be practically implemented for *Recursive Composition*. *BGF-D* meets all requirements for dynamic service composition: simplicity, scalability, deterministic running time, and provable guarantees on the cost of the solution.

References

1. Web Services Business Process Execution Language TC. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.
2. L. Breslau, C. Pei, F. Li, G. Phillips, S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proc. Infocom*, 1999.
3. Grid Workflow: BPWS4J. http://www.gridwork flow.org/snips/gridworkflow/space/BPWS4J.
4. Pipes:Rewire the Web. http://pipes.yahoo.com/pipes/.
5. RemoteMethods: Resource Home Web Services (service). http://www.remotemethods.com/.
6. Web Services Directory. http://www.wsindex.org/.
7. XMethods. http://www.xmethods.net.
8. Aggarwal, R., Verma, K., Miller, J. and Milnor, W. Constraint driven Web service composition in METEOR-S. In *IEEE SCC*, 2004.
9. R. Akkiraju, B. Srivastava, A. Ivan, R. Goodwin, and T. Syeda-Mahmood. SEMAPLAN: Combining Planning with Semantic Matching to Achieve Web Service Composition. In *Proc. ICWS*, 2006.
10. Eyhab Al-Masri and Qusay H. Mahmoud. Investigating Web Services on the World Wide Web. In *Proc. WWW*, 2008.
11. Caprara, A. et al. Approximation algorithms for knapsack problem with cardinality constraints. In *European Journal of OR*, 2000.
12. Dan, A. et al. Web services on demand: WSLA-driven automated management, In *IBM Systems Journal*, 2004.
13. Downey, R.G., and Fellows, M.R. Fixed-Parameter Tractability and Completeness II: On Completeness for W[1]. In *Theoretical Computer Science*, 141:109-131, 1995.
14. Uriel Feige. A Threshold of ln n for Approximating Set Cover. Journal of the ACM (JACM), v.45 n.4, p.634 - 652, July 1998.
15. W. Kongdenfha, B. Benatallah, J. Vayssire, R. Saint-Paul and F. Casati. Rapid Development of Spreadsheet-based Web Mashups. In *Proc. WWW*, 2009.
16. R. Raz and M. Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. Proc. STOC 1997.
17. A. Riabov, E. Bouillet, M. Feblowitz, Z. Liu and A. Ranganathan. Wishful Search: Interactive Composition of Data Mashups. In *Proc. WWW*, 2008
18. Tao Yu and Kewi-Jay Lin. Complex Services with Multiple QoS Constraints. In *ICSOC* , 2005.
19. Yu,T., and Lin, K.J. Service Selection Algorithms for Web Services with End-to-end QoS Constraints. In *Journal ISEBM*, 2005.
20. Cheap Air Tickets India. *http://www.zoomtra.com/*
21. Zeng, L et al. Quality Driven Web Service Composition. In *Proc. WWW*, 2003.

A Appendix

Theorem 1. *The Independent Set Composition problem is NP-hard.*

Proof. Let S be the set of items 1 to N for any instance K_I of the EkKP problem, with profits p_i and costs c_i respectively. Let the knapsack capacity by denoted by C. We create an instance C_I of the composition problem in the following manner. We define a composite service, which requires k basic services, and has a delay bound of C. Further, for each service j $(1 \leq j \leq k)$, we define N basic service providers, where each service provider s_i^j has delay c_i and cost $-p_i$. We first show that a feasible solution to one problem is also a feasible solution to the other.

Lemma 1. *Any solution to an instance C_I of the composition problem is also a solution to the corresponding EkKP problem instance K_I and vice versa.*

Proof. We first show that any solution (x_i^j) to the composition problem is also a solution to the original $EkKP$. Observe that the composition problem would pick exactly k service providers, one for each basic service. Hence, $\sum_{i,j} x_i^j = k$ and only k items will be picked in the knapsack. Further, the capacity constraint of the knapsack is not violated since $\sum_{i,j} c_i x_i^j < C$. The only thing remaining for us to show is that every solution to the $EkKP$ is also a solution for the composition problem. Consider any knapsack solution, which picks x_i copies of item i. The corresponding composition solution would pick x_i service providers with delay w_i and cost $-p_i$ respectively.

It is easy to verify that the optimal solution of the two problems are related, which leads to the hardness result.

Lemma 2. *The optimal solution O_I of the composition problem instance C_I is also an optimal solution for the EkKP problem instance K_I.*

Theorem 2. *The Recursive Composition Problem is at least as hard as the Weighted Set Cover Problem.*

Proof. Take any instance of the weighted Set Cover Problem with Universe $U = \{e_1, ..., e_n\}$ and the collection S of its subsets $s_j j \in \{1, m\}$. We create an equivalent recursive composition problem in the following manner. We create a composition problem with a composite service that requires n services, with one service corresponding to each element e_i in U. For each subset s_j, we create a provider with a cost function c_j equal to the weight w_j of the subset s_j. We then solve the Recursive Composition Problem with no delay bound and obtain the least cost solution CS to provide all the n services. It is easy to verify that the subsets corresponding to each selected provider provide a solution to the set cover problem. Further, one can show by contradiction that if this solution is not optimal for the set cover problem, then the solution CS is not optimal for the Recursive Composition problem. Hence, the set cover solution corresponding to CS is the optimal solution. This proves the result.

Theorem 3. *If the Goodness metric G is implemented as a heap ordered tree of depth 2 with all providers as leaf nodes each basic service as an intermediate node, the running time of BGF is given by $O(Mn\max\{M, n\})$.*

Proof. In order to prove Theorem 4, consider Fig. 1(b) and focus on the basic service providers. Observe that selecting one provider only effects the goodness values of the providers that provide the same service. Hence, we need to perform only n updates in each iteration. Further, *extractBest* needs to find the maximum of only M intermediate nodes, leading to the bound.

Lemma 4. *The number of disjoint sets with at least one service provider is $O(M)$.*

Proof. The proof is by induction.
Base Case: $M = 2$. It is easy to see that the number of disjoint service provider subsets is bounded by 3. Hence, the inductive hypothesis holds for 2.
Inductive Hypothesis: Let the above be true for all $M \leq k$.
Case $M = k + 1$: Let us look at two sub-cases. In the first case, let there be no provider who provides all the M services. We then divide the set of service providers into two disjoint sets. One may note that two such disjoint sets exist by picking the provider P_i with the largest $|os_i|$ as one disjoint set and the remaining basic services as the other set. It is easy to see (by the definition of Disjoint Set Composition) that no provider exists that provides services that span these two sets. Let the number of basic services in each of these two disjoint sets be $k1$ and $k2$ respectively. Hence, the total number of disjoint subsets is bounded by $O(k1) + O(k2)$ by the inductive hypothesis. This completes the proof for this case.
For the case where a provider exists who provides all the $k+1$ services, we create two disjoint sets by selecting basic services provided by the provider P_i with the second largest os_i as one of the two sets. A similar analysis leads to a bound of $O(k1) + O(k2) + 1$, which is O(M) again. This completes the proof.

Lemma 5. *If the solution S achieved by BGF at any step has a delay D, then S is the minimum cost solution which has a delay less than or equal to D.*
Proof. The proof is obtained by induction.
Observe that in the base case, we have the minimum delay solution possible. Hence, this is the only available eligible solution for that particular delay. Hence, the theorem holds for the base case.
Assume that the property holds after k steps and *BGF-D* select a new provider P_i with goodness g_i for the $k + 1^{th}$ swap. Also assume that the new provider increases the delay bound by d_i and decreases the cost by c_i. Let there be another provider P_i' that would have provided a better solution within the same delay bound. Let P_i' increase the delay by d_i' and decreases the cost by c_i'. In order for P_i' to be better than P_i, we require it to lead to a lower increase in delay (i.e. $d_i' < d_i$) and a larger reduction in cost ($c_i' > c_i$). By the definition of goodness property, $\frac{c_i}{d_i} > \frac{c_i'}{d_i'}, \forall$ provider P_i'. This leads to a contradiction and completes the proof.